Parallel processor architecture & algorithms

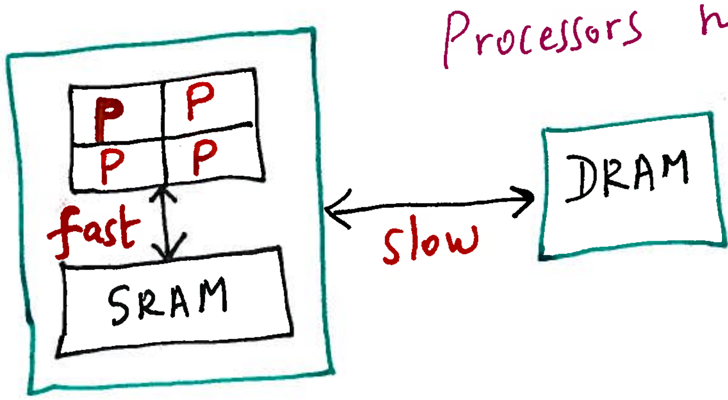
Intel 8086 (1981) : 5 MHz used in first IBM PC
80486 (1989) : 25 MHz
→ i486 because of a court ruling that prohibited the trademarking of numbers

Pentium (1993) : 66 MHz
Pentium 4 (2000) : 1.5 GHz deep ≈ 30-stage pipeline

Pentium D (2005) : 3.2 GHz
and then clock speed stopped increasing!

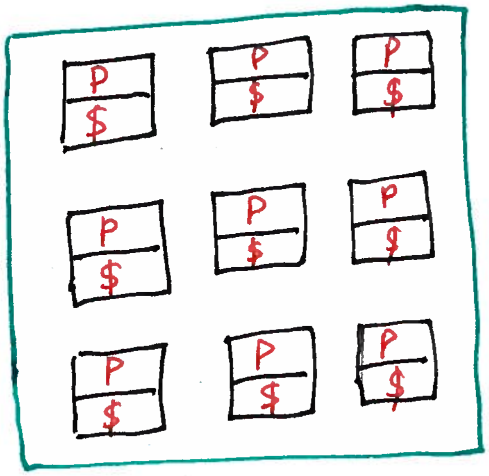
$\underbrace{\text{Quad core}}$ Xeon (2008) : 3 GHz
Key to performance scaling : increase the number of cores on chip.

Processors need data to compute on.

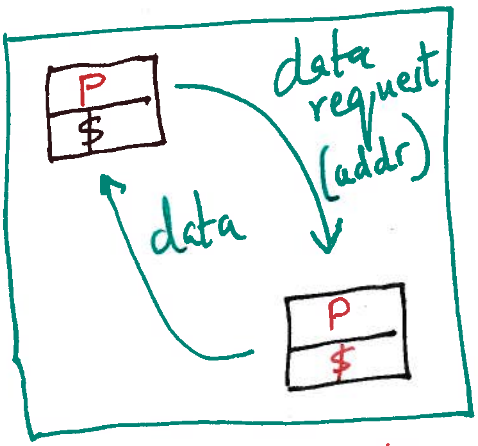


Problem: SRAM cannot support more than ~4 memory requests in parallel.

$: cache   P: processor



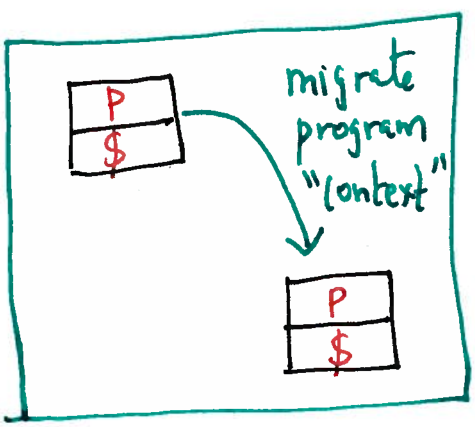
Most of the time program running on the processor accesses local memory or "cache" memory.



Every once in awhile, it accesses remote memory

Round-trip required

Research Idea: Execution Migration



When program running on a processor needs to access cache memory of another processor, it migrates its "context" to the remote processor & executes there.

One-way trip for data access

Context = $\underbrace{\text{Program Counter + Register File + ..}}_{\text{few kbits}}$

(can be larger than data to be accessed)

Assume we know or can predict the access pattern of a program

$m_1, m_2, \ldots m_N$ memory addresses

$p(m_1), p(m_2), \ldots p(m_N)$ processor caches for each m_i

Example: $p_1\ p_2\ p_2\ p_1\ p_1\ p_3\ p_2$

$\text{cost}_{mig}(s,d) = \text{distance}(s,d) + L$ ← load latency is a function of context size.

$\text{cost}_{access}(s,d) = 2 * \text{distance}(s,d)$

If $s == d$, costs are defined to be 0.

Problem: Decide when to migrate to minimize total memory cost of trace.

Example:
Start at p1

local accesses

$p_1\ p_2\ p_2\ p_1\ p_1\ p_3\ p_2$

migrate to p2 | migrate to p1 | remote access | remote access

costs

What can we use to solve this problem?

Dynamic Programming!

Program at p_1 initially, number of processors = Q

Subproblems?

$DP(k, p_i)$ = cost of optimal solution for the prefix $m_1 \ldots m_k$ of memory accesses when program starts at p_1 and ends up at p_i

$$DP(k+1, p_j) = \begin{cases} DP(k, p_j) + cost_{access}(p_j, p(m_{k+1})) & \text{if } p_j \neq p(m_{k+1}) \\ \min_{i=1}^{Q} \left(DP(k, p_i) + cost_{mig}(p_i, p_j) \right) & \text{if } p_j = p(m_{k+1}) \end{cases}$$

Complexity?

$$O(\underbrace{N \cdot Q}_{\text{number of subproblems}} \cdot Q)$$

Q ← cost per subproblem

$$= O(NQ^2)$$

My research group is building a 128-processor Execution Migration Machine that uses a migration predictor based on this analysis.

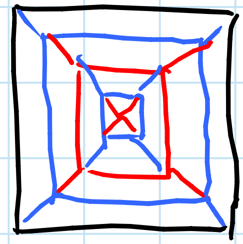
6.006	Lecture 24	Dec. 13, 2011

Erik's main research areas:

- computational geometry [6.850]
  - geometric folding algorithms [6.849]
  - self-assembly
- data structures [6.851]
- graph algorithms [6.889]
- recreational algorithms [SP.268]
- algorithmic sculpture

Geometric folding algorithms: [6.849, videos online]

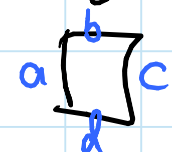
- design: algorithms to fold any polyhedral surface from a square of paper
  [Demaine, Demaine, Mitchell 2000; Demaine & Tachi 2011]
  - bicolor paper ⇒ can 2-color faces
  - OPEN: how to best optimize "scale factor"
  - e.g. best $n \times n$ checkerboard folding recently improved from $\sim n/2 \to \sim n/4$
- foldability: given a crease pattern, can you fold it flat?
  - NP-complete in general [Bern & Hayes 1996]
  - OPEN: $m \times n$ map with creases specified as mountain/valley [Edmonds 1997]
  - just solved: $2 \times n$ [Demaine, Liu, Morgan 2011]



- hyperbolic paraboloid [Bauhaus 1929] doesn't exist!
  [Demaine, Demaine, Hart, Price, Tachi 2009]
- understanding circular creases

- any straight-line graph can be made by folding flat & one straight cut
  [Demaine, Demaine, Lubiw 1998;
  Bern, Demaine, Eppstein, Hayes 1999]

Self-assembly: geometric model of computation
- glue: e.g. DNA strands, each pair has strength
- square tiles with glue on each side
- Brownian motion: tiles/constructions stick together if Σ glue strengths $\geq$ temperature

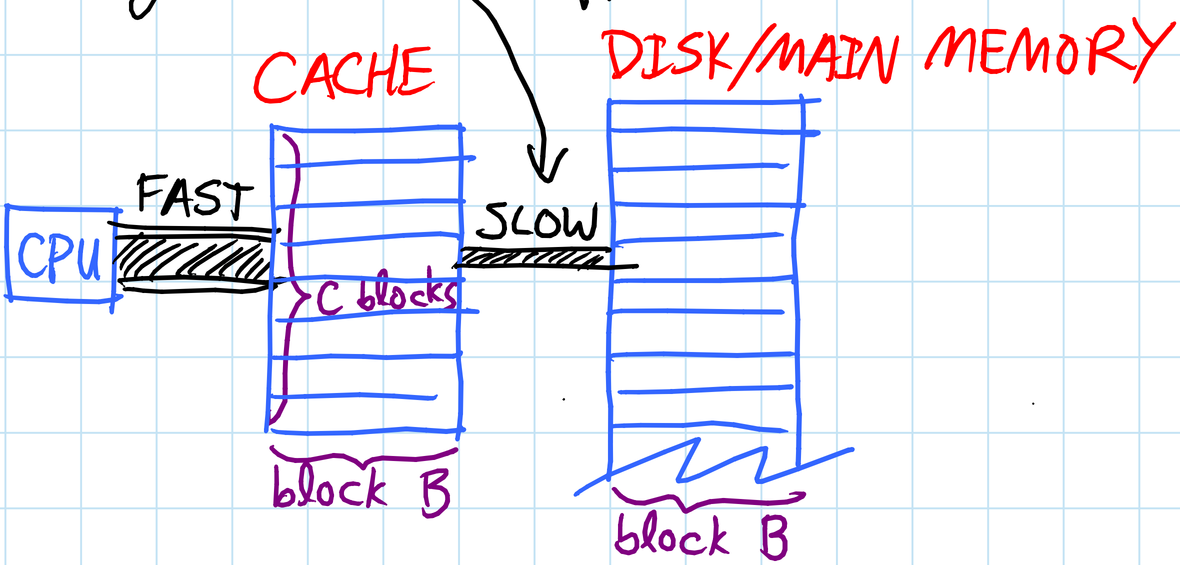


- can build $n \times n$ square using $O\left(\frac{\lg n}{\lg \lg n}\right)$ tiles
  [Rothemund & Winfree 2000]
  or using $O(1)$ tiles & $O(\lg n)$ "stages"
  algorithmic steps by the bioengineer
  [Demaine, Demaine, Fekete, Ishaque, Rafalin, Schweller, Souvaine 2007]
- can replicate ∞ copies of given unknown shape using $O(1)$ tiles & $O(1)$ stages
  [Abel, Benbernou, Damian, Demaine, Demaine, Flatland, Kominers, Schweller 2010]

Data structures: [6.851, videos next semester]

- <u>integer</u> data structures: store n integers in $\{0, 1, \ldots, u-1\}$ subject to insert, delete, predecessor, successor (on word RAM)
  - hashing does exact search in $O(1)$
  - AVL trees do all in $O(\lg n)$
  - $O(\lg \lg u)$/op. [van Emde Boas]
  - $O(\lg n / \lg \lg u)$/op. [fusion trees: Fredman & Willard]
  - $O(\sqrt{\lg n / \lg \lg n})$/op. [min of above]

- cache-efficient data structures:
  - <u>memory transfers</u> happen in blocks

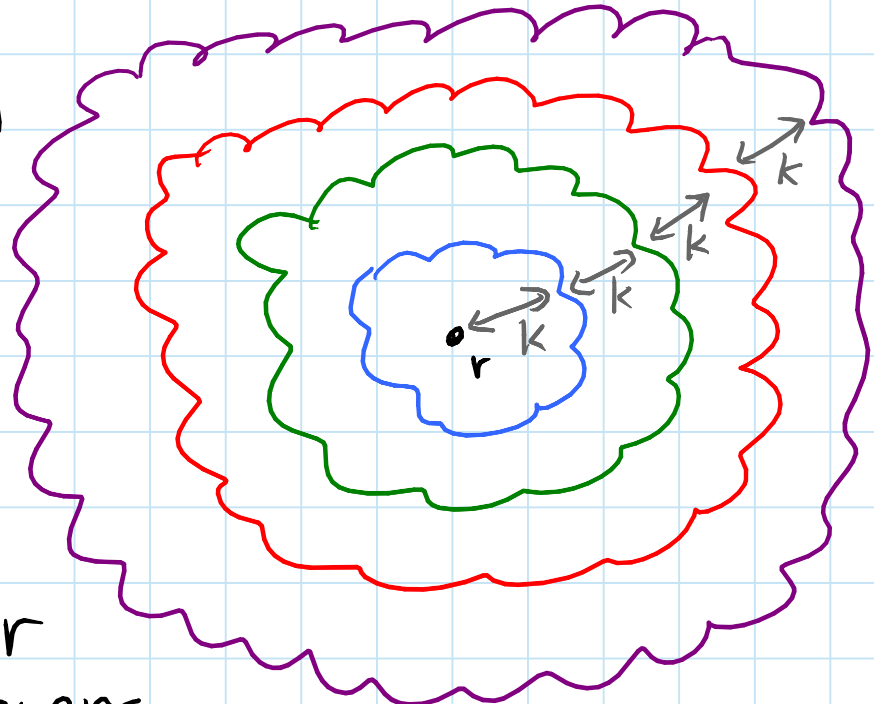


  - searching takes $\Theta(\log_B N)$ transfers (vs. $\lg n$)
  - sorting takes $\Theta(\frac{N}{B} \log_C \frac{N}{B})$ transfers

  - possible even if you don't know B & C!

(Almost) planar graphs: [6.889, videos online]

- Dijkstra in $O(n)$ time
  [Henzinger, Klein, Rao, Subramanian 1997]
- Bellman-Ford in $O(n \lg^2 n / \lg\lg n)$ time
  [Mozes & Wolff-Nilsen 2010]



- many problems NP-hard, even on planar graphs
- but can find a solution within $1+\varepsilon$ factor of optimal, for any ε
- run BFS from any root vertex r
- delete every k layers
- for many problems, solution messed up by only $1 + 1/k$ factor ($\Rightarrow k = 1/\varepsilon$)
- connected components of remaining graph have $<k$ layers ~ can solve via DP typically in $\sim 2^k \cdot n$ time

[Baker 1994 & others]

Recreational algorithms:

- many algorithms & complexities of games
  [some in SP.268 & our book
  Games, Puzzles, & Computation (2009)]
- $n \times n \times n$ Rubik's Cube diameter is $\Theta(n^2/\lg n)$
  [Demaine, Demaine, Eisenstat, Lubiw, Winslow 2011]
- Tetris is NP-complete
  [Breukelaar, Demaine, Hohenberger, Hoogeboom, Kosters, Liben-Nowell 2004]
- balloon twisting any polyhedron
  [Demaine, Demaine, Hart 2008]
- algorithmic magic tricks

Algorithms classes at MIT: (post-6.006)

- #1: 6.046: Intermediate Algorithms
  (more adv. algorithms & analysis, less coding)
- 6.047: Computational Biology
  (genomes, phylogeny, etc.)
- 6.854: Advanced Algorithms
  (intense survey of whole field)
- 6.850: Geometric Computing
  (working with points, lines, polygons, meshes, ...)
- 6.849: Geometric Folding Algorithms
  (origami, robot arms, protein folding, ...)
- 6.851: Advanced Data Structures
  (sublogarithmic performance)
- 6.852: Distributed Algorithms
  (reaching consensus in a network with faults)
- 6.853: Algorithmic Game Theory
  (Nash equilibria, auction mechanism design, ...)
- 6.855: Network Optimization
  (optimization in graph: beyond shortest paths)
- 6.856: Randomized Algorithms
  (how randomness makes algs. simpler & faster)
- 6.857: Network and Computer Security
  (cryptography)

Other theory classes:

- 6.045: Automata, Computability, & Complexity
- 6.840: Theory of Computing
- 6.841: Advanced Complexity Theory
- 6.842: Randomness & Computation
- 6.845: Quantum Complexity theory
- 6.440: Essential Coding Theory
- 6.441: Information Theory

Top 10 Uses of 6.006 Cushions

10. Sit on it: guaranteed inspiration in constant time
    (bring it to the final exam)
9. Frisbee (after cutting it into a circle)*
8. Sell as a limited-edition collectible on eBay
   (they'll probably never be made again—at least $5)
7. Put two back-to-back to remove branding*
   (so no one will ever know you took this class)
6. Holiday conversation starter… and stopper
   (we don't recommend re-gifting)
5. Asymptotically optimal acoustic paneling
   (for practicing piano & guitar fingering DP)
4. Target practice for your next LARP*
   (Live Action Role Playing)
3. Ten years from now, it might be all you'll remember about 6.006
   (maybe also this top ten list)
2. Final exam cheat sheet*
1. *Three words:* OkCupid profile picture