Lecture 5: Memory Hierarchy and Cache (Continued)

Cache: A safe place for hiding and storing things.
Webster's New World Dictionary (1976)

Jack Dongarra
University of Tennessee

1

Optimizing Matrix Addition for Caches

- Dimension A(n,n), B(n,n), C(n,n)
- A, B, C stored by column (as in Fortran)
- Algorithm 1:
  - for i=1:n, for j=1:n, A(i,j) = B(i,j) + C(i,j)
- Algorithm 2:
  - for j=1:n, for i=1:n, A(i,j) = B(i,j) + C(i,j)
- What is "memory access pattern" for Algs 1 and 2?
- Which is faster?
- What if A, B, C stored by row (as in C)?

2

Using a Simpler Model of Memory to Optimize

- Assume just 2 levels in the hierarchy, fast and slow
- All data initially in slow memory
  - m = number of memory elements (words) moved between fast and slow memory
  - tm = time per slow memory operation
  - f = number of arithmetic operations
  - tf = time per arithmetic operation < tm
  - q = f/m average number of flops per slow element access
- Minimum possible Time = f*tf, when all data in fast memory
- Actual Time = f*tf + m*tm = f*tf*(1 + (tm/tf)*(1/q))
- Larger q means Time closer to minimum f*tf

3

Simple example using memory model

- To see results of changing q, consider simple computation

```
s = 0
for i = 1, n
    s = s + h(X[i])
```

- Assume tf=1 Mflop/s on fast memory
- Assume moving data is tm = 10
- Assume h takes q flops
- Assume array X is in slow memory

- So m = n and f = q*n
- Time = read X + compute = 10*n + q*n
- Mflop/s = f/t = q/(10 + q)
- As q increases, this approaches the "peak" speed of 1 Mflop/s

4

Simple Example (continued)

- Algorithm 1

```
s1 = 0;  s2 = 0
for j = 1 to n
    s1 = s1+h1(X(j))
    s2 = s2 + h2(X(j))
```

° Algorithm 2

```
s1 = 0; s2 = 0
for j = 1 to n
    s1 = s1 + h1(X(j))
for j = 1 to n
    s2 = s2 + h2(X(j))
```

° Which is faster?

5

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      d[i][j] = a[i][j] + c[i][j];
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {   a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality

6

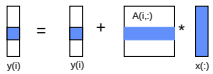
Optimizing Matrix Multiply for Caches

- Several techniques for making this faster on modern processors
  - heavily studied
- Some optimizations done automatically by compiler, but can do much better
- In general, you should use optimized libraries (often supplied by vendor) for this and other very common linear algebra operations
  - BLAS = Basic Linear Algebra Subroutines
- Other algorithms you may want are not going to be supplied by vendor, so need to know these techniques

7

Warm up: Matrix-vector multiplication y = y + A*x

```
for i = 1:n
        for j = 1:n
                y(i) = y(i) + A(i,j)*x(j)
```



8

Warm up: Matrix-vector multiplication y = y + A*x

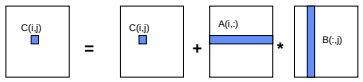
```
{read x(1:n) into fast memory}
{read y(1:n) into fast memory}
for i = 1:n
      {read row i of A into fast memory}
      for j = 1:n
            y(i) = y(i) + A(i,j)*x(j)
{write y(1:n) back to slow memory}
```

° m = number of slow memory refs = 3*n + n^2
° f = number of arithmetic operations = 2*n^2
° q = f/m ~= 2
° Matrix-vector multiplication limited by slow memory speed

9

Matrix Multiply C=C+A*B

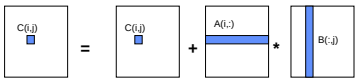
```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



10

Matrix Multiply C=C+A*B(unblocked, or untiled)

```
for i = 1 to n
  {read row i of A into fast memory}
  for j = 1 to n
    {read C(i,j) into fast memory}
    {read column j of B into fast memory}
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    {write C(i,j) back to slow memory}
```

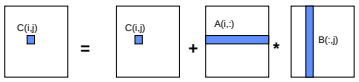


11

Matrix Multiply (unblocked, or untiled)

q=ops/slow mem ref

Number of slow memory references on unblocked matrix multiply
m = n^3 read each column of B n times
+ n^2 read each column of A once for each i
+ 2*n^2 read and write each element of C once
= n^3 + 3*n^2
So q = f/m = (2*n^3)/(n^3 + 3*n^2)
~= 2 for large n, no improvement over matrix-vector mult

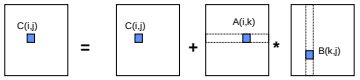


12

Matrix Multiply (blocked, or tiled)

Consider A,B,C to be N by N matrices of b by b subblocks where b=n/N is called the blocksize

```
for i = 1 to N
    for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```



13

Matrix Multiply (blocked or tiled)

q=ops/slow mem ref

Why is this algorithm correct?

Number of slow memory references on blocked matrix multiply
m = $N*n^2$ read each block of B N^3 times ($N^3 * n/N * n/N$)
+ $N*n^2$ read each block of A N^3 times
+ $2*n^2$ read and write each block of C once
= $(2*N + 2)*n^2$
So q = f/m = $2*n^3 / ((2*N + 2)*n^2)$
~= n/N = b for large n

So we can improve performance by increasing the blocksize b
Can be much faster than matrix-vector multiplty (q=2)

Limit: All three blocks from A,B,C must fit in fast memory (cache), so we cannot make these blocks arbitrarily large: $3*b^2 <= M$, so q ~= b <= sqrt(M/3)

Theorem (Hong, Kung, 1981): Any reorganization of this algorithm (that uses only associativity) is limited to q =O(sqrt(M))

14

Model

- As much as possible will be overlapped
- Dot Product:

```
ACC = 0
do i = x,n
  ACC = ACC + x(i) y(i)
end do
```

- Experiments done on an IBM RS6000/530
  - 25 MHz
  - 2 cycle to complete FMA can be pipelined
    - » => 50 Mflop/s peak
  - one cycle from cache

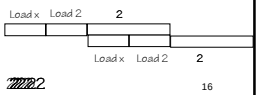
15

DOT Operation - Data in Cache

```
Do 10 I = 1, n
   T = T + X(I)*Y(I)
10 CONTINUE
```

- Theoretically, 2 loads for X(I) and Y(I), one FMA operation, no re-use of data
- Pseudo-assembler

```
LOAD   fp0,T
label:
LOAD   fp1,X(I)
LOAD   fp2,Y(I)
FMA    fp0,fp0,fp1,fp2
BRANCH label:
```



16

Matrix-Vector Product

- DOT version

```
DO 20 I = 1, M
    DO 10 J = 1, N
        Y(I) = Y(I) + A(I,J)*X(J)
10    CONTINUE
20 CONTINUE
```

- From Cache = 22.7 Mflops
- From Memory = 12.4 Mflops

17

Loop Unrolling

```
DO 20 I = 1, M, 2
   T1 = Y(I  )
   T2 = Y(I+1)
   DO 10 J = 1, N
     T1 = T1 + A(I,J  )*X(J)
     T2 = T2 + A(I+1,J)*X(J)
10 CONTINUE
   Y(I  ) = T1
   Y(I+1) = T2
20 CONTINUE
```

- 3 loads, 4 flops
- Speed of $y=y+A^Tx$, N=48

Depth	1	2	3	4	∞
Speed	25	33.3	37.5	40	50
Measured	22.7	30.5	34.3	36.5	
Memory	12.4	12.7	12.7	12.6	

- unroll 1: 2 loads : 2 ops per 2 cycles
- unroll 2: 3 loads : 4 ops per 3 cycles
- unroll 3: 4 loads : 6 ops per 4 cycles
- ...
- unroll n: n+1 loads : 2n ops per n+1 cycles
- problem: only so many registers

18

Matrix Multiply

- DOT version - 25 Mflops in cache

```
      DO 30 J = 1, M
         DO 20 I = 1, M
            DO 10 K = 1, L
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
10          CONTINUE
20       CONTINUE
30    CONTINUE
```

19

How to Get Near Peak

```
      DO 30 J = 1, M, 2
         DO 20 I + 1, M, 2
            T11 = C(I,  J )
            T12 = C(I,  J+1)
            T21 = C(I+1,J )
            T22 = C(I+1,J+1)
            DO 10 K = 1, L
               T11 = T11 + A(I,  K) *B(K,J )
               T12 = T12 + A(I,  K) *B(K,J+1)
               T21 = T21 + A(I+1,K)*B(K,J )
               T22 = T22 + A(I+1,K)*B(K,J+1)
10          CONTINUE
            C(I,  J ) = T11
            C(I,  J+1) = T12
            C(I+1,J ) = T21
            C(I+1,J+1) = T22
20       CONTINUE
30    CONTINUE
```

- Inner loop:
  - 4 loads, 8 operations, optimal.
- In practice we have measured 48.1 out of a peak of 50 Mflop/s when in cache

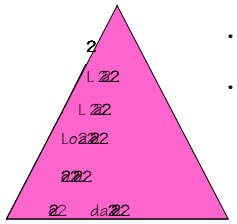
20

BLAS -- Introduction

- Clarity: code is shorter and easier to read,
- Modularity: gives programmer larger building blocks,
- Performance: manufacturers will provide tuned machine-specific BLAS,
- Program portability: machine dependencies are confined to the BLAS

21

Memory Hierarchy

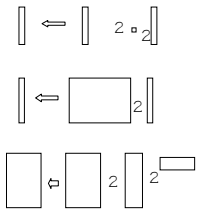


- Key to high performance in effective use of memory hierarchy
- True on all architectures

22

Level 1, 2 and 3 BLAS

- Level 1 BLAS Vector-Vector operations
- Level 2 BLAS Matrix-Vector operations
- Level 3 BLAS Matrix-Matrix operations



23

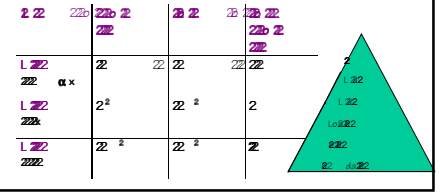
More on BLAS (Basic Linear Algebra Subroutines)

- Industry standard interface(evolving)
- Vendors, others supply optimized implementations
- History
  - BLAS1 (1970s):
    - » vector operations: dot product, saxpy ($y=\alpha*x+y$), etc
    - » $m=2*n$, $f=2*n$, q ~1 or less
  - BLAS2 (mid 1980s)
    - » matrix-vector operations: matrix vector multiply, etc
    - » $m=n^2$, $f=2*n^2$, q~2, less overhead
    - » somewhat faster than BLAS1
  - BLAS3 (late 1980s)
    - » matrix-matrix operations: matrix matrix multiply, etc
    - » $m >= 4n^2$, $f=O(n^3)$, so q can possibly be as large as n, so BLAS3 is potentially much faster than BLAS2
- Good algorithms used BLAS3 when possible (LAPACK)
- www.netlib.org/blas, www.netlib.org/lapack

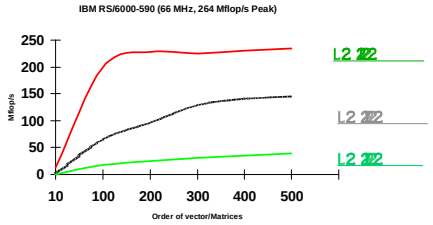
24

Why Higher Level BLAS?

- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

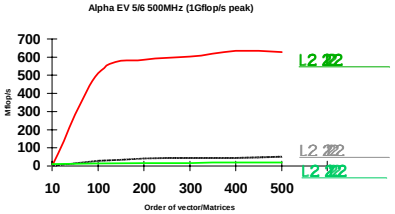


BLAS for Performance



- Development of blocked algorithms important for performance

BLAS for Performance



- Development of blocked algorithms important for performance

BLAS 3 (n-by-n matrix matrix multiply) vs
BLAS 2 (n-by-n matrix vector multiply) vs
BLAS 1 (saxpy of n vectors)

Fast linear algebra kernels: BLAS

Level 1 BLAS

Level 2 BLAS

Level 3 BLAS

- [illegible]
  - [illegible]
  - [illegible] 3[illegible]
- [illegible]
  - [illegible]
  - [illegible]
- [illegible]
  - [illegible]
  - [illegible]

Optimizing in practice

- Tiling for registers
  - loop unrolling, use of named "register" variables
- Tiling for multiple levels of cache
- Exploiting fine-grained parallelism within the processor
  - super scalar
  - pipelining
- Complicated compiler interactions
- Hard to do by hand (but you'll try)
- Automatic optimization an active research area
  - PHIPAC: www.icsi.berkeley.edu/~bilmes/phipac
  - www.cs.berkeley.edu/~iyer/asci_slides.ps
  - ATLAS: www.netlib.org/atlas/index.html

32

BLAS -- References

- **BLAS software and documentation can be obtained via:**
  - WWW: http://www.netlib.org/blas,
  - (anonymous) ftp ftp.netlib.org: cd blas; get index
  - email netlib@www.netlib.org with the message: send index from blas
- **Comments and questions can be addressed to:** lapack@cs.utk.edu

33

BLAS Papers

- C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic Linear Algebra Subprograms for Fortran Usage*, ACM Transactions on Mathematical Software, 5:308--325, 1979.
- J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson, *An Extended Set of Fortran Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14(1):1--32, 1988.
- J. Dongarra, J. Du Croz, I. Duff, S. Hammarling, *A Set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16(1):1--17, 1990.

34

Performance of BLAS

- [illegible]
  - [illegible]
- [illegible]
  - [illegible]
  - [illegible]
- [illegible]
- [illegible]
  - [illegible]

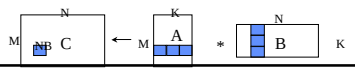
How To Get Performance From Commodity Processors?

- Today's processors can achieve high-performance, but this requires extensive machine-specific hand tuning.
- Routines have a large design space w/many parameters
  - blocking sizes, loop nesting permutations, loop unrolling depths, software pipelining strategies, register allocations, and instruction schedules.
  - Complicated interactions with the increasingly sophisticated microarchitectures of new microprocessors.
- A few months ago no tuned BLAS for Pentium for Linux.
- Need for quick/dynamic deployment of optimized routines.
- ATLAS - Automatic Tuned Linear Algebra Software
  - PhiPac from Berkeley

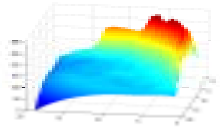
36

Adaptive Approach for Level 3

- Do a parameter study of the operation on the target machine, done once.
- Only generated code is on-chip multiply
- BLAS operation written in terms of generated on-chip multiply
- All tranpose cases coerced through data copy to 1 case of on-chip multiply
  - Only 1 case generated per platform

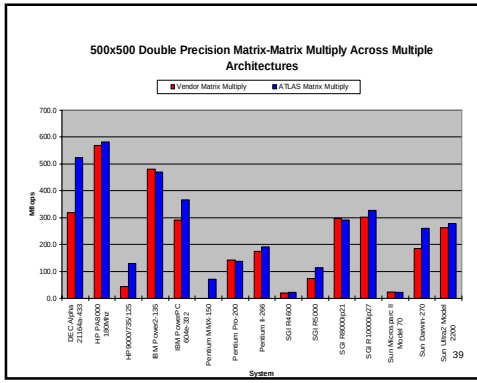


Code Generation Strategy

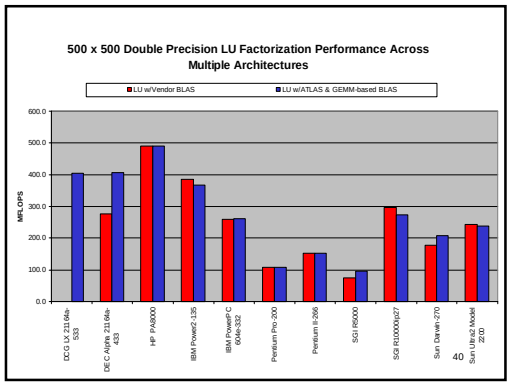


- On-chip multiply optimizes for:
  - TLB access
  - L1 cache reuse
  - FP unit usage
  - Memory fetch
  - Register reuse
  - Loop overhead minimization
- Takes a couple of hours to run.
- Code is iteratively generated & timed until optimal case is found. We try:
  - Differing NBs
  - Breaking false dependencies
  - M, N and K loop unrolling

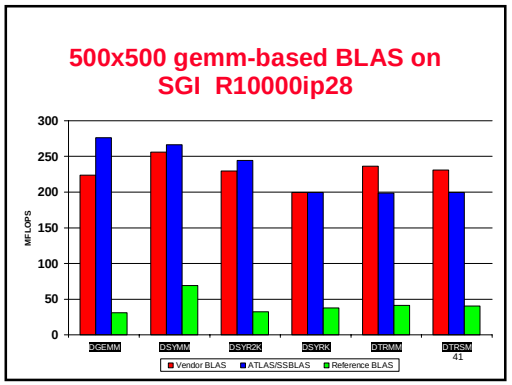
38



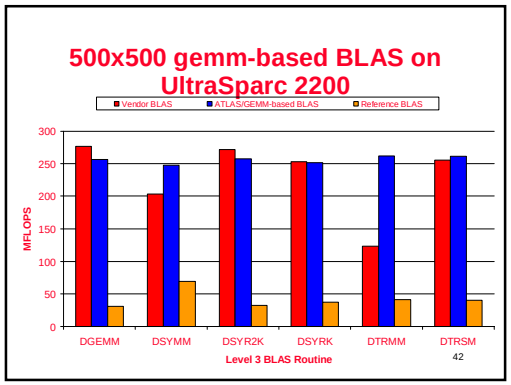
39



40



41

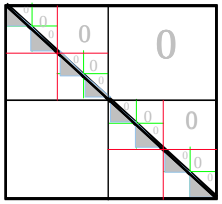


42

Recursive Approach for Other Level 3 BLAS

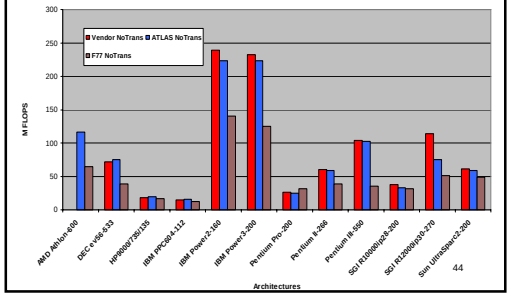
- Recur down to L1 cache block size
- Need kernel at bottom of recursion
  - Use gemm-based kernel for portability

Recursive TRMM



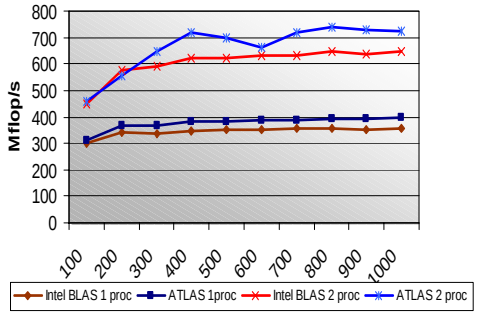
43

500x500 Level 2 BLAS DGEMV



44

Multi-Threaded DGEMM Intel PIII 550 MHz



ATLAS

- Keep a repository of kernels for specific machines.
- Develop a means of dynamically downloading code
- Extend work to allow sparse matrix operations
- Extend work to include arbitrary code segments
- See: http://www.netlib.org/atlas/

46

BLAS Technical Forum

http://www.netlib.org/utk/papers/blast-forum.html

- Established a Forum **to consider expanding the BLAS in light of modern software, language, and hardware developments.**
- Minutes available **from each meeting**
- Working proposals **for the following:**
  - Dense/Band BLAS
  - Sparse BLAS
  - Extended Precision BLAS
  - Distributed Memory BLAS
  - C and Fortran90 interfaces to Legacy BLAS

47

Strassen's Matrix Multiply

- The traditional algorithm (with or without tiling) has $O(n^3)$ flops
- Strassen discovered an algorithm with asymptotically lower flops
  - $O(n^{2.81})$
- Consider a 2x2 matrix multiply, normally 8 multiplies

Let M = [m11 m12] = [a11 a12] * [b11 b12]
[m21 m22] [a21 a22] [b21 b22]

Let p1 = (a12 - 122) * (b21 + b22) p5 = a11 * (b12 - b22)
p2 = (a11 + a22) * (b11 + b22) p6 = a22 * (b21 - b11)
p3 = (a11 - a21) * (b11 + b12) p7 = (a21 + a22) * b11
p4 = (a11 + a12) * b22

Then m11 = p1 + p2 - p4 + p6 Extends to nxn by divide&conquer
m12 = p4 + p5
m21 = p6 + p7
m22 = p2 - p3 + p5 - p7

48

Strassen (continued)

T(n) = Cost of multiplying nxn matrices
= 7*T(n/2) + 18*(n/2)^2
= O(n$^{log_2\ 7}$)
= O(n$^{2.81}$)

° Available in several libraries
° Up to several time faster if n large enough (100s)
° Needs more memory than standard algorithm
° Can be less accurate because of roundoff error
° Current world's record is O(n$^{2.376..}$)

Summary

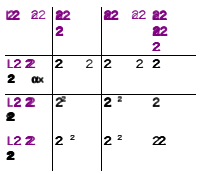
- Performance programming on uniprocessors requires
  - understanding of memory system
    - » levels, costs, sizes
  - understanding of fine-grained parallelism in processor to produce good instruction mix
- Blocking (tiling) is a basic approach that can be applied to many matrix algorithms
- Applies to uniprocessors and parallel processors
  - The technique works for any architecture, but choosing the blocksize b and other details depends on the architecture
- Similar techniques are possible on other data structures

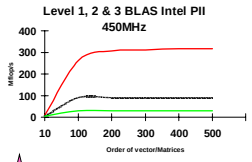
Summary: Memory Hierachy

- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is more important than memory hierarchy
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?

Performance = Effective Use of Memory Hierarchy

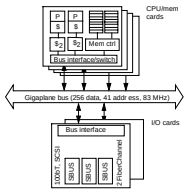
- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this





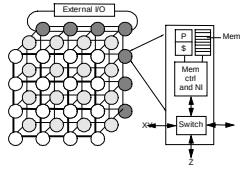
- Development of blocked algorithms important for performance

Engineering: SUN Enterprise



- Proc + mem card - I/O card
  - 16 cards of either type
  - All memory accessed over bus, so symmetric
  - Higher bandwidth, higher latency bus

Engineering: Cray T3E

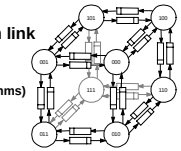


- Scale up to 1024 processors, 480MB/s links
- Memory controller generates request message for non-local references
- No hardware mechanism for coherence
  - » SGI Origin etc. provide this

Evolution of Message-Passing Machines

- Early machines: FIFO on each link
  - HW close to prog. Model;

gorithms)



CalTech Cosmic Cube (Seitz, CACM Jan 95)

Diminishing Role of Topology

- Shift to general links
  - DMA, enabling non-blocking ops
    - » Buffered by system at destination until recv
  - Store&forward routing
- Diminishing role of topology
  - Any-to-any pipelined routing
  - node-network interface dominates communication time

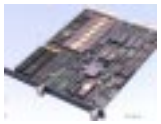
$H \times (T_0 + n/B)$

vs

$T_0 + HA + n/B$

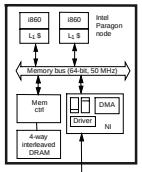
  - Simplifies programming
  - Allows richer design space
    - » grids vs hypercubes

Intel iPSC/1 -> iPSC/2 -> iPSC/860

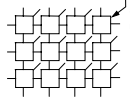


Example Intel Paragon

Sandia' s Intel Paragon XP/S-based Super computer



2D grid network with processing node attached to every switch

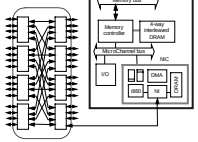


8 bits, 175 MHz, bidirectional

Building on the mainstream: IBM SP-2

- Made out of essentially complete RS6000 workstations
- Network interface integrated in I/O bus (bw limited by I/O bus)

General Inter connection network formed fr om 8-port switches



Berkeley NOW

- 100 Sun Ultra2 workstations
- Inteligent network interface
  - proc + mem
- Myrinet Network
  - 160 MB/s per link
  - 300 ns per hop

Thanks

- These slides came in part from courses taught by the following people:
  - Kathy Yelick, UC, Berkeley
  - Dave Patterson, UC, Berkeley
  - Randy Katz, UC, Berkeley
  - Craig Douglas, U of Kentucky
- Computer Architecture A Quantitative Approach, Chapter 8, Hennessy and Patterson, Morgan Kaufman Pub.