Some other topics

Anthony Lee

December 2024

Outline

Outline

Introduction

- A few topics today:
  - Pseudorandom number generators
  - Sparse matrices
  - Parallel computing
  - Random algorithms

Outline

Pseudorandom number generation (PRNG)

- Computers typically use PRNGs in algorithms.
- They can produce new "seemingly" random numbers much faster than external "truly" random sources.
- Old PRNGs are usually too simple.
  - E.g., the linear congruential generators described in the notes.
- Things that matter: the period (samples before repeating) and the quality.
- There are software packages that "test" whether random numbers pass tests.
  - Can be thought of as hypothesis tests with H_0 being that the numbers are i.i.d.

PRNGs: recent activity

- Until recently, the Mersenne Twister algorithm was very popular and has a huge period of $2^{19937} - 1$.
- There has recently been a flurry of activity.
  - Counter-based PRNGs (cryptographic-inspired).
  - PCG Family: permuted congruential generators.
  - xoshiro / xoroshiro.
  - Lots more!
- Different PRNGs have different apparent strengths, including speed.
- For parallel computing, useful to have skip-ahead functionality, or allocate each thread a different block of a large orbit.

Outline

Sparse matrix computations

- Many statistical computations involve matrices which contain very high proportions of zeroes: these are sparse matrices.
- Numerical linear algebra: most computations are additions, subtractions and multiplications of pairs of numbers.
- There is no point doing the computations involving 0: answer is known in advance.
- There is also no point storing the whole matrix; just store the non-zero values and their locations.
- Example in notes: a design matrix combining two categorical variables. Most entries are 0.
- Special routines for manipulating sparse matrices, e.g. decompositions that preserve sparsity.

An example

- Consider a simple problem that helped create a big company.

Figure:
http://web.archive.org/web/19981202230410/http://www.google.com/

Ranking webpages

- Ranking based on the average occupation time of each webpage for a random web surfer...
  - with probability α follows links uniformly at random on the page they are on?
    - on a page with no links, sample a page from a fixed distribution.
  - with probability $1 - \alpha$ samples a page from a given (possibly personal) distribution.

The internet (some of it, anyway)

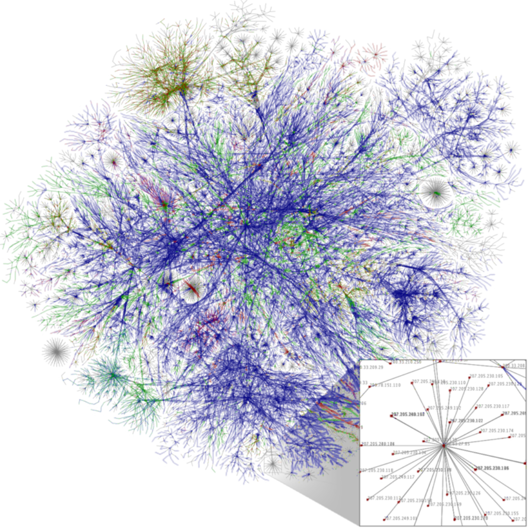

Figure: Source: Wikimedia commons

A stochastic matrix

- The relevant data is the $n \times n$ (directed) adjacency matrix A.
- A is a matrix of all zeros, except $A_{ij} = 1$ if there is a link from page i to page j.
- No self-links and A is a very sparse matrix.
- From this we create a substochastic matrix H via

$$H_{ij} = \frac{A_{ij}}{\sum_{k=1}^{n} A_{ik}},$$

  if the denominator is positive and $H_{ij} = 0$ otherwise.
- A stochastic matrix S is then defined via

$$S = H + dw^T,$$

  where d is a binary vector with $d_i = 1$ if and only if $\sum_{k=1}^{n} A_{ik} = 0$, and w is some simple distribution.

Random surfer stochastic matrix

- S is the transition matrix of the random surfer who just chooses links uniformly at random.
- Now we add the possibility of choosing from a "personalization" distribution. Set

$$G = \alpha S + (1 - \alpha) 1 p^T,$$

where p is the personalization distribution, which we assume satisfies $p_i > 0$ for all i.
- A and hence H are sparse. But S and G are dense!
- To compute average occupation time, use Perron–Frobenius for finite Markov chains:

$$\mu^T G^m \to \pi^T,$$

where μ is an arbitrary probability distribution and π is the (unique) stationary distribution given our assumptions.
- So we want to compute $\mu^T G^m$ for large m.

Sparse power iteration

- How can we do this without ever constructing G? Use

$$\begin{aligned}\nu^T G &= \mu^T \left[\alpha S + (1-\alpha)1p^T\right] \\ &= \nu^T \left[\alpha H + \alpha d w^T + (1-\alpha)1p^T\right] \\ &= \alpha \nu^T H + \alpha \left(\nu^T d\right) w^T + (1-\alpha)p^T,\end{aligned}$$

  to compute $\mu_k = \mu_{k-1}^T G$ for $k = 1, \dots, m$.
- The only matrix is the sparse matrix H, and the complexity is $\mathcal{O}(\#\text{links} + n)$.
- Example in lecture notes
- G would be 5 terabytes.
- It only takes seconds to compute π using above.

Outline

Why parallel computing?

- Traditionally, computing was largely serial.
- Algorithms designed to be run on a single machine in one thread on one core on one processor.
- Operating systems allow several processes to run simultaneously on one core.
- Think of your old personal computer: it looks like everything is running simultaneously!
- Since 2000s, shift to more cores on processors.
  - Physical limitations to making cores more powerful.
- GPUs are an extreme version of this.
- If time is the issue, need to compute in parallel!

How do algorithms run on a computer?

- A processor may have several cores.
- A core can execute instructions and access various forms of memory, usually arranged in a hierarchy.
  - Fast to slow: registers, caches, main memory, disk.
- A thread (of execution) is a sequence of instructions to be run on a core.
- Multiple threads can be part of the same process, and can thereby share resources with each other.
- Distinct processes do not share resources (at least directly).

Types of parallelism

- Running several processes in parallel (on one machine or several).
  - May have machines connected via a network, with messages passed between them.
- Running several threads in parallel within a process (on one machine).
- Coarse parallel computing: lots of independent computations to do.
  - Just run them all in parallel in different processes. Speedup is simple.
  - If processes require lots of memory, may need to look at threads anyway.

Lightweight parallelism

- Say you need to compute several numbers in a for loop, all of which are required for a subsequent step.
- Then several threads can be used to each do some of the computations.
- They share memory and are relatively lightweight to create/destory in comparison to processes.
- For GPUs, there are additional requirements for efficiency:
  - More ALUs/FPUs, less flow control.
  - Need blocks of computation to be identical, down to the instructions.
  - Also need memory to be laid out nicely for the computation.
- In practice, people use frameworks to assist with / avoid GPU programming.

CPU vs GPU: transistor allocation

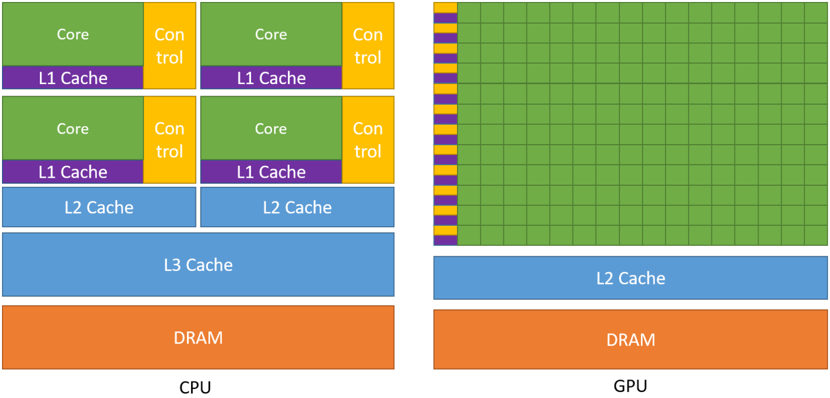

Figure: From https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Memory management

- High-level languages, e.g. R, Python.
  - Often we do "weird" things to enable fast computation, e.g. vectorizing computations in a way that necessitates the construction and destruction of arrays.
  - This is because we want to use "big" operations that have been compiled into machine code.
- Instructions, from low-level languages like C and even Julia.
  - For loops are ideal.
  - Allocation of memory, e.g. for arrays is very slow.
- This is important in serial computation, but more problematic in parallel.
- Try optimizing code in Julia or C and see the difference!

An idea of memory costs

- Rough speed of different types of memory access:
  - L1 cache (around 64KB) reference: 0.5ns.
  - L2 cache (around 256KB) reference: 7ns.
  - Main memory (around 4–8GB) reference: 100ns.
  - Disk seek: $10\text{ms} = 10^7\text{ns}$.
  - Solid State Drive: $0.1\text{ms} = 10^5$ ns.

An idea of memory costs

- Rough speed of different types of memory access:
  - L1 cache (around 64KB) reference: 0.5ns.
  - L2 cache (around 256KB) reference: 7ns.
  - Main memory (around 4–8GB) reference: 100ns.
  - Disk seek: 10ms = 10^7ns.
  - Solid State Drive: 0.1ms = 10^5 ns.
- By analogy:
  - L1 cache: reaching for something on our desk (e.g. 1 second)
  - L2 cache: fetching it instead from a drawer (14 seconds)
  - Main memory: going up a set of stairs into another room to fetch something (3 minutes, 20 seconds).
  - Disk: walking from the University of Warwick to Cape Town (South Africa) and back (5555 hours or 231 days).
  - Solid state drive: walk to Brighton (55.55 hours or 2.31 days).

Complexity model

- Simple model we often use to talk about algorithmic complexity:
  - e.g. every operation (arithmetic, memory access, etc.) takes 1 unit of time.
- Obviously not always accurate for certain types of computation.
- For multi-threaded computation, some speedup is lost to overheads, synchronization, etc.
- In some cases, different algorithms are appropriate for parallel computation.

Outline

Random algorithms

- These are algorithms that output (realizations of) random variables.
- Not to be confused with randomized algorithms that are "proper" algorithms:
  - For a given input there is a specific output, like a function.
  - Randomized may utilize randomness as a tool, e.g. randomized quicksort.
- Technically, a random algorithm is just an algorithm with additional random input.
- Lots of interesting questions about complexity of random vs deterministic algorithms.

Checking matrix multiplication

- Say we have three $n \times n$ matrices A, B and C.
- We want to know if $A \times B = C$.
- Basic deterministic algorithm: compute $A \times B$ in $O(n^{\alpha})$ time and check.
  - Best algorithm so far has $\alpha = 2.3727$.
- We will look at a way to check $A \times B = C$ such that
  - If $A \times B = C$, it always return "yes".
  - If $A \times B \neq C$, it returns "no" with probability at least $\frac{1}{2}$, and "yes" otherwise.

Freivalds' algorithm

- If $AB = C$, then

$$ABx = A(Bx) = Cx.$$

- So to check that $AB = C$ we will generate a uniformly random binary vector $\xi \in \{0, 1\}^n$.
- Then we compute $A(B\xi)$ and $C\xi$ and return "yes" if all elements are equal and "no" otherwise.
- This takes $O(n^2)$ time.
- If $AB = C$ then clearly we will always answer "yes".

When $AB \neq C$

- We check $AB\xi = C\xi$ for $\xi \sim \mathrm{Uniform}(\{0,1\}^n)$.
- This is like computing $r = (AB - C)\xi$ and checking if $r = 0$.
- If $AB \neq C$ then $D = (AB - C)$ has a non-zero element.
- Let d_{ij} be a nonzero element of D. We will look at

$$r_i = \sum_{k=1}^{n} d_{ik}\xi_k = d_{ij}\xi_j + \sum_{k=1,k\neq j}^{n} d_{ik}\xi_k = d_{ij}\xi_j + Y.$$

- Now, $\mathbb{P}(R_i = 0)$ is equal to

$$\mathbb{P}(R_i = 0|Y = 0)\,\mathbb{P}(Y = 0) + \mathbb{P}(R_i = 0|Y \neq 0)\,\mathbb{P}(Y \neq 0).$$

- But $\mathbb{P}(R_i = 0|Y = 0) = \mathbb{P}(\xi_i = 0) = \frac{1}{2}$ and $\mathbb{P}(R_i = 0|Y \neq 0) \leq \mathbb{P}(\xi_i = 1) = \frac{1}{2}$.
- So

$$\Pr(R_i = 0) \leq \frac{1}{2}\left[\Pr(Y = 0) + \Pr(Y \neq 0)\right] = \frac{1}{2}.$$

When $AB \neq C$

- When $AB \neq C$, we will return "no" with probability at least $\frac{1}{2}$.
- Therefore, we can repeat the procedure k times.
- The probability that we do not observe a "no" but $AB \neq C$ is less than 2^{-k}.
- If we do observe a "no" we can output "no" and we are always right.

Outline

Wrapping up

- There are lots of topics we have note covered.
- Unfortunately, many of these will be important to you!
- Hopefully some coverage of fundamental ideas.
- For research, we have to learn what is required to make progress.
- Feedback welcome!