CS315A/EE386A: Lecture 5

Parallel Programming Tips and Analysis of Parallel Applications

Kunle Olukotun
Stanford University

http://eeclass.stanford.edu/cs315a

Announcements

- PA1
  – Due Mon April 24
- PS1
  – Out today
  – Due Wed April 26
- New information sheet
  – Change in one of readings

Today's Outline: Tips and Analysis

- Some common parallel programming issues
  - Sharing & memory allocation
  - False sharing
  - Locking & deadlock
- Basic parallel application analysis
  - Speedup & Timing
  - Overheads & Efficiency
- Analysis of sequential and communication overheads
- Scalability of applications
  - How do you vary your dataset as *num_procs* increases?

Parallel Programming is Tough!

- A lot is happening *simultaneously* in any parallel program
  - Computation
  - Data communication
  - Locking & synchronization
  - It's easy to develop race conditions among all of these
- Bugs are hard to track down & find
  - Are often dependent upon runtime alignment of CPUs
    - So they can appear & disappear
  - Almost always differs from run to run
    - OS activity & interrupts occur at varying times
  - Even *reproducing* an error may be difficult

Being Systematic is *Crucial*

- Get your program to work on one processor
  - Make sure the algorithm, math, etc. is correct
  - Don't forget calculations based on *num_procs*!
- Insert synchronization
  - Lock/unlock pairs
  - Barriers (as many as possible, at first)
- Test carefully, with lots of `printfs`
  - Most debuggers only work on one thread at a time
    - Causes "debugged" thread to run "slowly" compared with others
  - So you need to print your own messages from *all* of them
    - Arriving at barriers (to make sure we all get there)
    - Around critical regions (check for deadlock)
    - Key data values (make sure they're reasonable)

Memory Sharing

- Be careful about what is *implicitly* shared or private
- *All* variables are implicitly private in heavyweight thread models
- Globals are implicitly *shared* in lightweight-thread models
  - Both pthreads and OpenMP are this way
- Stack-allocated variables can vary
  - pthreads: Implicitly private stacks, sharing is dangerous!
  - OpenMP: Depends on parallel region & default settings
- Heap-allocated variables are implicitly shared
  - Your control of pointers controls actual sharing

Memory Deallocation Hazards

- Be very wary of `&(`*stack_variable*`)` in threaded code
  - **Never** pass this pointer-to-the-stack to another thread
  - If original thread hits a `return`, variable will be deallocated
  - But other threads won't know about deallocation!
- Use the C `free` or C++ `delete` very carefully
  - Need to make sure **all** threads are done with memory first
  - . . . Or some may continue to use after the deallocation
  - Best to wait until *after* a barrier into a new phase
    - If the variable can't be accessed during the new phase
  - But can do while under "lock" protection
    - Make sure that you've locked **all** pointers to the memory block
    - One of the reasons why Java is popular for threaded programs

False Sharing

- Your program works, but seems slow. What's happening?
- You can get communication when you don't expect it!
  - Shared memory machines group variables into *cache lines*
  - All of these variables "act" like one larger variable . . . .
- A quick introduction to cache coherence:
  - Hardware acts as if it has a "R/W lock" on each line
  - Private, exclusive use is OK
  - Sharing read-only data is OK
  - ONLY ONE writer at a time!
- So we MUST isolate *actively written* variables
  - If not, other variables in the line will be "written" too!
    - At least as far as communication overhead is concerned

Which is better?

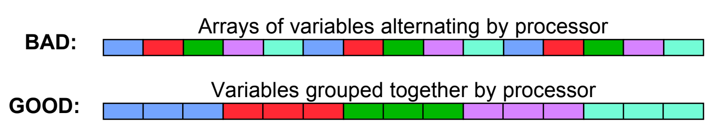
```
int sum[NUM_PROCS];
int product[NUM_PROCS];
. . .
int myData[NUM_PROCS];

. . .

sum[myNum]++;
product[myNum]*=2;
```

```
typedef struct
{
   int sum;
   int product;
   . . .
   int myData;
} Proc;
Proc x[NUM_PROCS];

. . .

x[myNum].sum++;
x[myNum].product*=2;
```


Avoiding False Sharing I

- Want variables written by different processors on *different* lines
- Want variables written by one processor *together*

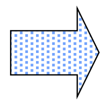
- For known private variables:
  - Sort them into groups *by processor*
  - Allocating variables on different stacks is GOOD
  - "Struct" of variables/processor is GOOD
  - Allocating arrays by [num_proc] is BAD



Avoiding False Sharing II

- Similar rules apply to other, more global data structures
  - Group variables read/written simultaneously together
  - Keep variables read/written at different times apart
  - Use arrays-of-structs, and not many arrays:

```
unsigned char R[Y_SIZE][X_SIZE];
unsigned char G[Y_SIZE][X_SIZE];
unsigned char B[Y_SIZE][X_SIZE];
int alpha[Y_SIZE][X_SIZE];
```

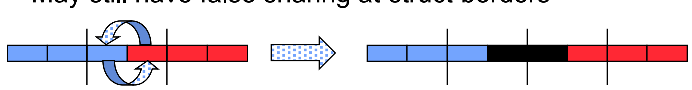


```
typedef struct
{
   unsigned char R, G, B;
   int alpha;
} Pixel;
Pixel frame[Y_SIZE][X_SIZE];
```

- Note that these tactics also help increase spatial locality, too!
  - Can even speed up single-processor code

Variable Padding

- Simply grouping variables may not be enough
  - May still have false sharing at struct borders



- In these cases, we need *padding* in our structs:
  - Or convert simple arrays to structs (now size-expensive!)

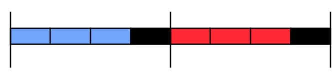
```
typedef struct {
   int sum;
   int product;
   . . .
   int myData;
   char pad[LINE_SIZE];
} Proc;
Proc x[NUM_PROCS];
```

```
typedef struct
{
   int value;
   char pad[LINE_SIZE];
} PaddedInt;

PaddedInt sum[NUM_PROCS];
PaddedInt product[NUM_PROCS];
```

Variable Alignment-and-Padding I

- Simple padding can waste a lot of space
  - Extra cache-line size block per variable/struct

- *Aligning* variables can allow us to minimize waste
  - Make sure that variables start at start of cache lines
  - Pad is now only **LINE_SIZE – sizeof(variable)**
  - Can save lots of space with structs just smaller than lines



Variable Alignment-and-Padding II

Declaration:

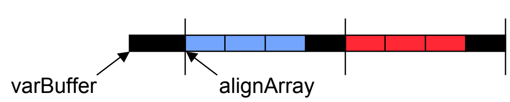
```
typedef struct
{
      int x, y, z;
      char pad[LINE_SIZE - 3*sizeof(int)];
} AlignVar;
char *varBuffer; AlignVar *alignArray;
```

Allocation:

```
varBuffer = malloc(SIZE*sizeof(AlignVar) + LINE_SIZE);
alignArray = (varBuffer & ~(LINE_SIZE-1)) + LINE_SIZE;
```

Use as: `alignArray[i]`

Deallocation: `delete(varBuffer);`



Using Locks Effectively

- We need to be careful to avoid over- or under- locking
  - Too few locks will increase contention
    - More processor time will be spent stalled waiting for locks
  - Too many locks will increase overhead
    - More memory delay to load in lock structures
    - More overhead code to lock
  - Need to balance between these
    - *Rule of Thumb:* # locks proportional to # of processors
    - Exact numerical relationship depends upon:
      - % of time that locks are locked
      - Size of locked regions
- Put locks in the same struct as the “protected” variables
  - Typically will be accessed *together*

Avoiding Deadlock I

- Locks are simple when used one at a time
  - Contention simply causes queueing for lock
  - Need to minimize lock critical regions or use more locks
- Locking 2 locks at once can risk *deadlock*!
  - *Generally* nest locking of second within first
    - Non-nested critical regions are a messy topic . . .
  - Can **cross-lock** if we lock in opposite order

Processor 1:	**Processor 2:**
`LOCK(lockOne);`	`LOCK(lockTwo);`
`LOCK(lockTwo);`	`LOCK(lockOne);`
. . .	. . .

Avoiding Deadlock II

- First line of defense: *single locking order*
  - Keep a list of your locks as you program
  - Always do nested lock-acquires in order of your list
- Second line of defense: *backing off locks*
  - May not be able to globally order some locks
    - Example: Locks on parts of irregular graph structures
  - Have to use code that can *escape* deadlock:

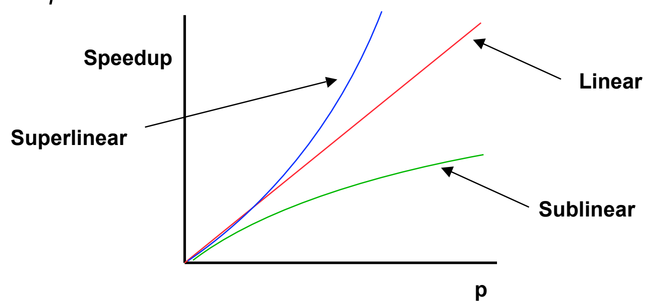
```
locked = FALSE;
while (!locked) {
 LOCK(lockOne);
 if (TRY_LOCK(lockTwo)) locked = TRUE;
        else { UNLOCK(lockOne); delay(BACKOFF_TIME); }
}
```


Avoiding Deadlock III

- Make sure that your locks are *nested properly*
  - We recommend that you use just like {}s
- Beware of "forgotten locks"
  - Make sure that you don't break out of a locked region!
  - Beware of: `break`, `return`, `longjmp`, `goto`
  - Can leave you with a "forgotten" set lock
  - *Next* use of lock will cause deadlock
    - May be MUCH later in the program's execution
    - VERY difficult to debug!
  - May want to search for these words in locked region

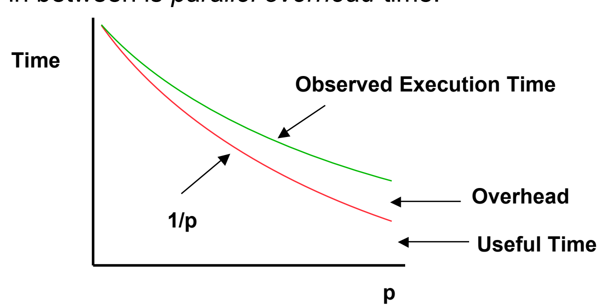
I've got execution times. Now what?

- Plot speedups: $S = T_{serial}/T_{parallel} = T_S/T_P$ vs. p = # of processors
  - Results are "mortar shot" speedup plots
  - *Linear:* Perfectly scalable application
  - *Sublinear:* Not infinitely scalable (usual case)
  - *Superlinear:* Occasional effect of more cache



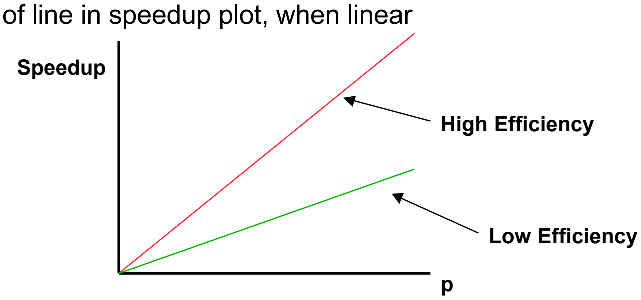
Or just plot times . . .

- Speedup is usually most interesting . . .
- . . . But times can show useful information, too
  - Plot times as a line vs. p
  - Plot *useful* execution as a known 1/p line
  - Space in between is *parallel overhead* time!



Overhead Analysis

- Parallel overhead (T_O) is our enemy
  - Represents wasted time on parallel processors
  - Difference between perfect linear & sublinear speedup
- Can also view in terms of parallel efficiency (E = S/p)
  - Represents % of time used usefully
  - Slope of line in speedup plot, when linear



What is Overhead?

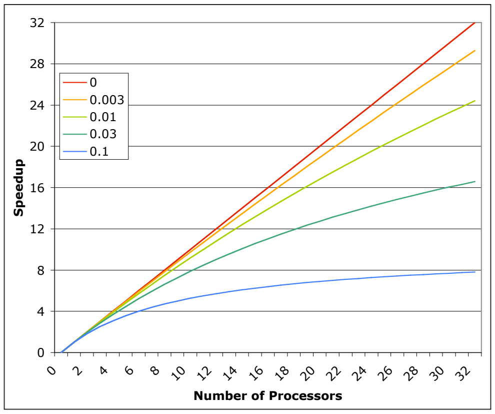
- Overhead (T_O) consists of two components
- "Sequential" code time (t_s)
  - Some processors are *idling*
  - Time spent on non-parallel code
  - Time spent repeating code on all processors
  - Time spent waiting at locks, barriers, etc.
  - Low-concurrency code (load imbalance)
- Communication overhead time (t_c)
  - Time wasted waiting for remote data to arrive
  - Scales in a system- and algorithm-dependent manner

Sequential Overhead

- Sequential overhead can be deadly
  - With large p, even a small sequential region can kill speedup
  - Amdahl's Law!

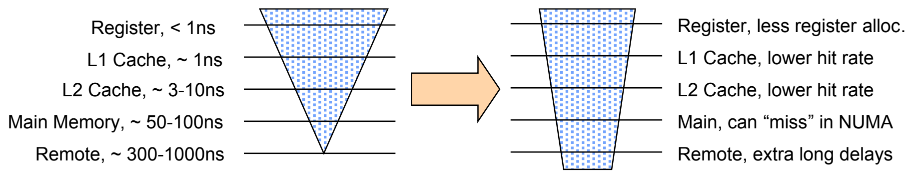
$$Speedup = \frac{1}{f_s + \frac{1 - f_s}{p}}$$

$$f_s = \frac{\frac{1}{Speedup} - \frac{1}{p}}{1 - \frac{1}{p}}$$



Communication Overhead

- Communication time is another parallel overhead
  - Two components in message passing:
    - Time to send/receive a message
    - Time spent stalled waiting at receive
  - Appears as "memory latency" in shared memory
    - Extra main memory accesses in UMA systems
      - Must determine lowering of cache miss rate vs. uniprocessor
    - Some accesses have higher latency in NUMA systems
      - Only a fraction of a % of these can be significant!



Computation-to-Communication Ratio

- Basic sequential overhead is fairly constant
  - Uniprocessor & replicated code times not a function of p
  - Must minimize these code blocks!
- Rest of sequential overhead can often be tuned away
  - Adjust static and dynamic tasks to balance load
  - Adjust locking structure to eliminate contention
  - Both may be affected by p
- But communication is *inherent* in an algorithm
  - Cannot be tuned away . . . only algorithm change can help
  - Thus C-to-C ratio = T_P/t_c is important for any *algorithm*

Some Sample C/C Ratios

- Dominant orders shown (n = *total* data size):

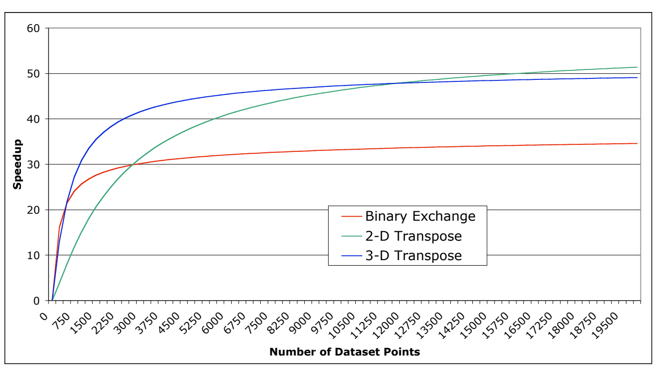
Application	Scaling of Computation	Scaling of Communication	Scaling of C/C Ratio
Matrix Multiply (striped)	$\frac{n^{\frac{3}{2}}}{p}$	n	$\frac{\sqrt{n}}{p}$
Matrix Multiply (blocked)	$\frac{n^{\frac{3}{2}}}{p}$	$\frac{n}{\sqrt{p}}$	$\sqrt{\frac{n}{p}}$
Ocean (striped)	$\frac{n}{p}$	$\sqrt{n}$	$\frac{\sqrt{n}}{p}$
Ocean (blocked)	$\frac{n}{p}$	$\sqrt{\frac{n}{p}}$	$\sqrt{\frac{n}{p}}$
LU	$\frac{n}{p}$	$\sqrt{\frac{n}{p}}$	$\sqrt{\frac{n}{p}}$
1-D FFT (using 2-D transpose)	$\frac{n \log n}{p}$	$\frac{n}{p}$	$\log n$

Using Computation/Communication Ratios

- Larger C/C ratios are usually better
  - More computation for every value communicated
  - More work to help "hide" communication latencies
  - Less likely for communication to be significant
  - Advantage of blocking is evident
- But be careful with small n
  - C/C ratio only approaches these *asymptotically*
  - Other factors may have more effect with small n
  - Less "scalable" algorithm may be better with small n

Comparison of Competing Algorithms

- FFT shows such a tradeoff
  - Binary exchange best with very small n
  - 3-D and then 2-D transpose best with larger n



Scalability: What is it?

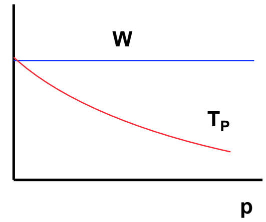
- Over Time:
  - Computer systems become larger and more powerful
    - More & more powerful processors
    - Also range of system sizes within a product family
  - Problem sizes become larger
    - Simulate the entire plane rather than the wing
  - Required accuracy becomes greater
    - Forecast the weather a week in advance rather than 3 days
- Scaling:
  - How do algorithms and hardware behave as systems, size, accuracies become greater?

Measuring "Scalability"

- We need to measure a "scaling performance"
- How do we *measure* it?
  - Depends upon how we *define* it
  - Need to measure "parallel speedup" for *our definition of work*
  - Different versions vary *parallel work* W(p) differently
- Several common ways to measure scalability:
  - Constant dataset ("Problem-constrained")
  - Dataset scaled by p ("Memory-constrained")
  - Constant time ("Time-constrained")
  - Constant efficiency ("Isoefficient")

Constant Dataset ("Problem Constrained")

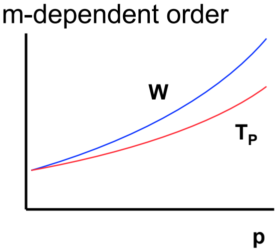
- This is the usual baseline used for speedup
  - Work is constant, W(n, p) = K
  - Execution time decreases by up to 1/p



- **Pros:**
  - Very simple to perform
  - Shows how parallel processors improve upon uniprocessors
- **Cons:**
  - Can run out of useful parallel work with large p
  - More hardware, so caches can cause superlinear speedup
  - Large parallel machines are rarely used as simple uniprocessor replacements

Scaled Dataset ("Memory Constrained")

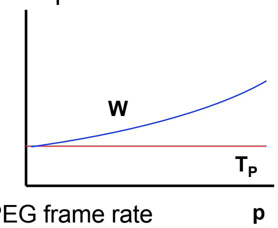
- Multiply the "base" data set size by p
  - Work function W(np,p) increases, algorithm-dependent order
  - Time increases with W(np,p)/p



- **Pros:**
  - Often easy to do
    - Just multiply some constants by p
  - Cache effects are essentially eliminated
  - Well-matched to most NUMA systems
    - Memory size scales with the processor count in these systems
- **Cons:**
  - Can result in loooong runs with high-order W(p) functions
  - Not as good with UMA systems, could be unrealistic

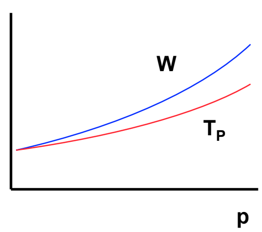
Constant Time ("Time Constrained")

- Keep the execution time T_P constant
  - Work function W(n, p) increases, algorithm-dependent order
  - Time stays constant
- **Pros:**
  - Cache effects are reduced dramatically
  - Useful in many useful contexts:
    - Hard limits on execution time, such as MPEG frame rate
    - How long "average user" will wait for results
- **Cons:**
  - "n" must be varied by inverting the work function
  - How do we do this? (vary data, timesteps, etc. . . . ?)



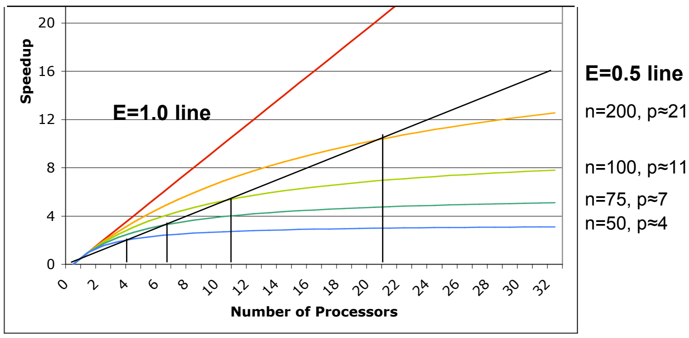
Constant Efficiency ("Isoefficient")

- Keep the efficiency linear *and* constant
  - Work function W(n, p) increases, algorithm-dependent order
  - Time usually increases, algorithm-dependent order
- **Pros:**
  - Cache effects are reduced dramatically
  - Good scaling technique, in general
    - Can pick any desired efficiency level
- **Cons:**
  - Can result in loooong runs with high-order W(p) functions
  - "n" must be varied by inverting the work function
  - Must solve the equation: $W(n,p) = \left(\frac{E}{1-E}\right) T_O(n,p)$



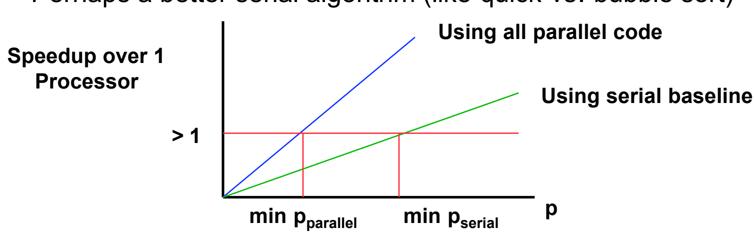
Isoefficiency Interpretation

- Draw speedup curves for different datasets together
- Draw a line through them, slope = E
- Intersections indicate isoefficient data set sizes



"Serial" Execution Time

- **WARNING:** Even "serial time" definition can vary
  - T_S is generally obtained by running parallel code on 1 CPU
  - Good for showing speedup trends
  - But bad for making parallel-vs.-serial choices
- Really need to run *serial* code on uniprocessor
  - No synchronization code overhead
  - Perhaps a better serial algorithm (like quick-vs.-bubble sort)



Summary & A Look Ahead

- We have examined some common sources of errors
  - Inter-thread memory access errors
  - False sharing
  - Problems with locking
- We went through the process of *analyzing* speedups
  - Determine speedup & efficiency
  - Examine sources of overhead
  - Look at scalability of applications
- Will next look at some *real* applications in more detail