18-447 Lecture 23: Illusiveness of Parallel Performance

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – peel back simplifying assumptions to understand parallel performance (or the lack of)

• Notices
  – Final Exam, Tuesday, 5/7, 1pm~4pm
    Resolve final exam conflicts this week!!

• Readings
  – P&H Ch 6
  – LogP: a practical model of parallel computation, Culler, et al. (advanced optional)
“Ideal” Linear Parallel Speedup

- Ideally, parallel speedup is linear with $p$

$$\text{Speedup} = \frac{\text{time}_{\text{sequential}}}{\text{time}_{\text{parallel}}}$$

\[\propto \frac{1}{p}\]

\[\propto p\]
Non-Ideal Speed Up

Never get to high speedup regardless of $p$!!
Parallelism Defined

- \( T_1 \) (work measured in time):
  - time to do work with 1 PE
- \( T_\infty \) (critical path):
  - time to do work with infinite PEs
  - \( T_\infty \) bounded by dataflow dependence
- Average parallelism:
  \[ P_{avg} = \frac{T_1}{T_\infty} \]
- For a system with \( p \) PEs
  \[ T_p \geq \max\{ \frac{T_1}{p}, T_\infty \} \]

When \( P_{avg} >> p \)
\[ T_p \approx \frac{T_1}{p}, \text{aka "linear speedup"} \]
Amdahl’s Law

- If only a fraction $f$ (by time) is parallelizable by $p$

\[
\text{time}_{\text{parallelized}} = \text{time}_{\text{sequential}} \cdot \left( (1-f) + \frac{f}{p} \right)
\]

\[
S_{\text{effective}} = \frac{1}{(1-f) + \frac{f}{p}}
\]

- if $f$ is small, $p$ doesn’t matter
- even when $f$ is large, diminishing return on $p$; eventually “1-f” dominates
Non-Ideal Speed Up

limited scalability, $P_{avg} < p$

Never get to high speedup regardless of $p$!!
Non-Ideal Speed Up

This may be acceptable if it is the best you can do
Parallelization Built-In Cost

• Cheapest algo may not be the most scalable, s.t.

\[
\text{time}_{\text{parallel-algo@p=1}} = K \cdot \text{time}_{\text{sequential-algo}} \quad \text{and} \quad K > 1
\]

and

\[
\text{Speedup} = \frac{p}{K} \quad \text{“parallel slowdown”}
\]

• Communication not free
  – PE may spend extra time in the act of sending or receiving data
  – PE may spend extra time waiting for data to be transferred from another PE
  – PE may spend extra time waiting for another PE to get to a particular point of the computation (a.k.a. synchronization)
Aside: Strong vs. Weak Scaling

• Strong Scaling (assumed so far)
  – what is $S_p$ as $p$ increases for constant work, $T_1$
    
    $\text{run same workload faster on new larger system}$
  – harder to speedup as (1) $p$ grows toward $P_{\text{avg}}$ and
    (2) communication cost increases with $p$

• Weak Scaling
  – what is $S_p$ as $p$ increases for larger work, $T_1' = p \cdot T_1$
    
    $\text{run a larger workload faster on new larger system}$
  – $S_p = \frac{\text{time}_{\text{sequential}}(p \cdot T_1)}{\text{time}_{\text{parallel}}(p \cdot T_1)}$

• Which is easier depends on
  – how $P_{\text{avg}}$ scales with work size $T_1'$
  – scaling of bottlenecks ($\text{storage, BW, etc}$)
Continuing from Last Lecture

• Parallel Thread Code (Last Lecture)

```c
void *sumParallel(void *id) {
    long id=(long) id;
    psum[id]=0;
    for(long i=0;i<(ARRAY_SIZE/p);i++)
        psum[id]+=A[id*(ARRAY_SIZE/p) + i];
}
```

• Assumed “+” takes 1 unit-time; everything else free

\[
T_1 = 10,000 \\
T_\infty = \lceil \log_2 10,000 \rceil = 14 \\
P_{\text{average}} = 714
\]

What would you predict is the real speedup on a 28-core ECE server?
Need for more detailed analysis

- What cost were left out in “everything else”?  
  - explicit cost: need to charge for all operations (branches, LW/SW, pointer calculations . . . .)  
  - implicit cost: **communication and synchronization**

- PRAM-like models (Parallel Random Access Machine) capture cost/rate of parallel processing but assume  
  - zero latency and infinite bandwidth to share data between processors  
  - zero overhead cycles to send and receive

Useful when analyzing complexity but not for performance tuning
Arithmetic Intensity: Modeling Communication as “Lump” Cost
Arithmetic Intensity

- An algorithm has a cost in terms of operation count
  \[ \text{runtime}_{\text{compute-bound}} = \frac{\# \text{ operations}}{\text{FLOPS}} \]

- An algorithm also has a cost in terms of number of bytes communicated (ld/st or send/receive)
  \[ \text{runtime}_{\text{BW-bound}} = \frac{\# \text{ bytes}}{\text{BW}} \]

- Which one dominates depends on
  - ratio of FLOPS and BW of platform
  - ratio of ops and bytes of algorithm

- Average Arithmetic Intensity (AI)
  - how many ops performed per byte accessed
  - \# operations / \# bytes

FLOPS=floating-point operations per second
Roofline Performance Model

[Williams&Patterson, 2006]

Attainable Performance of a system (op/sec)

\[
\text{runtime} > \max \left( \frac{\# \text{op}}{\text{FLOPS}}, \frac{\# \text{byte}}{\text{BW}} \right) \\
> \# \text{op} \cdot \max \left( \frac{1}{\text{FLOPS}}, \frac{1}{(\text{AI} \cdot \text{BW})} \right)
\]

\[
\text{perf} = \min (\text{FLOPS}, \text{AI} \cdot \text{BW})
\]
Parallel Sum Revisited with AI

• Last lecture we said
  – 100 threads perform 100 +’s each in parallel, and
  – between 1~7 (plus a few) +’s each in the parallel reduction
  – $T_{100} = 100 + 7$
  – $S_{100} = 93.5$

• Now we see
  – AI is a constant, 1 op / 8 bytes (for doubles)
  – Let BW be total bandwidth (byte/cycle) shared by threads on a multicore
    Performance
    $$Perf_p < \min\{ p \text{ ops/cycle, } AI \times BW \}$$
  – useless to parallelize beyond $p > BW/8$

What about a multi-socket system?
Interesting AI Example: MMM

for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    for(k=0; k<N; k++)
      C[i][j]+=A[i][k]*B[k][j];

• $N^2$ data-parallel dot-product’s
• Assume $N$ is large s.t. 1 row/col too large for on-chip
• Operation count: $N^3$ float-mult and $N^3$ float-add
• External memory access (assume 4-byte floats)
  – $2N^3$ 4-byte reads (of $A$ and $B$) from DRAM
  – $\ldots$ $N^2$ 4-byte writes (of $C$) to DRAM $\ldots$
• Arithmetic Intensity $\approx 2N^3/(4 \cdot 2N^3)=1/4$

GTX1080: 8 TFLOPS vs 320GByte/sec
More Interesting AI Example: MMM

\[
\text{for}(i0=0; \ i0<N; \ i0+=N_b) \\
\quad \text{for}(j0=0; \ j0<N; \ j0+=N_b) \\
\quad \quad \text{for}(k0=0; \ k0<N; \ k0+=N_b) \quad \{ \\
\quad \quad \quad \text{for}(i=i0; \ i<i0+N_b; \ i++) \\
\quad \quad \quad \quad \text{for}(j=j0; \ j<j0+N_b; \ j++) \\
\quad \quad \quad \quad \quad \text{for}(k=k0; \ k<k0+N_b; \ k++) \\
\quad \quad \quad \quad \quad \quad C[i][j] += A[i][k] * B[k][j]; \\
\quad \quad \}\ \\
\}
\]

• Imagine a ‘\(\frac{N}{N_b}\)’x’‘\(\frac{N}{N_b}\)’ MATRIX of \(N_b \times N_b\) matrices
  – inner-triple is straightforward matrix-matrix mult
  – outer-triple is MATRIX-MATRIX mult
• To improve AI, hold \(N_b \times N_b\) sub-matrices on-chip for data-reuse
  need to copy block (not shown)
AI of blocked MMM Kernel ($N_b \times N_b$)

```c
for (i = i0; i < i0 + N_b; i++)
    for (j = j0; j < j0 + N_b; j++) {
        t = C[i][j];
        for (k = k0; k < k0 + N_b; k++)
            t += A[i][k] * B[k][j];
        C[i][j] = t;
    }
```

- Operation count: $N_b^3$ float-mul and $N_b^3$ float-add
- When $A$, $B$ fit in scratchpad ($2 \times N_b^2 \times 4$ bytes)
  - $2N_b^3$ 4-byte on-chip reads ($A$, $B$) (fast)
  - $3N_b^2$ 4-byte off-chip DRAM read $A$, $B$, $C$ (slow)
  - $N_b^2$ 4-byte off-chip DRAM writeback $C$ (slow)
- Arithmetic Intensity = $2N_b^3 / (4 \cdot 4N_b^2) = N_b / 8$
• **AI** is a function of algorithm and problem size
• Higher **AI** means more work per communication and therefore easier to scale
• Recall strong vs. weak scaling
  – strong=increase perf on fixed problem sizes
  – weak=increase perf on proportional problem sizes
  – weak scaling easier if **AI** grows with problem size
LogP Model:
Components of Communication Cost
LogP

- A parallel machine model with explicit communication cost
  - **Latency**: transit time between sender and receiver
  - **overhead**: time used up to setup a send or a receive (cycles not doing computation)
  - **gap**: wait time in between successive send’s or receive’s due to limited transfer bandwidth
  - **Processors**: number of processors, i.e., computation throughput

[Diagram]

- **overhead**\(_{tx}\) from CPU to NI
- **latency** from NI to interconnect
- **overhead**\(_{rx}\) from NI to CPU
- **gap**\(_{tx}\) from CPU to NI
- **gap**\(_{rx}\) from NI to CPU
Message Passing Example

```c
if (id==0) //assume node-0 has A initially
    for (i=1;i<p;i=i+1)
        SEND(i, &A[SHARE*i], SHARE*sizeof(double));
else
    RECEIVE(0,A[]) //receive into local array

sum=0;
for(i=0;i<SHARE;i=i+1) sum=sum+A[i];

remain=p;
do {
    BARRIER();
    half=(remain+1)/2;
    if (id>=half&&id<remain) SEND(id-half,sum,8);
    if (id<(remain/2)) {
        RECEIVE(id+half,&temp);
        sum=sum+temp;
    }
    remain=half;
} while (remain>1);
```

SHARE=HOWMANY/p

[based on P&H Ch 6 example]
Parallel Sum Revisited with LogP

How long?

1: if (id==0)
2: for (i=1;i<100;i=i+1)
3: SEND(i, &A[100*i], 100*sizeof(double));
4: else RECEIVE(0, A[])  

• assuming no back-pressure, node-0 finishes sending to node-99 after 99× overhead of SEND( )
• first byte arrives at node-99 some network latency later
• the complete message arrives at node-99 after 100*sizeof(double)/network_bandwidth
• node-99 finally ready to compute after the overhead to RECEIVE( )

What if 100*sizeof(double)/network_bandwidth greater than the overhead to SEND( )?
Parallel Sum Revisited with LogP

• ideally, this step is computed $p=100$ times faster than summing 10,000 numbers by one processor

• big picture thinking, e.g.,
  – is the time saved worth the data distribution cost?
  – if not, actually faster if parallelized less

• fine-tooth comb thinking, e.g.,
  – node-1 begins work first; node-99 begins work last
    ⇒ minimize overall finish time by assigning more work to node-1 and less work to node-99
  – maybe latency and bandwidth are different to different nodes

Performance tuning is a real craft
do {
    \textbf{BARRIER}();
    \texttt{half=}(\texttt{remain}+1)/2;
    \texttt{if (id>=half&&id<remain)} {
        \texttt{SEND(id-half,sum,8);}
    }
    \texttt{if (id<(\texttt{remain}/2))} {
        \texttt{RECEIVE(id+half,&temp);}
        \texttt{sum=sum+temp;}
    }
    \texttt{remain=half;}
} while (\texttt{remain}>1);
Parallel Sum Revisited with LogP

BARRIER();
do {
    BARRIER();
    half=(remain+1)/2;
    if (id>=half&&id<remain) SEND(id-half,sum,8);
    if (id<(remain/2)) {
        RECEIVE(id+half,&temp);
        sum=sum+temp;
    }
    remain=half;
} while (remain>1);

- how does one build a BARRIER()?
- do we need to synchronize each round?
- is this actually faster than if all nodes sent to node-0?

What if \( p \) is small? What if \( p \) is very large?

Real answer is a combination of techniques
LogP applies to shared memory too

When $C_0$ is reading $psum[0+half]$, the value originates in the cache of $C_{"\text{half}"}$
- $L$: time from $C_0$’s cache miss to when data retrieved from the cache of $C_{"\text{half}"}$ (via cache coherence)
- $g$: there is a finite bandwidth between $C_0$ and $C_{"\text{half}"}$
- $o$: as low as a LW instruction but also pay for stalls

```c
do {
    pthread_barrier_wait(…);
    half=(remain+1)/2;
    if (id<(remain/2))
    
        psum[id]=psum[id]+
        psum[id+half];
    
    remain=half;
} while (remain>1);
```
Implications of Communication Cost

- Large $g$—can’t exchange a large amount of data
  - must have lots of work per byte communicated
  - only scalable for applications with high AI
- Large $o$—can’t communicate frequently
  - can only exploit coarse-grain parallelism
  - if DMA, amount of data not necessarily limited
- Large $L$—can’t send data at the last minute
  - must have high average parallelism (more work/time between production and use of data)
- High cost in each category limits
  - the kind of applications that can speed up, and
  - how much they can speed up
Parallelization not just for Performance

- Ideal parallelization over $N$ CPUs
  - $T = \frac{Work}{k_{perf} \cdot N}$
  - $E = (k_{switch} + \frac{k_{static}}{k_{perf}}) \cdot Work$
    - $N$-times static power, but $N$-times faster runtime
  - $P = N \left( k_{switch} \cdot k_{perf} + k_{static} \right)$

- Alternatively, forfeit speedup for power and energy reduction by $s_{freq} = \frac{1}{N}$ (assume $s_{voltage} \approx s_{freq}$ below)
  - $T = \frac{Work}{k_{perf}}$
  - $E'' = (k_{switch} / N^2 + \frac{k_{static}}{(k_{perf} \cdot N)}) \cdot Work$
  - $P'' = k_{switch} \cdot k_{perf} / N^2 + \frac{k_{static}}{N}$

Also works with using $N$ slower-simpler CPUs