Increasing hardware utilization through parallelism

• Types
  • Instruction-Level Parallelism
  • Shared memory
  • Distributed memory

• Algorithmic considerations
Physics puts constraints on hardware design.
We have multicore hyper-threaded pipelined vector processors.
Instruction-level parallelism (ILP)

Multiple instructions that can be run at once
Only a CPU without a pipeline would have no ILP
Compiler and CPU both attempt to maximize ILP

- You can help
- Compare optimization lectures
Different forms of parallelism

Multiple instructions, multiple data (MIMD)
- Threads
  - Independent units of execution
  - Some shared data
- Processes
  - No shared data

Single instruction, multiple data (SIMD) ***
- Vectorization instructions (e.g. SSE, Altivec)

Multiple instructions, single data (MISD)
- Pipelining, branch prediction
- Fault-tolerant systems
Parallelism everywhere

SETI@home

The Search for Extraterrestrial Intelligence at HOME

Data Analysis
- Computing Fast Fourier Transform: 87%
- Doppler drift rate: 0.1776 Hz/sec
- Frequency resolution: 0.074506 Hz
- Strongest Peak: power 139.27
  - (4184.6 Hz at 26.84 seconds, drift rate 0.047 Hz/sec)
- Strongest Gaussian: power 0.71, fit 2.750
  - (5136.7 Hz at 25.17 seconds, drift rate 0.000 Hz/sec)
- Overall: 9.068% done
- CPU time: 0 hr 19 min 21.8 sec

Data Info
- From: 6 hr 33 min 50 sec RA, +28 deg 30 min 36 sec Dec
- Recorded on: Sun Feb 21 17:00:20 1999 GMT
- Source: Arecibo Radio Observatory
- Base Frequency: 1.419599607 GHz

User Info
- Name: David
- Data units completed: 2726
- Total computer time: 52 hr 15 min 45.8 sec
Multithreading on a Single Core

- Regular switching between threads
- Also switching when a thread *waits*
  - Can wait for user input
  - Might wait for synchronization on another thread
  - Any synchronous I/O call can cause a wait
- More threads than cores can make sense if
  - If you have a lot of I/O
  - Simultaneous Multithreading (SMT or Hyper-threading)
- Thread priority will control scheduling
- Cost of thread creation/destruction
- **Cost of context switching**
Asynchronous I/O
1. Make an I/O call
2. Do something
3. Check status for I/O

I/O latency doesn’t change, but you can use useful work instead.

This could be done with threads instead... but each thread comes at a cost
Symmetric Multithreading

- Called HyperThreading by Intel
- Double register files
- Context switching extremely cheap
- Benefits are application-specific:
  - Hide memory latency
  - Decreased cache size/thread
- Helps if memory latency is a bottleneck
Vectorization

- True SIMD
- SSE (Intel & AMD) & AltiVec (IBM & Motorola)
  - 128-bit registers: 16 chars, 8 shorts, 4 floats, 2 doubles.
- AVX: 256-bit registers
- Usage:
  - *Intrinsics* provided by compiler
  - Auto-vectorization by compiler
Parallelism everywhere

Dreamworks, Industrial Light&Magic
Multithreading on Multiple Cores

- Threads will still sometimes wait...
- ...possibly resuming on a different core
  - Consequences for cache behavior
  - Cache coherency maintained by hardware – sort of
- Thread *CPU affinity* can be set manually for performance reasons
  - This will make your code more sensitive to scheduling or other processes running
  - Possibly recommended for a dedicated machine
Inter-thread communication

Used to be expensive

Getting cheaper, at least between some threads

Synchronizing between all threads is expensive

Modern profilers can help assess the overhead
OpenMP

Allows simple implementation of threading
Basic constructs for stating several paths to be executed at the same time
Execute loop iterations separately
Realized through compiler pragmas
Some code can work with and without OpenMP
void M_mult(float A[n][n], float B[n][n], float C[n][n]) {
    int i, j, k;
    float sum;

    #pragma omp parallel for, local(i, j, k, sum)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++){
            sum = 0;
            for (k=0; k<n; k++)
                sum += A[i][k]*B[k][j];
            C[i][j]=sum;
        }
OpenMP

Compile with `-fopenmp` (in GCC)

Difficulty:

- The iterations in your for-loop must be *independent*
- Properly declaring local, private and shared variables
Memory allocation in Multicore Environments

NUMA and memory affinity is one consideration

NUMA: Non-Uniform Memory Access

Memory allocated on one CPU might consistently be used by another

Libraries and e.g. heap implementations may not be NUMA-aware

Usually: *NUMA memory allocated on* “first-touch” basis
Heap implementation might lock all threads
Java and some malloc implementations keep a “thread cache” of memory soon to be allocated
Some mallocs might not even be thread-safe
Be sure to link to the right runtime library (should generally be ok)
Other Shared Resources

- Libraries might not be thread-safe at all:
  - Undefined behavior when used concurrently
  - Undefined behavior when used from multiple threads \textit{at all}
  - Libraries might not be re-entrant

- Libraries might scale badly:
  - Might use a single lock for all threads
  - Unwarranted use of resources on a per-thread level
Message Passing Interface

Standardized communication between processes
Not shared memory
- Although can be implemented using shared memory

Ideal for lots of memory, no shared caches

Can be distributed over closely connected clusters
Solution

OpenMP on closely connected cores
MPI between machines
Sockets for truly distributed computing
Parallelism Everywhere

Engineering: Finite Element Method
GPUs – Complicated parallelism

- **Multiple Multiprocessors**
  - Each multiprocessor: Single Instruction Multiple Thread (SIMT)

- **Warps**
  - 32 (or so) identical threads, different datasets
  - Some shared memory
  - A single program counter

- **Branching is expensive**

- **So are memory accesses** (any thread will stall all threads)

- **Solution:** Multiple warps in each multiprocessor
  - One warp runs while others access memory
Some words on Cell

Cell Broadband Engine
POWER-based
Main core (PPU)
8 Synergistic Processing Element (SPE)
  • Vector machine
  • Very simple cores

Not a shared-memory architecture
What would normally be a cache is an explicit local memory
Asynchronous access to main memory
Prefetching goes from nice to mandatory
Algorithms

An algorithm:
Do A
Do B
Do C

In order to parallelize, one must determine which of A, B, and/or C are independent. There may also be parallelism inside A, B, or C.

Once again, well-structured code is key. Avoid side-effects!
Performance considerations

- What is the bottleneck?
  - Computation
  - Communication bandwidth
  - Comm. latency (synchronization)
  - Load balancing

- What can we do about it?
  - Increase parallelism
  - Increase arithmetic intensity
  - Change algorithm
Arithmetic Intensity

- Ratio of math operations and data transfer operations
- Units are e.g. operations/word
- Can vary according to
  - Algorithm
  - Implementation
  - Problem size (per node)
- A process with low arithmetic intensity is likely to be bandwidth bound — multithreading will not help
Parallel algorithmics

- Fork-join parallelism
Amdahl's Law

- For a given algorithm, there's a maximum possible speedup
  - \( P_{\text{serial}} = \) serial proportion of runtime
  - \( P_{\text{parallel}} = \) parallelizable proportion
  - \( 1 = P_{\text{parallel}} - P_{\text{serial}} \)
  - \( S_{\text{max}} = 1/(P_{\text{serial}}) \)

- If your program is 80\% parallelizable:
  - Only 5x faster with \( \infty \) number of processors
Dependency Analysis

Directed Acyclic Graph (DAG)
Work/span Law

- For a given algorithm, there's a maximum possible speedup
  - Work = total number of execution units
  - Span = length of longest execution path
  - Maximum parallelism = work/span

- In the toy example: $9/5 = 1.8$
Parallel Algorithm Design

- Avoiding synchronization stall
  - Locks
  - Peer-to-peer communication
  - Overlap with useful work
  - Transactional memory

- Minimizing communication cost
  - Overlap with computation
  - Recompute locally instead of transferring
  - Postpone communication calls and coalesce into one
Load balancing is always an issue
Improving Load Balancing

Option 1: Be very smart

Option 2: Schedule small work units at the end

Example:
split work into
3 large,
2 medium,
6 small pieces
• Can be easy, can be tough

function quicksort(array, start, end)
    if (end - start < 2)
        swapifgt(array, start, end)
    return

    index = getPivot(array, start, end)
    partition(array, start, index, end)
    quicksort(start, index)
    quicksort(index, end)
Parallelizing Quicksort

Problem:

1 thread

2 threads

4 threads

8 threads

Problem:
Task-based parallelism

- Define task from
  - input/output data
  - operations on data (kernel or function)
- Hand off the task to a runtime
- Runtime has a scheduling policy that gives tasks to worker threads
Benefits

- A runtime can select what is executed, where, and when
- Scheduler and runtime can be very smart and hardware-aware
- Reduce tight ordering in an algorithm
- Avoid stalling on unnecessary synchronization points
- ... But writing a smart implementation by hand can always yield even better performance.
Parallelizing Quicksort

- Reformulating the algorithm

```java
def quicksort(array, start, end):
    stack s
    s.push({start, end})
    parallel while(! s.isempty())
        task = s.pop()
        if(task.end-task.start < 2)
            swapifgt(array, task.start, task.end)
            continue
        index = getPivot(array, task.start, task.end)
        partition(array, task.start, index, task.end)
        s.push({start, index})
        s.push({index, end})
```

global scheduler s

function quicksort(array, start, end)
    s.push({start, end})

function workerLoop()
    while(s.isempty()) // wait for task
        task = s.pop();
        if(task.end - task.start < 2)
            swapifgt(array, task.start, task.end)
            return
        index = getPivot(array, task.start, task.end)
        partition(array, task.start, index, task.end)
        s.push({start, index})
        s.push({index, end})
Transactional memory

Bring database semantics to single memory accesses

The problem of threading is that multiple accesses need to be synchronized

Locking before every read and write is expensive

- Most of the time, no conflict will arise

Instead, rollback and throw an exception in the exceptional case where a conflict actually arises

Keep transactions short, as rollbacks will be expensive instead
Hardware transactional memory is not coming soon.

Software transactional memory can be useful. Often seen as “lock-free” algorithms

1) Do edits
2) Check that no one was doing anything behind your back
(Quite tricky, but can be done)

A little tip:
Atomic exchange of pointer to “state” object
Counter increased for any change
Other important aspects of high-performance parallel computing

- Lots of underlying technology that makes threads possible:
  - e.g. Operating System process scheduling
  - Network technology
  - Network topology

- Supercomputing centers
  - Uppmax, SNIC
  - Grids
Network Topologies

- Suppose you have a *lot* of processors
- How do you hook them up?
  - Ring/mesh/torus, 1-d, 2-d, 3-d?
  - Fully connected?
  - Tree hierarchy?
- How does this impact usability?
Parallelism Everywhere

- Disaster management

Tsunami Propagation Forecast
Contours of forecasted maximum wave amplitudes [cm], detailing tsunami energy propagation.

Event ID: hvpd9.4
Earthquake Magnitude: 8.9
Earthquake Location: [38.349,142.409], "near the east coast of Honshu, Japan"

Origin Time: 05:46:28 (UTC)
Date: 3/11/2011

West Coast and Alaska Tsunami Warning Center