Vector Processing (Con’t)
Intro to Multiprocessing
March 8th, 2010
John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

Recall: Vector Programming Model
Scalar Registers
r15
v15
r0
[0] [1] [2] [VLRMAX-1]
Vector Registers
Vector Length Register VLR

Vector Arithmetic Instructions
ADDV v3, v1, v2

Vector Load and Store Instructions
LV v1, r1, r2

Recall: Vector Unit Structure
Elements 0, 4, 8, …
Elements 1, 5, 9, …
Elements 2, 6, 10, …
Elements 3, 7, 11, …

Lane
Vector Registers
Functional Unit
Memory Subsystem

Vector Memory-Memory vs. Vector Register Machines
• Vector memory-memory instructions hold all vector operands in main memory
• The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
• Cray-1 ('76) was first vector register machine

Example Source Code
for (i=0; i<N; i++)
{
    C[i] = A[i] + B[i];
    D[i] = A[i] - B[i];
}

Vector Memory-Memory Code
ADDV C, A, B
SUBV D, A, B

Vector Register Code
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMMAs make it difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar break-even point was around 2 elements

⇒ Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
   (we ignore vector memory-memory from now on)

Automatic Code Vectorization

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

```
Scalar Sequential Code

<table>
<thead>
<tr>
<th>Iter. 1</th>
<th>Iter. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>load</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
</tr>
<tr>
<td>store</td>
<td>store</td>
</tr>
</tbody>
</table>
```

```
Vectorized Code

<table>
<thead>
<tr>
<th>Vector Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
</tr>
<tr>
<td>add</td>
</tr>
<tr>
<td>store</td>
</tr>
</tbody>
</table>
```

Vectorization is a massive compile-time reordering of operation sequencing
requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length
Solution: Break loops into pieces that fit into vector registers, “Stripmining”

```
for (i=0; i<N; i++)
    C[i] = A[i] + B[i];
```

```
Load Unit Multiply Unit Add Unit
```

```
Complete 24 operations/cycle while issuing 1 short instruction/cycle
```

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions
- Example machine has 32 elements per vector register and 8 lanes

```
Load Unit
```

```
Multiply Unit
```

```
Add Unit
```

```
Instruction issue
```

```
``
Vector Chaining
• Vector version of register bypassing
  – introduced with Cray-1

Vector Chaining Advantage
• Without chaining, must wait for last element of result to be written before starting dependent instruction
• With chaining, can start dependent instruction as soon as first result appears

Vector Startup
Two components of vector startup penalty
– functional unit latency (time through pipeline)
– dead time or recovery time (time before another vector instruction can start down pipeline)

Dead Time and Short Vectors
Cray C90, Two lanes
4 cycle dead time
Maximum efficiency 94%
with 128 element vectors

T0, Eight lanes
No dead time
100% efficiency with 8 element vectors
Vector Scatter/Gather

Want to vectorize loops with indirect accesses:
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]

Indexed load instruction (Gather)

```
LV vD, rD # Load indices in D vector
LVI vc, rC, vD # Load indirect from rC base
LV vB, rB # Load B vector
ADDV.D vA, vB, vC # Do add
SV vA, rA # Store result
```

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:
```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector mask (or flag) registers
– vector version of predicate registers, 1 bit per element
...and maskable vector instructions
– vector operation becomes NOP at elements where mask bit is clear

Code example:
```
CVM # Turn on all elements
LV va, rA # Load entire A vector
SGTVD.D va, F0 # Set bits in mask register where A>0
LV va, rB # Load B vector into A under mask
SV va, rA # Store A back to memory under mask
```

Masked Vector Instructions

Simple Implementation
– execute all N operations, turn off result writeback according to mask

| M[0] | A[0] | B[0] |

Write Enable

Density-Time Implementation
– scan mask vector and only execute elements with non-zero masks

| M[0] | A[0] | B[0] |

Write Enable

Compress/Expand Operations

• Compress packs non-masked elements from one vector register contiguously at start of destination vector register
  – population count of mask vector gives packed vector length

• Expand performs inverse operation

| M[0] | A[0] | B[0] |

Compress

Expand

Used for density-time conditionals and also for general selection operations
**Administrivia**

- Exam: one week from Wednesday (3/17)
  - Location: 310 Soda
  - TIME: 6:00-9:00
  - This info is on the Lecture page (has been)
  - Get on 8 ½ by 11 sheet of notes (both sides)
  - Meet at LaVal’s afterwards for Pizza and Beverages

- I have your proposals.
  - We need to meet to discuss them
  - Time this week? Wednesday after class

**Vector Reductions**

Problem: Loop-carried dependence on reduction variables

```c
sum = 0;
for (i=0; i<N; i++)
  sum += A[i];  // Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```c
# Rearrange as:
sum[0:VL-1] = 0                 // Vector of VL partial sums
for(i=0; i<N; i+=VL)            // Stripmine VL-sized chunks
  sum[0:VL-1] += A[i:i+VL-1]; // Vector sum
# Now have VL partial sums in one vector register
do {
  VL = VL/2;                    // Halve vector length
  sum[0:VL-1] += sum[VL:2*VL-1] // Halve no. of partials
} while (VL>1)
```

**Novel Matrix Multiply Solution**

- Consider the following:
  ```c
  /* Multiply a[m][k] * b[k][n] to get c[m][n] */
  for (i=1; i<m; i++) {
    for (j=1; j<n; j++) {
      sum = 0;
      for (t=1; t<k; t++)
        sum += a[i][t] * b[t][j];
      c[i][j] = sum;
    }
  }
  ```

- Do you need to do a bunch of reductions? NO!
  - Calculate multiple independent sums within one vector register
  - You can vectorize the j loop to perform 32 dot-products at the same time (Assume Max Vector Length is 32)

- Show it in C source code, but can imagine the assembly vector instructions from it

**Optimized Vector Example**

```c
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
  for (j=1; j<n; j+=32) {/* Step j 32 at a time. */
    sum[0:31] = 0; /* Init vector reg to zeros. */
    for (t=1; t<k; t++) {
      a_scalar = a[i][t]; /* Get scalar */
      b_vector[0:31] = b[t][j:j+31]; /* Get vector */
      /* Do a vector-scalar multiply. */
      prod[0:31] = b_vector[0:31]*a_scalar;
      /* Vector-vector add into results. */
      sum[0:31] += prod[0:31];
    }/* Unit-stride store of vector of results. */
    c[i][j:j+31] = sum[0:31];
  }
}
```
Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2)
- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
  - requires superscalar dispatch to keep multiply/add/load units busy
  - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors

“Vector” for Multimedia?

- Intel MMX: 57 additional 80x86 instructions (1st since 386)
  - similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC
- 3 data types: 8 8-bit, 4 16-bit, 2 32-bit in 64bits
  - reuse 8 FP registers (FP and MMX cannot mix)
- Short vector: load, add, store 8 8-bit operands
- Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...
  - use in drivers or added to library routines; no compiler

VLIW: Very Large Instruction Word (revisited)

- Each “instruction” has explicit coding for multiple operations
  - In IA-64, grouping called a “packet”
  - In Transmeta, grouping called a “molecule” (with “atoms” as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  - Need compiling technique that schedules across several branches

Recall: Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td>ADD.D F2,F0,F2</td>
<td>ADD.D F4,F6,F2</td>
<td>DSUBUI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td>BNEZ R1,LOOP</td>
<td>9</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D -8(R1),F8</td>
<td>ADD.D F20,F24,F26,F26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -40(R1),F24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D -0(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SS)
Problems with 1st Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA-64: instruction set architecture
  - 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW
  - Hardware checks dependencies
    (interlocks \(\Rightarrow\) binary compatibility over time)
- 3 Instructions in 128 bit “bundles”; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence \(\Rightarrow\) 3 instr
- Predicated execution (select 1 out of 64 1-bit flags) \(\Rightarrow\) 40% fewer mispredictions?
- Speculation Support:
  - deferred exception handling with “poison bits”
  - Speculative movement of stores above stores + check to see if incorrect
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 \(\mu\) process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 \(\mu\) process
  - Caches: 32 KB I, 32 KB D, 128 KB L2i, 128 KB L2d, 9216 KB L3
10 Stage In-Order Core Pipeline

What is Parallel Architecture?

- A parallel computer is a collection of processing elements that cooperate to solve large problems
  - Most important new element: It is all about communication!
- What does programmer (or OS or Compiler) think about?
  - Models of computation:
    » PRAM? BSP? Sequential Consistency?
  - Resource Allocation:
    » how powerful are the elements?
    » how much memory?
- What mechanisms must be in hardware vs software
  - What does a single processor look like?
    » High performance general purpose processor
    » SIMD processor/Vector Processor
  - Data access, Communication and Synchronization
    » how do the elements cooperate and communicate?
    » how are data transmitted between processors?
    » what are the abstractions and primitives for cooperation?

Flynn’s Classification (1966)

- SISD: Single Instruction, Single Data
  - conventional uniprocessor
- SIMD: Single Instruction, Multiple Data
  - one instruction stream, multiple data paths
  - distributed memory SIMD (MPP, DAP, CM-1&2, Maspar)
  - shared memory SIMD (STARAN, vector computers)
- MIMD: Multiple Instruction, Multiple Data
  - message passing machines (Transputers, nCube, CM-5)
  - non-cache-coherent shared memory machines (BBN Butterfly, T3D)
  - cache-coherent shared memory machines (Sequent, Sun Starfire, SGI Origin)
- MISD: Multiple Instruction, Single Data
  - Not a practical configuration

Examples of MIMD Machines

- Symmetric Multiprocessor
  - Multiple processors in box with shared memory communication
  - Current MultiCore chips like this
  - Every processor runs copy of OS
- Non-uniform shared-memory with separate I/O through host
  - Multiple processors
    » Each with local memory
    » general scalable network
  - Extremely light “OS” on node provides simple services
    » Scheduling/synchronization
  - Network-accessible host for I/O
- Cluster
  - Many independent machine connected with general network
  - Communication through messages
**Categories of Thread Execution**

<table>
<thead>
<tr>
<th>Time (processor cycle)</th>
<th>Superscalar</th>
<th>Fine-Grained</th>
<th>Coarse-Grained</th>
<th>Multiprocessing</th>
<th>Multithreading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simultaneous</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Parallel Programming Models**

- **Programming model** is made up of the languages and libraries that create an abstract view of the machine
  - **Control**
    - How is parallelism created?
    - What orderings exist between operations?
    - How do different threads of control synchronize?
  - **Data**
    - What data is private vs. shared?
    - How is logically shared data accessed or communicated?
  - **Synchronization**
    - What operations can be used to coordinate parallelism
    - What are the atomic (indivisible) operations?
  - **Cost**
    - How do we account for the cost of each of the above?

**Simple Programming Example**

- Consider applying a function \( f \) to the elements of an array \( A \) and then computing its sum:
  \[
  \sum_{i=0}^{n-1} f(A[i])
  \]

  - **Questions:**
    - Where does \( A \) live? All in single memory? Partitioned?
    - What work will be done by each processors?
    - They need to coordinate to get a single result, how?

  

  \[
  A = \text{array of all data}
  \]
  \[
  fA = f(A)
  \]
  \[
  s = \text{sum}(fA)
  \]

**Programming Model 1: Shared Memory**

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables.
Simple Programming Example: SM

• Shared memory strategy:
  – small number p << n=size(A) processors
  – attached to single memory

• Parallel Decomposition:
  – Each evaluation and each partial sum is a task.
• Assign n/p numbers to each of p procs
  – Each computes independent “private” results and partial sum.
  – Collect the p partial sums and compute a global sum.

Two Classes of Data:
• Logically Shared
  – The original n numbers, the global sum.
• Logically Private
  – The individual function evaluations.
  – What about the individual partial sums?

\[ \sum_{i=0}^{n-1} f(A[i]) \]

Shared Memory “Code” for sum

\[ \text{Thread 1} \]
\[ \text{for } i = 0, n/2-1 \]
\[ s = s + f(A[i]) \]

\[ \text{Thread 2} \]
\[ \text{for } i = n/2, n-1 \]
\[ s = s + f(A[i]) \]

• Problem is a race condition on variable s in the program
• A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

A Closer Look

\[ A = [3,5] \]  \( f = \text{square} \)
\[ \text{static int s = 0;} \]

\[ \text{Thread 1} \]
\[ \text{local_s1= 0} \]
\[ \text{for } i = 0, n/2-1 \]
\[ \text{local_s1 = local_s1 + f(A[i])} \]
\[ \text{lock(lk)}; \]
\[ \text{s = s + local_s1} \]
\[ \text{unlock(lk)}; \]

\[ \text{Thread 2} \]
\[ \text{local_s2= 0} \]
\[ \text{for } i = n/2, n-1 \]
\[ \text{local_s2 = local_s2 + f(A[i])} \]
\[ \text{lock(lk)}; \]
\[ \text{s = s + local_s2} \]
\[ \text{unlock(lk)}; \]

• Since addition is associative, it’s OK to rearrange order
• Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

Improved Code for Sum

\[ \text{Thread 1} \]
\[ \text{local_s1= 0} \]
\[ \text{for } i = 0, n/2-1 \]
\[ \text{local_s1 = local_s1 + f(A[i])} \]
\[ \text{lock(lk)}; \]
\[ \text{s = s + local_s1} \]
\[ \text{unlock(lk)}; \]

\[ \text{Thread 2} \]
\[ \text{local_s2= 0} \]
\[ \text{for } i = n/2, n-1 \]
\[ \text{local_s2 = local_s2 + f(A[i])} \]
\[ \text{lock(lk)}; \]
\[ \text{s = s + local_s2} \]
\[ \text{unlock(lk)}; \]

• Assume A = [3,5], f is the square function, and s=0 initially
• For this program to work, s should be 34 at the end
  - but it may be 34,9, or 25
• The atomic operations are reads and writes
  - Never see ½ of one number, but += operation is not atomic
  - All computations happen in (private) registers
What about Synchronization?

- All shared-memory programs need synchronization
- Barrier – global (coordinated) synchronization
  - simple use of barriers -- all threads hit the same one
    ```c
    work_on_my_subgrid();
    barrier;
    read_neighboring_values();
    barrier;
    ```
- Mutexes – mutual exclusion locks
  - threads are mostly independent and must access common data
    ```c
    lock *l = alloc_and_init(); /* shared */
    lock(l);
    access_data
    unlock(l);
    ```
- Need atomic operations bigger than loads/stores
  - Actually – Dijkstra's algorithm can get by with only loads/stores, but
    this is quite complex (and doesn’t work under all circumstances)
  - Example: atomic swap, test-and-test-and-set
- Another Option: Transactional memory
  - Hardware equivalent of optimistic concurrency
  - Some think that this is the answer to all parallel programming

Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW


° First possible solution – what could go wrong?

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>send xlocal, proc2</td>
<td>send xlocal, proc1</td>
</tr>
<tr>
<td>receive xremote, proc2</td>
<td>receive xremote, proc1</td>
</tr>
<tr>
<td>s = xlocal + xremote</td>
<td>s = xlocal + xremote</td>
</tr>
</tbody>
</table>

° If send/receive acts like the telephone system? The post office?

° Second possible solution

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>send xlocal, proc2</td>
<td>send xlocal, proc1</td>
</tr>
<tr>
<td>receive xremote, proc2</td>
<td>receive xremote, proc1</td>
</tr>
<tr>
<td>s = xlocal + xremote</td>
<td>s = xlocal + xremote</td>
</tr>
</tbody>
</table>

° What if there are more than 2 processors?

MPI – the de facto standard

° MPI has become the de facto standard for parallel computing using message passing
° Example:

```c
for(i=1;i<numprocs;i++) {
    sprintf(buff, "Hello \%d ! ", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG,
        MPI_COMM_WORLD);
}
for(i=1;i<numprocs;i++) {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG,
        MPI_COMM_WORLD, &stat);
    printf("\%d: \%s\n", myid, buff);
}
```
° Pros and Cons of standards
  - MPI created finally a standard for applications development in the HPC community → portability
  - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation
Which is better? SM or MP?

- Which is better, Shared Memory or Message Passing?
  - Depends on the program!
  - Both are “communication Turing complete”
    - i.e. can build Shared Memory with Message Passing and vice-versa

Advantages of Shared Memory:
- Implicit communication (loads/stores)
- Low overhead when cached

Disadvantages of Shared Memory:
- Complex to build in way that scales well
- Requires synchronization operations
- Hard to control data placement within caching system

Advantages of Message Passing
- Explicit Communication (sending/receiving of messages)
- Easier to control data placement (no automatic caching)

Disadvantages of Message Passing
- Message passing overhead can be quite high
- More complex to program
- Introduces question of reception technique (interrupts/polling)

What characterizes a network?

- Topology (what)
  - physical interconnection structure of the network graph
  - direct: node connected to every switch
  - indirect: nodes connected to specific subset of switches

- Routing Algorithm (which)
  - restricts the set of paths that msgs may follow
  - many algorithms with different properties
    - gridlock avoidance?

- Switching Strategy (how)
  - how data in a msg traverses a route
  - circuit switching vs. packet switching

- Flow Control Mechanism (when)
  - when a msg or portions of it traverse a route
  - what happens when traffic is encountered?

Example: Multidimensional Meshes and Tori

- $n$-dimensional array
  - $N = k_{d-1} \times \ldots \times k_0$ nodes
  - described by $n$-vector of coordinates $(i_{n-1}, \ldots, i_0)$

- $n$-dimensional $k$-ary mesh: $N = k^n$
  - $k = \sqrt[n]{N}$
  - described by $n$-vector of radix $k$ coordinate

- $n$-dimensional $k$-ary torus (or $k$-ary $n$-cube)?

Links and Channels

- transmitter converts stream of digital symbols into signal that is driven down the link
- receiver converts it back
  - tran/rcv share physical protocol
- trans + link + rcv form Channel for digital info flow between switches
- link-level protocol segments stream of symbols into larger units: packets or messages (framing)
- node-level protocol embeds commands for dest communication assist within packet
**Clock Synchronization?**

- Receiver must be synchronized to transmitter
  - To know when to latch data
- **Fully Synchronous**
  - Same clock and phase: Isochronous
  - Same clock, different phase: Mesochronous
    » High-speed serial links work this way
    » Use of encoding (8B/10B) to ensure sufficient high-frequency component for clock recovery
- **Fully Asynchronous**
  - No clock: Request/Ack signals
  - Different clock: Need some sort of clock recovery?

![Diagram of data transmission](image)

**Conclusion**

- Vector is alternative model for exploiting ILP
  - If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
  - Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- **VLIW: Explicitly Parallel**
  - Trace Scheduling: Select primary “trace” to compress + fixup code
- Itanium/EPIC/VLIW is not a breakthrough in ILP
  - If anything, it is as complex or more so than a dynamic processor
- **Multiprocessing**
  - Multiple processors connect together
  - It is all about communication!
- **Programming Models:**
  - Shared Memory
  - Message Passing
- **Networking and Communication Interfaces**
  - Fundamental aspect of multiprocessing