CS 612

Software Systems for High-performance Architectures

Course Organization

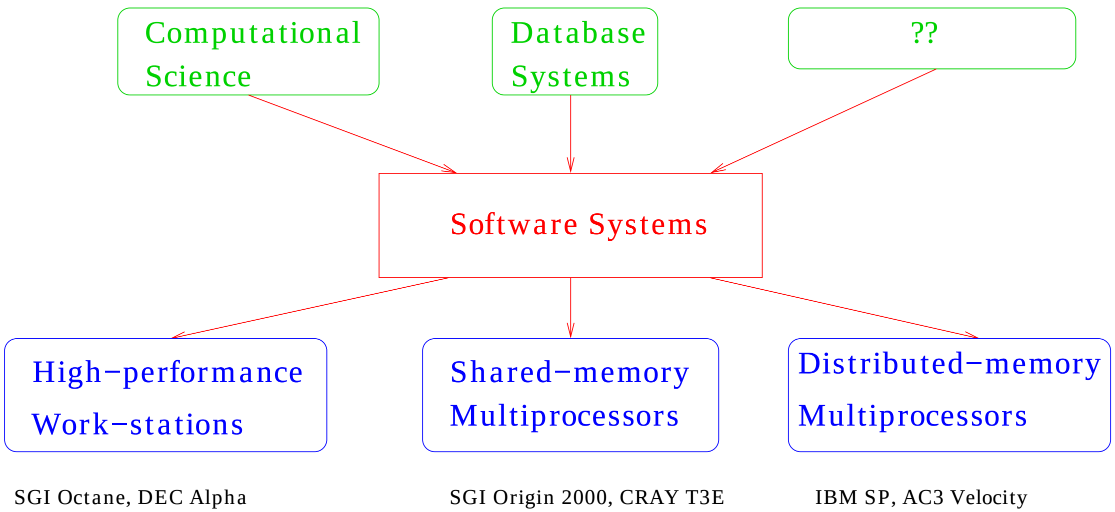
- Lecturer:Paul Stodghill, stodghil@cs.cornell.edu, Rhodes 496
- TA: Rohit Fernandes, rohitf@cs.cornell.edu, Rhodes 490
- URL: `http://www.cs.cornell.edu/Courses/cs612/2002SP/`
- Prerequisites: Experience in writing moderate-sized (about 2000 lines) programs, and interest in software for high-performance computers. CS 412 is desirable but not essential.
- Lectures: two per week
- Course-work: Four or five assignments which will involve programming on work-stations, and a substantial final project.

Resources

- Books (Recommended, not required)
  - "Advanced Compiler Design and Implementation", Steve Muchnick, Morgan Kaufmann Publishers.
  - "Introduction to Parallel Computing", Vipin Kumar et al, Benjamin/Cummings Publishers.
  - "Computer Architecture: A Quantitative Approach", Hennessy and Patterson, Morgan Kaufmann Publishers.
- Conferences
  - "ACM Symposium on Principles and Practice of Parallel Programming"
  - "ACM SIGPLAN Symposium on Programming Language Design and Implementation"
  - "International Conference on Supercomputing"
  - "Supercomputing"

Objective

We will study software systems that permit applications programs to exploit the power of modern high−performance computers.



− some emphasis on applications and architecture

− primary emphasis on restructuring compilers, parallel languages (HPF), and libraries (OpenMP, MPI).

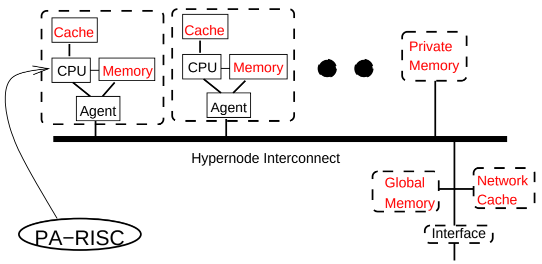
Conventional Software Environment

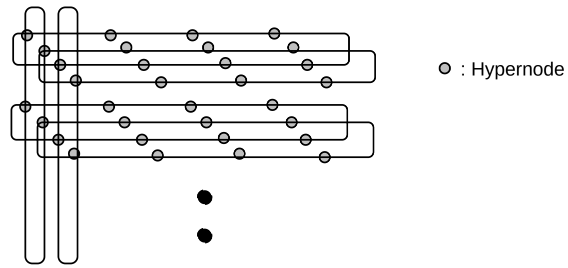
- Languages: FORTRAN,C/C++, Java
- Compiler: GNU (Dragon-book optimizations)
- O/S: UNIX, Win32

This software environment is not adequate for modern high-performance computers.

To understand this, let us look at some high-performance computers.

The HP/CONVEX Exemplar: A Shared−Memory MultiProcessor





Hypernode

Network of hypernodes

Parallelism:

Coarse−grain parallelism: processors operate in parallel
Instruction−level parallelism: each processor is pipelined

Memory latencies:

Processor cache 10 ns
CPU private memory 500 ns
Hypernode private memory 500 ns
Network cache 500 ns
Interhypernode shared memory 2 microsec

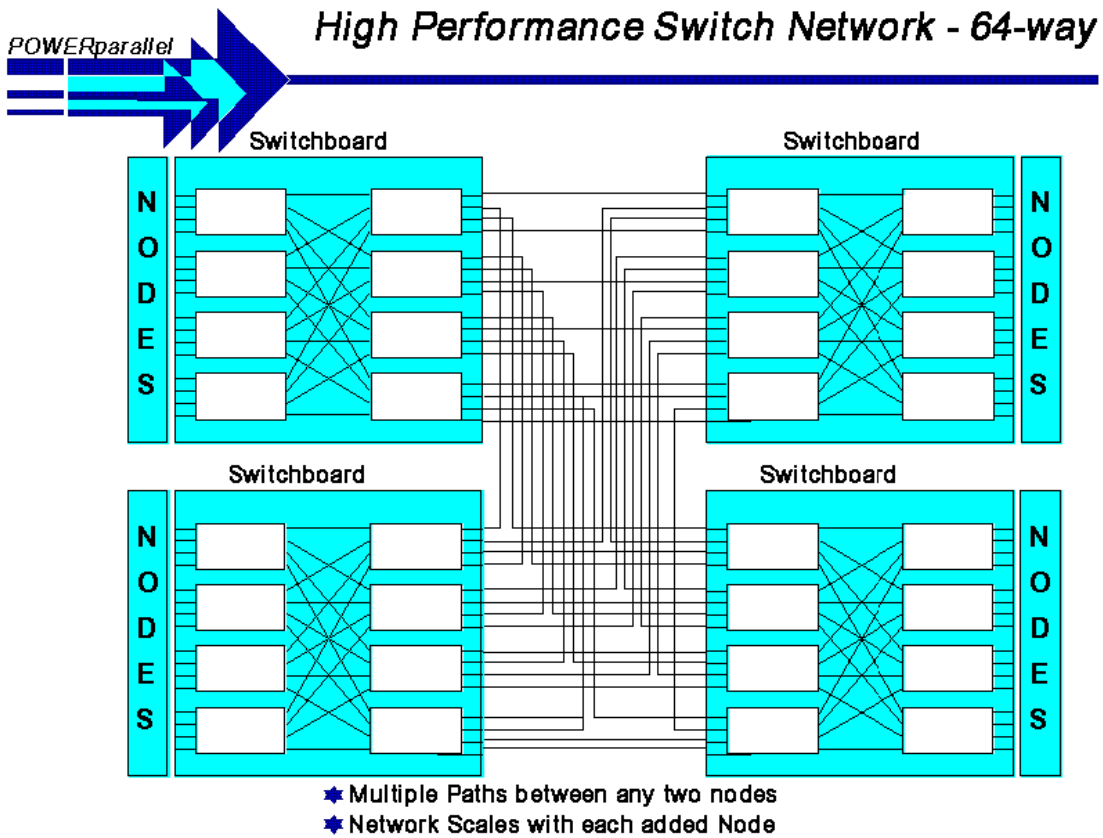
Within hypernode: SMP
(Symmetric MultiProcessor)

Across hypernodes: NUMA
(Non−uniform Memory Access machine)

Locality of reference is extremely important

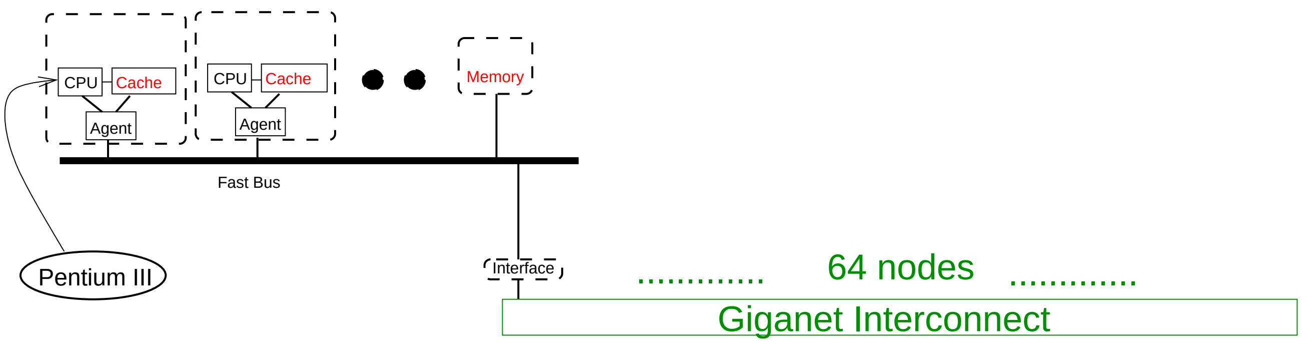
Programming model: C/FORTRAN + OpenMP

Distributed-memory computers: each processor has a different address space (eg. IBM SP-2)



Programming model: C/FORTRAN + MPI

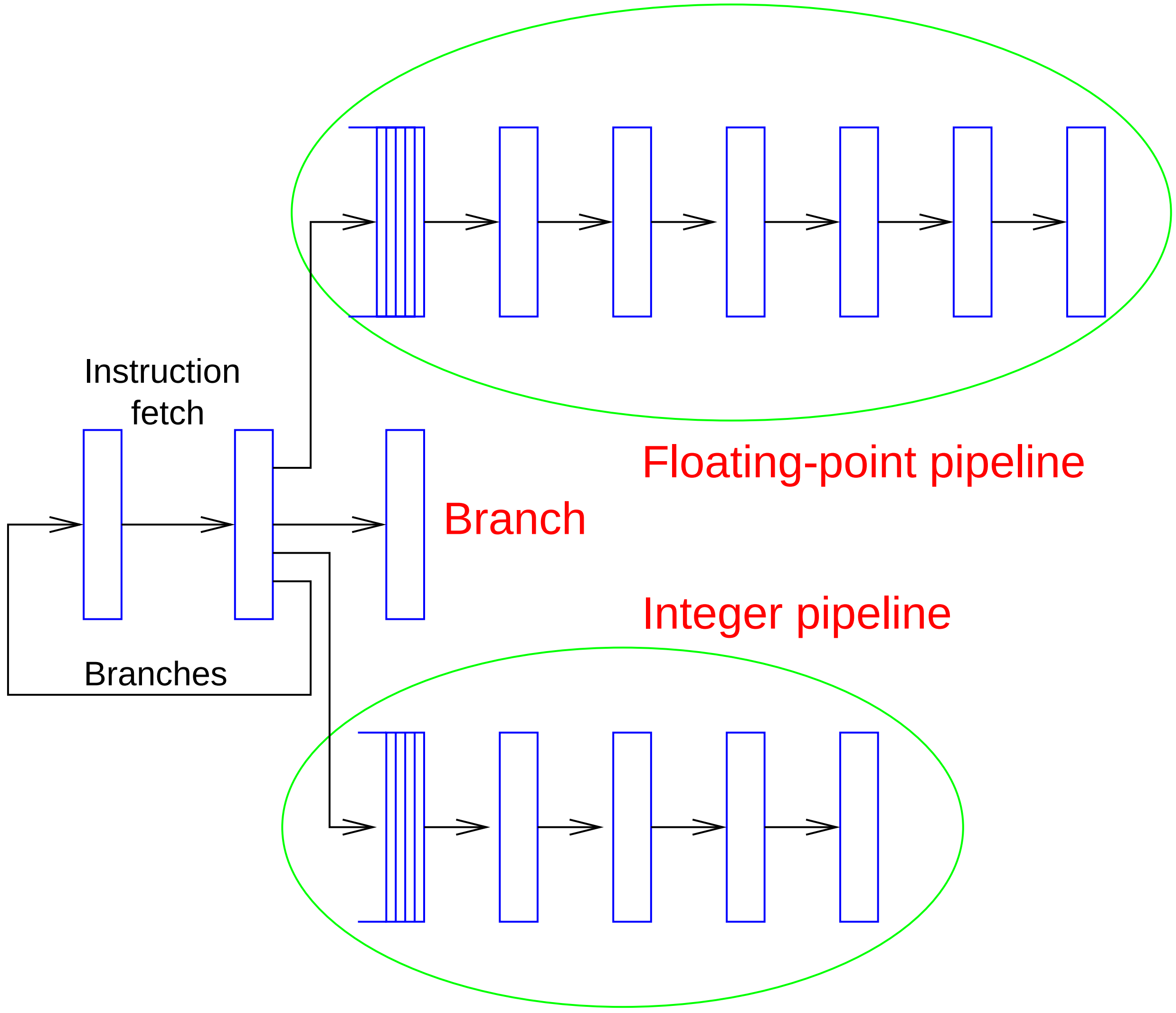
The AC3 Cluster: network of SMP nodes



- Each node is a 4-way SMP with Pentium III processors
- 64 nodes are connected by a Giganet interconnect
- Within each node, we have a shared-memory multiprocessor
- Across nodes, we have a distributed-memory multiprocessor

=> Programming such a hybrid machine is even more complex!

Pipelined processors: must exploit instruction-level parallelism



Lessons for software

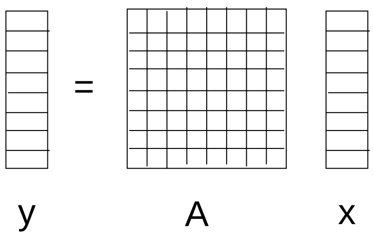
To obtain good performance on such high-performance computers, an application program must

- exploit coarse-grain parallelism
- exploit instruction level parallelism
- exploit temporal and spatial locality of reference

Let us study how this is done, and understand why it is so hard to worry about both parallelism and locality simultaneously.

Exploiting coarse-grain Parallelism

```
do j = 1..N
 do i = 1..N
  Y[i] = Y[i] + A[i,j]*X[j]
```



Each row of the matrix can be multiplied by x in parallel.
(ie., inner loop is a parallel loop)
If addition is assumed to be commutative and associative,
then outer loop is a parallel loop as well.

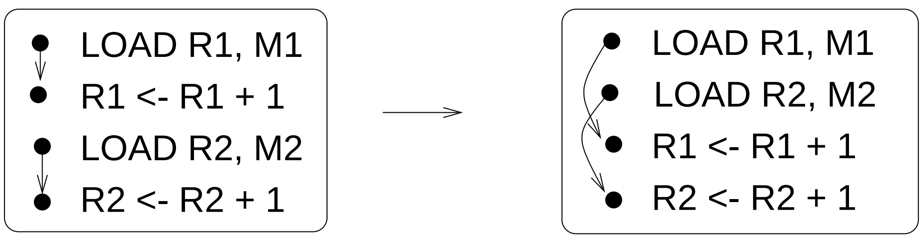
Question: How do we tell which loops are parallel?

```
do i = 1 ..N
 x(2*i + 1) = ...x(2*i) ....
```

```
do i = 1 ..N
 x(i+1) = ....x(i) ....
```

One of these loops is parallel, the other is sequential!

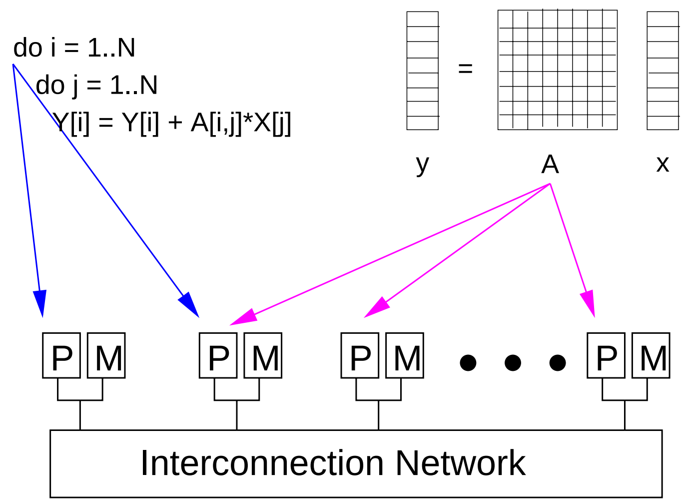
To exploit pipelines, instructions must be scheduled properly.



LOADs are not overlapped — LOADs are overlapped

- Software pipelining: instruction reordering across loop boundaries
- Hardware vs software:
  - superscalar architectures: processor performs reordering on the fly (Intel P6, AMD K5, PA-8000)
  - VLIW, in-order issue architectures: hardware issues instructions in order (CRAY , DEC ALPHA 21164)

Exploiting locality (I)



Computation distribution: which iterations does a processor do?

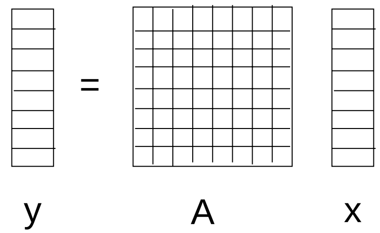
Data distribution: which data is mapped to each processor?

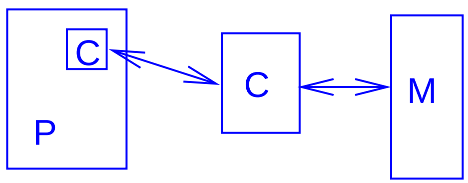
Computation and data distributions should be "aligned"
to optimize locality: a processor should "own" the data it needs
for its computation. Misaligned references: communication

Question: What are good distributions for MVM?

Exploiting locality (II)

```
do i = 1..N
  do j = 1..N
    Y[i] = Y[i] + A[i,j]*X[j]
```





Processor has a 1st and 2nd level cache and local memory

Uniprocessor locality

- Program must have spatial and temporal locality of reference to exploit caches.
- Straight-forward coding of most algorithms results in programs with poor locality.
- Data shackling: automatic blocking of codes to improve locality

Worrying simultaneously about parallelism and locality is hard.

Radical solution: multithreaded processors

- Forget about locality.
- Processor maintains a pool of active threads.
- When current thread makes a non-local memory reference, processor switches to different thread.
- If cost of context-switching is small, this can be a win.
- Tera, IBM Blue Gene machine

Summary

To obtain good performance, an application program must

- exploit coarse-grain parallelism
- exploit temporal and spatial locality of reference
- exploit instruction level parallelism

Systems software must support

- low-cost process management
- low-latency communication
- efficient synchronization

Mismatch with conventional software environments:

- Conventional languages do not permit expression of parallelism or locality.
- Optimizing compilers focus only on reducing the operation count of the program.
- O/S protocols for activities like inter-process communication are too heavy-weight.
- New problems: load balancing

=>

Need to re-design languages, compilers and systems software to support applications that demand high-performance computation.

Lecture Topics

- Applications requirements: examples from computational science
- Architectural concerns: shared and distributed memory computers, memory hierarchies, multithreaded processors, pipelined processors
- Explicitly parallel languages: MPI and OpenMP APIs
- Restructuring compilers: Program analysis and transformation
- Memory hierarchy management: Block algorithms, tiling and shackling.

Lecture Topics (cont.)

- Automatic parallelization: shared and distributed memory parallelization, High Performance FORTRAN.
- Program optimization: control dependence, static single assignment form, dataflow analysis, optimizations.
- Exploiting instruction level parallelism: instruction scheduling, software pipelining.
- Object-oriented languages: Object models and inheritance