CSE 160
Lecture 9

Load balancing and Scheduling
Some finer points of synchronization
NUMA
Announcements

• The Midterm: Tuesday Nov 5\textsuperscript{th} in this room
  ‣ Covers everything in course through Thu. 11/1
  ‣ Closed book; may bring a 8x11” sheet of paper
  ‣ Midterm review session
    Friday Nov 1\textsuperscript{st} at 4pm in CSB 002
  ‣ Both of Wednesday’s sections are cancelled

• Midterm review on Thursday
  ‣ Some readings relevant to the review
  ‣ Q2 return and questions
  ‣ Review questions will be posted later today
  ‣ \textbf{Be prepared} to ask questions on Thursday

• SDSC Tour on Friday at 1.15 PM
Today’s lecture

• Some finer points of synchronization
• Scheduling and load balancing
• NUMA Architectures
The Particle simulation in OpenMP

• Two routines contain the work to be parallelized according to `openmp #pragmas`
• What’s not inside **OMP parallel for** should run serially
• Synchronization?

```c
void SimulateParticles(){
    for( int step = 0; step < nsteps; step++ ) {
        apply_forces(particles,n);

        move_particles(particles,n);

        VelNorms(particles,n,uMax,vMax,uL2,vL2);
    }
}
```
Software design issues - synchronization

• We’ve seen 2 different primitives that synchronize
  ‣ Mutexes
  ‣ Atomics
  ‣ (1 more, condition variables will come later)

• When applicable, Atomics can be simpler to use than mutexes, and may be more efficient

• When we use mutexes, we need to be sure that we unlock the lock and do so promptly
  ‣ Failing to do so in a timely manner can hurt performance
  ‣ Failing to do so at all can lead to deadlock
Making sure we unlock the lock

• A common problem is to have no way of unlocking the lock when a function exits abnormally, i.e. an exception
• C++ provides the lock_guard class template to help avoid this difficulty
• Implements the RAII principle
  Resource Allocation is Initialization
• See Williams 38-40
RAII and Lock_guard

• The lock_guard constructor acquires (locks) the provided lock constructor argument
• When a lock_guard destructor is called, it releases (unlocks) the lock
• How can we improve this code?

```cpp
int val;
std::mutex valMutex;
...
{
    std::lock_guard<std::mutex> lg(valMutex); // lock and automatically unlock
    if (val >= 0)
        f(val);
    else
        f(-val); // pass negated negative val
} // ensure lock gets released here
```
Code Structure Issues

• Beware of returning or passing references to members that must be updated with synchronization applied.
• Think about what can eventually be done with those references, and use care in making a call to an external function within a critical section.

```cpp
class some_data {
    ... 
    public:
        void do_something();
};

class data_wrapper {
    private:
        some_data data;
        std::mutex m;
    public:
        template<typename Function>
        void process_data(Function func) {
            std::lock_guard<std::mutex> l(m);
            func(data);
        }
};
```
Today’s lecture

• Some finer points of synchronization

• Scheduling and load balancing

• NUMA Architectures
Non-uniform workloads

• In the assignment, particles are distributed randomly, so the work applied to each box to push the particles is also uniform.

• But in some applications, the particles aren’t distributed evenly … a uniform (BLOCK) partitioning is inappropriate.
Load Balancing

• Boxes carry varying amounts of work depending on local particle density
• Particles can redistribute themselves over processors
• 2 Alternatives
  ‣ Static: cyclic mapping
  ‣ Dynamic: self scheduling
Static approach: BLOCK CYCLIC

- Divide bins pieces of size CHUNK
- Core k gets chunks $k, k+NT*CHUNK, k+2*NT*CHUNK, \ldots$
- Also called *round robin* or *block cyclic*

```c
#pragma omp parallel for schedule(static, 2)
for( int i = 0; i < n; i++ ) {
    for (int j = 0; j < n; j++) {
        doWork(i,j);
    }
}
```
Second approach: dynamic scheduling

- Unlike CYCLIC decomposition, workload assignments are made on demand
- We subdivide workload into pieces of size CHUNK
- Also called *processor self scheduling*

```c
#pragma omp parallel for \\ schedule(dynamic, 2)
for( int i = 0; i < n; i++ ) {
    for (int j = 0; j < n; j++) {
        doWork(i,j);
    }
}
```
Tradeoffs in dynamic scheduling

- Dynamic workload assignments: processors schedule themselves
- Each thread samples a unique (and disjoint) set of indices, changes from run to run
- A shared counter or work queue represents the work
- User tunes work granularity (chunk size) trading off the overhead of workload assignment against increased load imbalance
  - Finest granularity: each point is a separate task
  - Coarsest granularity: one block per processor

![Diagram showing the relationship between increasing granularity, running time, and increasing load imbalance.](image)

Increasing granularity →

Running time

High overheads

Increasing Load imbalance

©2013 Scott B. Baden / CSE 160 / Fall 2013
Higher dimensional partitioning

- With either BLOCK CYCLIC or DYNAMIC scheduling, we can split in higher dimensions
- Can help improve workload imbalance

\[
\begin{align*}
\text{[ CYCLIC(2), CYCLIC(2)]} & \quad \text{[CYCLIC(2), BLOCK]} & \quad \text{[BLOCK, CYCLIC(1)]}
\end{align*}
\]
Tradeoffs in choosing the chunk size

- **CHUNK=1**: each box needs data from all neighbors
  - Every processor loads all neighbor data into its cache!
  - Compare with [BLOCK,BLOCK]
- **CHUNK=2**: each box in a chunk of 4 boxes needs ¼ of the data from 3 neighboring chunks
  - Each processor loads 3 chunks of neighbor data into cache
- **CHUNK=4**: Only edge boxes in a chunk need neighbor data, 20 boxes: processor loads 1.25 chunks of neighbor data
How does self scheduling work?

SelfScheduler S(n, P, chunk);

while ( S.getChunk(mymin))
    for i = mymin to mymin + chunk - 1
        work(i, chunk);
    end for
end while
Implementation

- Critical section can be costly, but OMP atomic is restricted to simple update statements, e.g. `++ +=`
- C++ atomic<T> not guaranteed to be lock free, but probably more efficient
- OMP and C++ implementations

```c
#include <openmp.h>

boolean getChunk(int& mymin){
    #pragma omp critical // Inefficient
        // Crit Sect
        k = _counter;
        _counter += _chunk;
    if ( k > (_n - _chunk) // not past last chunk
        return false;
    mymin = k;
    return true;
}
```

```c
#include <atomic.h>

boolean getChunk(int& mymin){
    mymin = _counter.fetch_add(_chunk)
    if (mymin > (_n - _chunk) // not past last chunk
        return false;
    else return true;
}
```
Today’s lecture

- OpenMP
- NUMA Architectures
Multiprocessor organization

• Recall that with shared memory, the hardware automatically performs the global to local mapping using address translation mechanisms

• 2 types, depends on uniformity of memory access times
  ‣ **UMA**: *Uniform* Memory Access time
    Also called a Symmetric Multiprocessor (SMP)
  ‣ **NUMA**: *Non-Uniform* Memory Access time

©2013  Scott B. Baden / CSE 160 / Fall 2013
NUMA Architectures

- The address space is global to all processors, but memory is physically distributed
  AKA *distributed shared memory architectures*
- Software/hardware support to monitor sharers
- Stanford Dash; NUMA nodes of the Cray XE-6, SGI, Altix (up to 512 cores), Origin 2000, Gordon system at San Diego Supercomputer Ctr

en.wikipedia.org/wiki/Non-Uniform_Memory_Access
Some terminology

• Every block of memory has an associated home: the specific processor that physically holds the associated portion of the global address space
• Every block also has an owner: the processor whose memory contains the actual value of the data
• Initially home = owner, but this can change …
• … if a processor other than the home processor writes a block
How do we build NUMA

- A **directory** keeps track of sharers, one for each block of memory (often cache line)
- Point-to-point messages manage coherence
- Each block has an associated **home**: the specific processor that physically holds the associated portion of the global address space
- Every block also has an **owner**: the processor whose memory contains the actual value of the data
- Initially home = owner, but this can change …
- … if a processor other than the home processor writes a block
Inside a directory

- Each processor has a 1-bit “sharer” entry in the directory
- There is also a dirty bit and a PID identifying the owner in the case of a dirt block
Operation of a directory

- P0 loads A
- Set directory entry for A (on P1) to indicate that P0 is a sharer
Operation of a directory

- P2, P3 load A (not shown)
- Set directory entry for A (on P1) to indicate that P0 is a sharer
Acquiring ownership of a block

- P0 writes A
- P0 becomes the owner of A
Acquiring ownership of a block

- P0 becomes the owner of A
- P1’s directory entry for A is set to \textit{Dirty}
- Outstanding sharers are invalidated
- Access to line is blocked until all invalidations are acknowledged
Change of ownership

P0 stores into A (home & owner)
P1 stores into A (becomes owner)
P2 loads A

Store A, #y

Store A, #x
(home & owner)

Load A

Directory

©2013 Scott B. Baden / CSE 160 / Fall 2013
Forwarding

P0 stores into A (home & owner)
P1 stores into A (becomes owner)
P2 loads A
home (P0) forwards request to owner (P1)

Store A, #y
(home & owner)

Load A

©2013 Scott B. Baden / CSE 160 / Fall 2013
Today’s lecture

- Consistency
- NUMA
  - Example NUMA System: Cray XE-6
- Performance programming
Cray XE6 node

- 24 cores sharing 32GB main memory
- Packaged as 2 AMD Opteron 6172 processors “Magny-Cours”
- Each processor is a directly connected Multi-Chip Module: two hex-core dies living on the same socket
- Each die has 6MB of shared L3, 512KB L2/core, 64K L1/core
  - 1MB of L3 is used for cache coherence traffic
  - Direct access to 8GB main memory via 2 memory channels
  - 4 Hyper Transport (HT) links for communicating with other dies
- Asymmetric connections between dies and processors

www.nersc.gov/users/computational-systems/hopper/configuration/compute-nodes/
XE-6 Processor memory interconnect (node)

- Asymmetric connections between dies and processors
  - Direct access to 8GB main memory via 2 memory channels
  - 4 Hyper Transport (HT) links for communicating with other dies
XE-6 Processor memory interconnect (node)

http://www. Hector.ac.uk/cse/documentation/Phase2b/#arch
Fin