Announcements

- Course account usage
  - NO MORE DEDICATED QUEUE RUNS!
  - you will not be penalized for less-than-ideal speedup
  - 300 more units each have been added to the Exemplar and Origin accounts
- Meeting to discuss project
  - please set up an appointment ASAP
- Lectures
  - Lecture 9 rescheduled to Tuesday, Nov. 3rd, 101 WWH, 7:30pm-9:00pm
  - No class on Nov. 5th and Nov. 12th
- Department colloquia: 1302 WWH, 11:30am-12:30pm
  - October 30th: Josep Torrellas, “The Illinois IA-COMA Project”
  - November 13th: Jaswinder Pal Singh

Outline

- Last lecture: Scalable Shared Memory Machines (contd.)
  - case study: SGI Cray Origin 2000
  - performance optimizations
    - relaxed memory consistency models
    - programmable protocol processors
      - case study: Stanford FLASH
- This lecture: Programming for Performance
  - (spillover from Lecture 7): coherent replication in main memory
  - performance concerns common to all programming models
    - load-balance, synchronization, locality
  - data-parallel programming models
    - case study: Fortran D
    - data decomposition
    - compilation strategies

Overcoming Capacity Limitations

- Problem
  - in CC-NUMA systems, processor cache replicates data directly on reference
    - no copy in (local) main memory
  - consequence:
    - hardware cache determines capacity of local replication
    - artifactual communication to remote memory on miss
- Solutions
  - tertiary caches
    - a cache for remote accesses
    - e.g., Sequent NUMA-Q, Convex Exemplar
  - already present if nodes of the machine are small-scale SMPs
  - problem:
    - home node stays fixed (even if only one processor accesses block)
      - statically established tertiary cache can be wasteful
  - cache-only memory architectures (COMA)
Cache-Only Memory Architectures (COMA)

- Treat all of main-memory as a hardware-controlled cache
  - e.g. Kendall Square Research (KSR) machines
  - replication capacity is now limited by main-memory size
  - migration is automatic
    - data moves to main memories of nodes which access it
    - attraction memory
  - programmer need not worry about initial data distribution
    - focus on inherent communication and false sharing

- Disadvantage: Hardware complexity
  - storage
    - per-block tags in main memory
    - extra memory overhead needed for replication in the attraction memories
  - complicated coherence protocol
    - no home node: need to find where the data resides
    - how to handle replacement of last copy of a block?

COMA: Performance Trade-Offs

Application characteristics

- Very low miss rate
  - CC-NUMA = COMA

- Mostly capacity misses and/or poor initial data placement
  - CC-NUMA < COMA

- Mostly coherence misses
  - CC-NUMA > COMA

Recap: 4-step Parallelization Process

<table>
<thead>
<tr>
<th>Step</th>
<th>Architecture-Dependent?</th>
<th>Major Performance Goals</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>Mostly no</td>
<td>Expose enough concurrency but not too much</td>
</tr>
<tr>
<td>Assignment</td>
<td>Mostly no</td>
<td>Balance workload</td>
</tr>
<tr>
<td>Orchestration</td>
<td>Yes</td>
<td>Reduce noninherent communication via data locality</td>
</tr>
<tr>
<td>Mapping</td>
<td>Yes</td>
<td>Put related processes on the processor if necessary</td>
</tr>
</tbody>
</table>

Table 2.1 Steps in the Parallelization Process and Their Goals
Recap: Programming as Successive Refinement

• Partitioning often independent of architecture, and done first
  – view machine as a collection of communicating processors
    • PRAM + communication costs
  – goals
    • balancing the workload + reducing wait time at synchronization points
    • reducing inherent communication
    • reducing extra work

• Then, interactions with architecture (orchestration)
  – view machine as extended memory hierarchy
    • extra communication due to architectural interactions
    • cost of communication depends on how it is structured
  – goals
    • reducing amount of communication
      – inherent: change logical data sharing patterns in algorithm
      – artifactual: exploit spatial, temporal locality in extended hierarchy
    • structuring communication to reduce cost

Recap: Summary of Performance Tradeoffs

\[
speedup_{\text{parallel}}(p) \leq \frac{\text{sequential work}}{\max\left(\text{work on any processor + synchronization wait time + data access costs + overheads}\right)}
\]

• Load balance
  – fine-grain tasks
  – random or dynamic assignment
• Inherent communication volume
  – coarse-grained tasks
• Extra work
  – coarse-grained tasks
  – simple assignment
• Artifactual communication costs
  – big transfers: amortize overhead and latency
  – small transfers: reduce contention and occupancy

Implications of Architectural Support

• Architectures provide varying degrees of support for naming, synchronization, and communication
  – small-scale shared-memory machines
    • (naming) individual memory locations in a global name space,
      (synchronization) hardware locks, (communication) snoop-based coherence at cache-line granularity
  – large-scale distributed memory machines
    • (naming) private name spaces, (synchronization) implicit in message-passing,
      (communication) relatively heavy-weight
  – large-scale shared memory machines
    • (naming) individual memory locations in a global name space,
      (synchronization) remote atomic operations, (communication) coherent data transfer with moderate overhead
  – this support helps resolve tradeoffs between load-balance, communication costs (locality), and synchronization wait times

⇒ What remains must be addressed by the compiler/programmer

Outline for the Remainder of the Course

• Programming for performance
  – data parallel models (Lecture 8, today)
    • programmer expresses parallelism at a high level
    • compiler responsible for bridging the gap
    • shortcoming: limited applicability
  – message-passing models (Lecture 9, November 3rd)
    • wide variation in architectural support
    • run-time libraries/programmer responsible for performance
  – shared-memory models (Lecture 10, November 19th)
    • hardware responsible for delivering performance
    • programmer writes code to take advantage of hardware

• Hardware/Software Tradeoffs (Lecture 10, November 19th)
• Future Directions (Lecture 11, December 3rd)
Recap: Data Parallel Models

- Dynamic allocation of shared data
  - G_MALLOC (global malloc)

- Concurrent loops
  - for_all
  - parallel processes are implicitly active: only within for_all body

- Decomposition of data and computation
  - DECOMP arr[BLOCK, *, nprocs]
    - specifies assignment of data elements to processes
  - owner-computes: specifies assignment of iterations to processes

- Collective operations
  - REDUCE, others such as broadcast, etc.
  - all-to-all operations, implemented efficiently by the underlying system

Fortran D Language

- Goals
  - enable machine-independent parallel programming
  - compiler responsible for producing correct, efficient code
  - project history
    - started at Rice University under Ken Kennedy in 1990
    - leveraged previous work at U. Illinois and Rice on vectorizing compilers
    - current incarnation: The D System
    - primary influence for High Performance Fortran (HPF)

- Rationale
  - data-decomposition is key  have the programmer specify it
    - influences load-balance and locality
  - the compiler can then infer
    - which tasks are concurrent?
    - how they should be mapped to processors?
    - how to orchestrate interactions (communication, synchronization)?

Fortran D Language (contd.)

- Fortran 77 plus
  - support for data decomposition
    - FORALL
    - other stuff: irregular data distribution, dynamic data decomposition, etc.

- Support for data decomposition

  - DECOMPOSITION D(N,N) REAL A(N,N)
    - ALIGN A(1:J) with D(J-2,I+3)
    - DISTRIBUTE D(:, BLOCK) DISTRIBUTE D(CYCLIC..)

Compiling Fortran D for MIMD Machines

- Given a data decomposition (for all arrays in the program), decide
  1. which processor executes which set of loop iterations
     - owner-computes rule
     - avoids load-balance and locality concerns with explicit decomposition
     - automatic choice of a good data distribution is an active research area
  2. necessary synchronization between dependent loop iterations
     (for distributed-memory machines)
  3. generate message sends/receives to provide access to non-local data

  Dependence analysis
  Data decomposition analysis
  Partitioning analysis
  Communication analysis

  Message vectorization
  Collective communication
  Run-time processing
  Pipelined computations

  Program Analysis
  Program Optimization
  Code Generation
Program Analysis

4 steps

• Dependence Analysis
  – determine a statement execution order (loop iteration order) which preserves original program semantics
  – identify private scalars and array variables
  – recognize reductions

• Data Decomposition Analysis
  – determine the data decomposition for each reference to a distributed array
  – required, because dynamic decompositions are allowed

• Partitioning Analysis
  – data and computation partitioning to processors

• Communication Analysis
  – determine the variable references which cause nonlocal data accesses

Program Analysis: Dependence Analysis

• Types of dependences: $S_1 \delta S_2$
  – loop-independent: occurs in the same iteration
  – loop-dependent: occurs across iterations
  – true (flow): $S_1$ writes a memory location later read by $S_2$
  – anti: $S_1$ reads a memory location later written by $S_2$
  – output: $S_1$ writes a memory location later written by $S_2$
  – input: $S_1$ reads a memory location later read by $S_2$

Program Analysis: Data Decomposition Analysis

• Goal: determine the decomposition for each array reference

• Analysis similar to def-use chains
  – reaching decompositions: the set of decomposition specifications that may reach an array reference
  – compute similar to reaching definitions
    • requires intra- and inter-procedural analysis
    • may require run-time checks or cloning to generate the right code
      – if multiple decompositions reach a procedure
    – only an issue because of Fortran D dynamic decomposition

Program Analysis: Partitioning Analysis

• Goal: divide overall data and computation among processors

• Two-step process
  – first, partition all arrays onto processors (using decomposition directives)
  – then, derive functional decomposition (using owner-computes rule)

• Regular section descriptors: […] , $l_i : u_i , s_i , …$
  – iteration sets: set of loop iterations
  – index sets: set of array locations
  – need to convert between global and local (per-processor) indices

• Step 1: compute local index sets
  – the local array portion owned by every processor

• Step 2: compute local iteration sets of individual array references $R$
  – the set of loop iterations that cause $R$ to access processor-local data
  – boundary conditions handled by augmenting RSDs: pre, mid, and post sets
Partitioning Analysis: Example

REAL A(100,100), B(100,100)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(:, BLOCK)

do k=1, time
  do j = 2, 99
    do i = 2, 99
      A(i,j) = 0.25*( B(i,j-1)+B(i-1,j)+B(i+1,j)+B(i,j+1) )
    enddo
  enddo
enddo

Program Analysis: Communication Analysis

• Goal: determine variable references which cause nonlocal accesses

• Examine all rhs references: for each reference
  – construct the index set accessed by each processor
  – nonlocal index set = resulting RSD – local index set

  – for the Jacobi example:

Local index set for B: [1:100, 1:25]
Local iteration set for Proc(2:3):
  [ 1:time, 1:25, 2:99 ]

Nonlocal index set for Proc(2:3):
  for array B: [2:99, 0], [2:99, 26]

Program Optimization

• Goals: exploit parallelism and reduce communication overhead
  – using the local index sets, local iteration sets, and nonlocal index sets from the program analysis phase

• Focus on communication-overhead reduction
  – communication required to access locations in nonlocal index sets
  – naïve approach: run-time resolution
    • insert guarded send/recv operations directly preceding each nonlocal reference
    • problem: very inefficient (small, frequent messages)!
  – message vectorization: place communication in outermost (legal) loop
  – communication selection: choose between primitives
    • individual send/receive, collective operations, etc.
  – other optimizations: (more in Lecture 9)
    • message pipelining, message coalescing and aggregation
    • use of non-blocking messages

Program Optimization: Message Vectorization

• Goal: place communication operations in outermost (allowable) loop

For each nonlocal reference R,
  - examine cross-processor true-dependences with R as sink
  - insert communication operation
  at the deepest loop level that carries any of these dependences
Code Generation

- SPMD node program
  - program partitioning
    - allocate storage only for locally-owned data + overlap regions
    - loop bound reduction: only iterations from the local iteration set
      - sometimes need guard introduction
  - message generation
    - nonlocal index set: data which needs to be sent to a particular processor
    - compute owners of this section
      - possibly multiple processors (each contributing a portion of the RSD)
    - convert local indices of receiving processors to that of sending processors
    - insert guarded (non-blocking) sends and receives for each piece

```
REAL A(100,100), B(100,100)
DECOMPOSITION D(100,100)
ALIGN A, B with D
DISTRIBUTE D(:, BLOCK)
```

do k=1, time
  do j = 2, 99
    do i = 2, 99
      A(i,j) = 0.25*( B(i,j-1)+B(i-1,j)+B(i+1,j)+B(i,j+1) )
    enddo
  enddo
  do j = 2, 99
    do i = 2, 99
      B(i,j) = A(i,j)
    enddo
  enddo
enddo

Next Lecture

- Programming for performance (contd.)
  - message-passing models
    - messaging layers
    - multithreading
    - message pipelining and aggregation