CS 4230, Parallel Computing
Basic Terminology,
Scaling Laws

Ganesh Gopalakrishnan

School of Computing, Univ of Utah

Jan 10, 2017

The world of computing has changed

Complexity

- Power dissipation is a first-rate concern
  - Packaging costs, cooling, fan
- Energy bills add up in large-scale installations
  - "Race to finish" (compute fast, idle) reduces energy, time
- Data movement costs energy
- One can compute thousands of CPU operations with the same energy one spends moving data even a short distance
- Chip dimensions are approaching 2cm
- Light travels 30cm in one nano second in vacuum
  - clock period is 0.25 nS, so travels 7.5 cm in one clock period in vacuum
  - far less on chip
    - communication costs time and energy
    - communication delays force locality (computations that finish in one clock period must be localized)

CPU chips are a miracle of engineering

One of the most astounding of human creations

- Thumb-nail sized
- Yet, companies with 100K engineers designs these thumbnails
- CPU chips contain several billion transistors
- More per humans on the planet

There is a lot of parallelism inside chips

Parallelism

- The success of computing rests on the amount of parallelism we can hide (tuck inside) chips
- Batches of instructions are picked up and executed
- Memory subsystems, networks, etc are all engaged in fervent activity in parallel

Caches are hugely important

Locality!

- Access to registers and TLB : “instantaneous”
- L1 cache : almost CPU cycles
- Other caches : many cycles
- Main memory : huge latency
- Much of a CPU today is cache

Chip photos from slide deck Lec1

See EnergyNomenclature.pdf

Compilers are Crucial to Efficiency

Programs mapped to machine code

- User programs must express intent
- Over-expressing (e.g. for loops with fixed order) “confuse” compilers
- Language constructs such as “forall” are
  - good for the user (clearer intent)
  - good for the compiler (more parallel code)

Operating Systems are Crucially Important too

Role of OS

- Provide mechanisms to create processes and threads
- Allocate memory spaces for processes, threads
  - Stack sizes
  - Handle exceptions thrown
  - Allocate, deallocate memory
  - Handle scheduling decisions

Processes, Threads, Tasks

Some basics

- Processes : isolated memory
- Threads : shared memory
  - Multiple within a process
- Tasks : work capability or simply "work"
  - Bind to threads when available
- One way to ensure load balancing: *work stealing*
  - Pioneered in Cilk
  - Now standard in all advanced runtimes
    - TBB supports it
    - OMP (OpenMP) will soon

Hardware Hierarchy

The hierarchy of hardware

- Motherboard contain sockets (cavities)
- Sockets house CPU multicore chips
- Multicore chips house cores
- Each core may support more than one hardware thread (SMT)
- Software threads bind to hardware threads

See Jernej Babic's slides mentioned in Reading1.pdf

Simplifying things a bit

For now, we discuss all the things happening within one process

- That is, a bunch of shared memory threads
- These threads ideally run on separate HW threads (or cores)
  - Non-ideal to run more than one software thread on one core, although you can time-multiplex and do so

Will adding more "processors" (parallel computing capability) always speed things up?

Two ideas exist in this space:

- Strong scaling: Yes, as we add processors, we get faster
- Weak scaling: No, but if we also grow the problem size proportional to the # processors (*the same problem-size per processor*) we will solve bigger problems over the same amount of time
- When can we expect such scaling behaviors?
  - Strong: Achieve this only if the serial part is small in overhead (so with serialization bottlenecks (e.g., bad locks), we won't get this)
    - This is known as Amdahl's law
  - Weak: Achieve this only if the communication among the processors grows slowly
    - This is known as Gustafson-Barsis's law

Amdahl's Law

- Two helpful terms:
  - Speedup: $s = \frac{T_{serial}}{T_{parallel}}$
  - Parallel efficiency: s/p ($= 1$ for $s = p$)
- Let a program have a fraction $0 \leq k \leq 1$ be non-parallelizable
- If the program takes runtime T,
  - it takes $T \cdot k$ for the serial part
  - and $T \cdot (1 - k)$ for the parallel part
- If we put P processors to "grind away" the parallel part,
  - the total time becomes $T \cdot k + \frac{T \cdot (1-k)}{P}$
  - the speedup is now $\frac{T}{T \cdot (k + \frac{(1-k)}{P})}$
  - which simplifies to $\frac{P}{1 + k \cdot (P-1)}$
  - Now, $\lim_{P \to \infty} \dfrac{P}{1 + k \cdot (P-1)} = \dfrac{1}{k}$
  - Thus, with $k = 0.1$, we get only $10\times$ speedup

Depiction of Amdahl's Law from Reinders' Book

Depiction of Gustafson-Barsis Law from Reinders' Book

Depiction of Data Parallelism (Reinders)

Depiction of Task Parallelism (Reinders)

Depiction of Pipelining Parallelism (Reinders)

Depiction of Mixed Solutions (Reinders)

Depiction of Hybrid Pipelining / Data Parallel (Reinders)

Helps eliminate pipelining bottlenecks

The Importance of Parallel Programming Patterns

Much like the advent of structured parallel programming