## Outline

- Bureaucracy: Timeline, Homework
- Review: Pthreads, OpenMP
- Synchronization
- Parallelizing loops
- Optimizations in Practice

## Synchronization

- High level synchronization included in the common core (the full OpenMP specification has *many* more):
  - Critical section
  - Barrier

## Synchronization: critical section

```
float  res;
#pragma omp parallel
{     float B;   int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+=nthrds){
          B =  big_job(i);
#pragma omp critical
          res += consume (B); // Thread wait: One at a time...
      }
}
```

## Synchronization: Barrier

- Barrier: a point in a program all threads much reach before any threads are allowed to proceed.
- It is a “stand alone” pragma meaning it is not associated with user code … it is an executable statement.

```
double Arr[8], Brr[8];
int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{   int id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      if (id==0) numthrds = nthrds;
      Arr[id] = big_ugly_calc(id, nthrds);
#pragma omp barrier
      Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

## Synchronization: Using a critical section to remove impact of false sharing

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 4
void main ()
{    int nthreads; double  pi=0.0;
     step = 1.0/(double) num_steps;
     omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
  {
    int i, id, nthrds;    double x, sum; // scalar sum per thread
    id = omp_get_thread_num();
              nthrds = omp_get_num_threads();
              if (id == 0)   nthreads = nthrds;
      for (i=id, sum=0.0;i< num_steps; i=i+nthrds) {
          x = (i+0.5)*step;
          sum += 4.0/(1.0+x*x); // no array, no sharing
      }
             #pragma omp critical
             pi += sum * step; // critical section avoids conflicts during updates
  }
}
```

## Parallelizing loops

- OpenMP easily parallelizes loops
  - Easiest when: No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor and runtime calculate loop bounds for each thread directly from serial source
- The loop **#pragma omp for** construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
{
#pragma omp for
   for (I=0;I<N;I++){ // I is made private to each thread
      big_ugly_calc(I);
   } // All threads wait here before proceeding
}
```

# Parallelizing loops (this time with feeling)

- Sequential: for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
- Parallel region:

```
#pragma omp parallel
{
    int id, i, Nthrds, istart, iend;
    id = omp_get_thread_num();
    Nthrds = omp_get_num_threads();
    istart = id * N / Nthrds;
    iend = (id+1) * N / Nthrds;
    if (id == Nthrds-1)iend = N;    for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i];}
}
```

- Parallel with for: (could put on the same line)

```
#pragma omp parallel
#pragma omp for
    for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

## Loop scheduling

- `schedule` clause determines how loop iterations are divided among the thread team; no one best way
- **`static([chunk])`** divides iterations statically between threads (default if no hint)
- Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
- Default **`[chunk]`** is **`ceil( # iterations / # threads )`**
- **`dynamic([chunk])`** allocates **`[chunk]`** iterations per thread, allocating an additional **`[chunk]`** iterations when a thread finishes
- Forms a logical work queue, consisting of all loop iterations
- Default **`[chunk]`** is 1
- **`guided([chunk])`** allocates dynamically, but **`[chunk]`** is exponentially reduced with each allocation

## Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
     j = 5;
     for (i=0;i< MAX; i++) {
         j +=2;
         A[i] = big(j);
    }
```

```
int i,  A[MAX];
    #pragma omp parallel for
     for (i=0;i< MAX; i++) {
         int j = 5 + 2*(i+1);
          A[i] = big(j);
    }
```

## Static vs. Dynamic

- Suppose you are doing block matrix multiplication, i.e. you split each matrix into $B^2$ submatrices, each of size N/B-by-N/B and will do $B^3$ submatrix multiplications (B is relatively small, say ~10)
- However, matrices are sparse and submatrices have varying number of nonzeros (this is a real use case).

```
#pragma omp parallel for
  for (int i=0;i< B; i++)
  {
     for (int j=0;j< B; j++)
     {
          for (int k=0;k< B; k++)
          {
                 SpGEMM(A(i,k), B(k,j), C(i,j)); // Sparse Matrix Multiply
          }
     }
}
```

# Static vs. Dynamic (again)

- Why did we use collapse? Without it, OpenMP only parallelizes the outermost loop
- We could put nested omp parallel constructs but that can create too much parallelism generation overhead
- Instead, **collapse**(2) says to the compiler to treat the whole B*B iteration space of the first two loops (following the program) as a single contiguous iteration space to divide up

```
#pragma omp parallel for collapse(2)
  for (int i=0;i< B; i++)
  {
     for (int j=0;j< B; j++)
     {
          for (int k=0;k< B; k++)
          {
                 SpGEMM(A(i,k), B(k,j), C(i,j));
          }
     }
}
```

## Static vs. Dynamic (again…)

- If chunks are very large and have variable work; there isn't much any schedule can do for you.
- But if chunks are rather small and have variable work, you should try dynamic

```
#pragma omp parallel schedule(dynamic) for collapse(2)
  for (int i=0;i< B; i++)
  {
     for (int j=0;j< B; j++)
     {
          for (int k=0;k< B; k++)
          {
                 SpGEMM(A(i,k), B(k,j), C(i,j));
          }
     }
}
```

## Reduction

- We are combining values into a single accumulation variable … there is a true dependence between loop iterations that can’t be trivially removed
- This is a very common … it is called a “reduction”.
- Support for reduction operations is included in most parallel programming environments.

```
double  ave=0.0, A[MAX];    int i;
   for (i=0;i< MAX; i++) {
         ave + = A[i];
   }
   ave = ave/MAX;
```

## Reduction

- OpenMP reduction clause: **reduction** (op : list)
  - Inside a parallel construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - Updates occur on the *local copy*
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in “list” must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
        ave + = A[i];
 } // implicit barrier here
 ave = ave/MAX;
```

## OpenMP: Reduction operands initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

| Operator | Initial Value |
|---|---|
| + | 0 |
| * | 1 |
| - | 0 |
| min | Largest pos number |
| max | Largest negative number |
| & | ~0 |
| \| | 0 |
| ^ | 0 |
| && | 1 |
| \|\| | 0 |

## Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
void main ()
{    int i;       double x, pi, sum = 0.0;
      step = 1.0/(double) num_steps;
      #pragma omp parallel
      {
           double x;
         #pragma omp for reduction(+:sum)
        for (i=0;i< num_steps; i++){
          x = (i+0.5)*step;
          sum = sum + 4.0/(1.0+x*x);
        }
       }
      pi = step * sum;
}
```

## The nowait clause

- Implicit barriers exist at the end of most (but not all) OpenMP constructs
- Barriers cause waiting (expensive).  You need to understand when they are implied and how to skip them when its safe to do so.

```
double A[big], B[big], C[big];
#pragma omp parallel
{
    int id=omp_get_thread_num();
    A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
    for(i=0;i<N;i++){C[i]=big_calc3(i,A);} // Implicit barrier here
#pragma omp for nowait
    for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); } // No barrier here
    A[id] = big_calc4(id);
} // Implicit barrier here due to parallel
```

## Memory model

- Shared memory programming model:
  - Most variables are shared by default
- Global variables are **shared** among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: Globally scoped variables, static
  - Both languages: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms(Fortran) or functions(C) called from parallel regions are **private**
  - Automatic variables within a statement block are **private**

## Data Sharing

```
double A[10];
  int main() {
  int index[10];
  #pragma omp parallel
        work(index);
  printf(“%d\n”, index[0]);
 }
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

- A, index and count are shared by all threads.
- temp is local to each thread

## Data sharing: Attributes

- One can selectively change storage attributes for constructs using the following clauses (note: list is a comma-separated list of variables)
  - shared(list)
  - private(list)
  - firstprivate(list)
- These can be used on parallel and for constructs … other than *shared* which can only be used on a parallel construct
- Force the programmer to explicitly define storage attributes
  - default (none)

## Data sharing: private

- **private(var)** creates a new local copy of var for each thread.
- The value of the private copies are *uninitialized*
- The value of the original variable is unchanged after the region

```
void wrong() {
   int tmp = 0;
#pragma omp parallel for private(tmp)
   for (int j = 0; j < 1000; ++j)
     tmp += j; // tmp wasn’t initialized here
   printf(“%d\n”, tmp); // tmp is 0 here! (outside of for scope)
}
```

## Data sharing: firstprivate

- Variables initialized from a shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) { // Each thread gets own incr with initial 0
    if ((i%2)==0) incr++;
    A[i] = incr;
}
```

## Data sharing: Summary

- Are A,B,C private to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

```
variables:  A = 1, B = 1, C = 1

#pragma omp parallel private(B)  firstprivate(C)
```

Inside this parallel region ...
● "A" is shared by all threads; equals 1
● "B" and "C" are private to each thread.
– B's initial value is undefined
– C's initial value equals 1

Following the parallel region ...
● B and C revert to their original values of 1
● A is either 1 or the value it was set to inside the parallel region

## Data sharing: default

- default(none): Forces you to define the storage attributes for variables that appear inside the static extent of the construct … if you fail the compiler will complain.   Good programming practice!
- The compiler would complain about j and y, which is important since you don’t want j to be shared

```
#include <omp.h>
int main()
{
    int i, j=5;      double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
       for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(“ x is %f\n”,(float)x);
}
```

## Tasks

- Tasks are *independent* units of work
- Tasks are composed of:
  - code to execute
  - data to compute with
- Threads are assigned to perform the work of each task.
- The thread that encounters the **task** construct may execute the task immediately.
- The threads may defer execution until later

## Tasks

- The task construct includes a structured block of code
- Inside a parallel region, a thread encountering a task construct will package up the code block and its data for execution
- Tasks can be nested: i.e. a task may itself generate tasks.

## Tasks: single

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the **single** block (can remove the barrier with a nowait clause).

```
#pragma omp parallel
{
   do_many_things();
#pragma omp single
   {     exchange_boundaries();   }
   do_many_other_things();
}
```

## Tasks: task directive

#pragma omp task [clauses] structured-block

```
#pragma omp parallel
{
  #pragma omp single
   {
       #pragma omp task
           fred();
       #pragma omp task
        daisy();
       #pragma omp task
          billy();
   } // All tasks complete before this barrier is released
}
```

## Tasks: When are they completed?

- At thread barriers (explicit or implicit)
  - applies to all tasks generated in the current parallel region up to the barrier
- At **taskwait** directive
  - i.e. wait until all tasks defined within the scope of the current task have completed.
  - #pragma omp taskwait
  - Note: applies only to tasks *generated in the current task, not to “descendants”* .
  - To also wait for descendents, there is the taskgroup *region*
  - When a thread encounters a taskwait construct, the current task is suspended until all child tasks that it generated before the taskwait region complete execution

## Tasks: task directive

#pragma omp task [clauses] structured-block

```
#pragma omp parallel
{
  #pragma omp single
   {
        #pragma omp task
            fred();
        #pragma omp task
         daisy();
        #pragma taskwait // fred() and daisy() must complete before billy() starts
        #pragma omp task
           billy();
   } // All tasks complete before this barrier is released
}
```

## Data sharing: task defaults

- The behavior you want for tasks is usually **firstprivate**, because the task may not be executed until later (and variables may have gone out of scope)
  - Variables that are **private** when the task construct is encountered are firstprivate by default
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared by default

```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
      int C;
      compute(A, B, C); // B is firstprivate, C is private
   }
}
```

## Serial Fibonacci Numbers

```
int fib (int n) {
   int x,y;
   if (n < 2) return n;

   x = fib(n-1);
   y = fib(n-2);
   return (x+y);
}

int main() {
   int NW = 5000;
   fib(NW);
}
```

- $Fn = F_{n-1} + F_{n-2}$
- Inefficient $O(2^n)$ recursive implementation!

# Parallel Fibonacci Numbers

```
int fib (int n) {   int x,y;

   if (n < 2) return n;

#pragma omp task shared(x)

   x = fib(n-1);

#pragma omp task shared(y)

   y = fib(n-2);

#pragma omp taskwait

   return (x+y);

}

int main() {  int NW = 5000;

   #pragma omp parallel

   {

       #pragma omp single

            fib(NW);

   }

}
```

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete (enforced with **taskwait**)
- By default, only 2 threads will be active in most implementations. Set OMP_MAX_ACTIVE_LEVELS with  n>1 to get n-levels of nested parallelism
- x,y are local, and so by default they are private to current task
  - must be shared on child tasks so they don’t create their own **firstprivate** copies at this level!

## Synchronization (slight return)

- High level synchronization:
  - critical
  - barrier
  - atomic
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Synchronization is used to impose order constraints and to protect access to shared data

## Synchronization: atomic

- Atomic provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
        double B;
        B =  DOIT();

 #pragma omp atomic // faster than critical
    X +=  big_ugly(B);
}
```

## Low-level synchronization: flush

- Defines a sequence point at which a thread enforces a consistent view of memory.
- For variables visible to other threads and associated with the flush operation (the flush-set)
  - The compiler can’t move loads/stores of the flush-set around a flush:
  - All previous read/writes of the flush-set by this thread have completed
  - No subsequent read/writes of the flush-set by this thread have occurred
  - Variables in the flush set are moved from temporary storage to shared memory.
  - Reads of variables in the flush set following the flush are loaded from shared memory.

## Low-level synchronization: flush

- A flush operation is implied by OpenMP synchronizations, e.g.,
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

# Takeaway

- Programming shared memory machines
  - May allocate data in large shared region without too many worries about where
  - Memory hierarchy is critical to performance
    - Even more so than on uniprocessors, due to coherence traffic
  - For performance tuning, watch sharing (both true and false)
- Semantics
  - Need to lock access to shared variable for read-modify-write
  - Sequential consistency is the natural semantics
    - Write race-free programs to get this
  - Architects worked hard to make this work
    - Caches are coherent with buses or directories
    - No caching of remote data on shared address space machines
  - But compiler and processor may still get in the way
    - Non-blocking writes, read prefetching, code motion…
    - Avoid races or use machine-specific fences carefully

## Shared Memory Programming

- PTHREADS is the POSIX Standard
  - Portable but; relatively heavyweight; low level
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory, openmp.org
- TBB: Thread Building Blocks
  - Intel C++ template library for parallel (multicore) computing
- Java threads
  - Built on top of POSIX threads; Object within Java language

# Parallel programming and Simulation

- Parallelism and data locality both critical to performance
  - Recall that moving data is the most expensive operation
- Real world problems have parallelism and locality:
  - Many objects operate independently of others.
  - Objects often depend much more on nearby than distant objects.
  - Dependence on distant objects can often be simplified.
  - Example of all three: particles moving under gravity
- Scientific models may introduce more parallelism:
  - When a continuous problem is discretized, time dependencies are generally limited to adjacent time steps.
    - Helps limit dependence to nearby objects (eg collisions)
  - Far-field effects may be ignored or approximated in many cases.
- Many problems exhibit parallelism at multiple levels

## Kinds of Simulation (from discrete to continuous)

- Discrete event systems:
  - “Game of Life,” Manufacturing systems, Finance, Circuits, Pacman, …
- Particle systems:
  - Billiard balls, Galaxies, Atoms, Circuits, Pinball …
- Lumped variables depending on continuous parameters: a.k.a. Ordinary Differential Equations (ODEs)
  - Structural mechanics, Chemical kinetics, Circuits, Star Wars: The Force Unleashed
- Continuous variables depending on continuous parameters: a.k.a. Partial Differential Equations (PDEs)
  - Heat, Elasticity, Electrostatics, Finance, Circuits, Medical Image Analysis, Terminator 3: Rise of the Machines

## Simulation outline

- Discrete event systems
  - Time and space are discrete
- Particle systems
  - Important special case of lumped systems
- Lumped systems (ODEs)
  - Location/entities are discrete, time is continuous
- Continuous systems (PDEs)
  - Time and space are continuous

## Circuit Simulation

| **Level** | **Primitives** | **Example** |
|---|---|---|
| Instruction | Instructions | SPIM |
| Cycle | Functional Units | VHDL/ Verilog |
| RTL | Registers, muxes, etc | VHDL/ Verilog |
| Gate | Gate, FF, memory | VHDL/ Verilog |
| Switch | Transistor | COSMOS |
| Circuit | Resistor, Capacitor | SPICE |
| Device | Electrons, Semiconductor | ... |

## A Model Problem: Sharks and Fish

- Illustration of parallel programming
- Original version (discrete event only) called WATOR
- Basic idea: sharks and fish living in an ocean
  - rules for movement (discrete and continuous)
  - breeding, eating, and death
  - forces in the ocean
  - forces between sea creatures
- 6 problems (S&F1 - S&F6)
  - Different sets of rules, to illustrate different phenomena

## A Model Problem: Sharks and Fish

- S&F 1. Fish alone move continuously subject to an external current and Newton's laws *(completely parallel)*
- S&F 2. Fish alone move continuously subject to gravitational attraction and Newton's laws
- S&F 3. Fish alone play the "Game of Life" on a square grid
- S&F 4. Fish alone move randomly on a square grid, with at most one fish per grid point *(need to keep at most one fish per cell, so need to deal with possible races, deadlock)*
- S&F 5. Sharks and Fish both move randomly on a square grid, with at most one fish or shark per grid point, including rules for fish attracting sharks, eating, breeding and dying
- S&F 6. Like Sharks and Fish 5, but continuous, subject to Newton's laws

## Discrete Event Systems

- Systems are represented as:
  - finite set of variables.
  - the set of all variable values at a given time is called the state.
  - each variable is updated by computing a *transition function* depending on the other variables.
- System may be:
  - synchronous: at each discrete timestep evaluate all transition functions; also called a state machine.
  - asynchronous: transition functions are evaluated only if the inputs change, based on an “event” from another part of the system; also called event driven simulation.
- Example: The “game of life:” (Conway)
  - Also known as Sharks and Fish #3:
  - Space divided into cells, rules govern cell contents at each step

## Parallelism in Game of Life (S&F 3)

- The simulation is synchronous
  - use two copies of the grid (old and new), “ping-pong” between them
  - the value of each new grid cell depends only on 9 cells (itself plus 8 neighbors) in old grid.
  - simulation proceeds in timesteps-- each cell is updated at every step.
- Easy to parallelize by dividing physical domain: Domain Decomposition
  - Locality is achieved by using large patches of the ocean
- Only boundary values from neighboring patches are needed.
- How to pick shapes of domains?

## Game of Life (S&F 3)

- Only need two grids: ping pong between them, avoid race conditions
- Load balanced because each processor gets equal #grid cells
- Square domains seem natural, but lots of choices in general, how choose?
- Rules:
  - Any live cell with two or three live neighbors survives.
  - Any dead cell with three live neighbors becomes a live cell.
  - All other live cells die in the next generation. Similarly, all other dead cells stay dead.

```
Repeat
    compute locally to update local system
    barrier()
    exchange state info with neighbors
    finish updates
until done simulating
```

## Synchronous Circuit Simulation

- Circuit is a graph made up of subcircuits connected by wires
  - Component simulations need to interact if they share a wire.
  - Data structure is (irregular) graph of subcircuits.
  - Parallel algorithm is timing-driven or synchronous:
    - Evaluate all components at every *timestep* (determined by known circuit delay)
- Graph partitioning assigns subgraphs to processors
  - Determines parallelism and locality.
  - Goal 1 is to evenly distribute subgraphs to nodes  (load balance).
  - Goal 2 is to minimize edge crossings (minimize communication).
  - Easy for meshes, NP-hard in general, so we will approximate (future lecture)

## Asynchronous Simulation

- Synchronous simulations may waste time:
  - Simulates even when the inputs do not change
- Asynchronous (event-driven) simulations update only when an event arrives from another component:
  - No global time steps, but individual events contain time stamp.
  - Example: Game of life in loosely connected ponds (don’t simulate empty ponds).
  - Example: Circuit simulation with delays (events are gates changing).
  - Example: Traffic simulation (events are cars changing lanes, etc.).
- Asynchronous is more efficient, but harder to parallelize
  - On distributed memory, events are naturally implemented as messages between processors (eg using MPI), but how do you know when to execute a “receive”?

# Scheduling Asynchronous Circuit Simulation

- Conservative:
  - Only simulate up to (and including) the minimum time stamp of inputs.
  - Need deadlock detection if there are cycles in graph
  - Example: Pthor circuit simulator
- Speculative (or Optimistic):
  - Assume no new inputs will arrive and keep simulating.
  - May need to backup if assumption wrong, using timestamps
- Example: Timewarp [D. Jefferson], Parswec [Wen,Yelick].
- Optimizing load balance and locality is difficult:
  - Locality means putting tightly coupled subcircuit on one processor.
  - Since “active” part of circuit likely to be in a tightly coupled subcircuit, this may be bad for load balance.

## Summary of Discrete Event Simulations

- Model of the world is discrete
  - Both time and space
- Approaches
  - Decompose domain, i.e., set of objects
  - Run each component ahead using
    - Synchronous: communicate at end of each timestep
    - Asynchronous: communicate on-demand
      - Conservative scheduling – wait for inputs
        - need deadlock detection
      - Speculative scheduling – assume no inputs
        - roll back if necessary