Memory Managing Algorithms on Distributed Systems

Katie Becker and David Rodgers
External Memory Algorithms using a Coarse Grained Paradigm

• Written by Jens Gustedt, March 2004
• Main idea: Present a framework that allows for algorithms in external memory settings that were originally designed for coarse grained architectures
  – External Memory Settings
    • External Storage, i.e. Large Disk Array
    • To only access parts of the data at any one time during the execution of the algorithm.
  – Coarse Grained Architecture – Moving lots of data at one time
Framework and Simulations

• Use the Parallel Resource Optimal Computation (PRO) model to transform a serial algorithm into a parallel algorithm for a coarse grain system
  – Trades restriction on the internal versus external memory size for an independence of latency of the hardware. Therefore, performance is bound to only computing time and bandwidth.

• Then used Soft Synchronized Computing in Round for Adequate Parallelization (SSCRAP) for simulation of PRO algorithms in an external memory setting.
PRO

• Method of defining an optimal parallel algorithm relative to a sequential algorithm.
• A PRO-algorithm is required to be both time- and space-optimal.
• A parallel algorithm is said to be time- (or work-) optimal if the overall computation and communication cost involved in the algorithm is proportional to the time complexity of the sequential algorithm used as a reference.
• Similarly, it is said to be space-optimal if the overall memory space used by the algorithm is of the same order as the memory usage of the underlying sequential version.
PRO Submodels

• **Architecture** – allows for system composed of $p$ distributed processors, each of which has memory size $M(n) = O((S_A(n)/p)$

• **Execution** – simulate the execution of a program by doing as much computation as necessary between messages, then send no more than 1 message. (Superstep)

• **Cost** – sum of all running times $= T(n)$
SSCRAP

- Scalable simulator used to mimic many processors on a single processor
- Is used for benchmarking algorithms
- It provides a high abstraction level, making the real evolved communications transparent for the user and efficiently handles data exchanges and inter-process synchronizations
- Interfaced with two parallel architectures: distributed memory (cluster of PCs) and shared memory
Experiments for running parallel PRO algorithms with SSCRAP

• Have had successful runs on different platforms
  – PC
  – Multiprocessor workstations (SUN)
  – Mainframe with 56 processors (SGI)

• For the following examples:
  – CPU Pentium 4 2x2.0 GHz
  – RAM 1 GB
  – Bus speed 99 MHz
  – Disk swap 2 GB
    available file system 20 GB (software raid)
    bandwidth read/write 55/70 MB/sec
  – OS GNU/linux
1st Test - Sorting

- In place quicksort was used as a subroutine for the sorting routine
- Performed on a vector of doubles
- Results
  - File mapping takes much more time than running the program entirely in RAM. On the other hand, corresponding running times were reliable beyond the swap boundary
  - Factor in bandwidth of 20 between RAM and disk access is maintained, meaning the out-of-core computation is not slower than 20 times the in-core computation
Sorting
2nd Test Algorithm - Random Permutation Generation

• Problem with linear time complexity
• Most costly operation is random memory access
  – Tends to have many cache misses
• Computation time is also quite high, since random (pseudo) numbers need to be generated
Random Permutation Generation
Results

• Coarse grained parallel models like PRO and their simulations using the SSCRAP library enable us to visualize the use of parallel programs to map memory to disk files.
• The principle bound in problem size is related to the availability of a resource that is extensible and cheap (disk space).
• Main bottleneck for computation time as a whole is the bandwidth of the external storage device.
Cache-Oblivious Algorithms

• Written by Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran

• Main idea: Guarantee that data is loaded exactly once and removed at most once
Matrix Multiplication Memory View

16 x 16 Matrix

Cache Map

Cache Map

Cache Map
Memory Problem With Matrix Multiplication

• Problem: Memory must be loaded and unloaded repeatedly to complete the matrix multiplication
• Proposed Solution: Find a method to guarantee the loading and unloading happens at most once
• First Method: Patches (done in class)
• Problem: Dependent that there is a consistent amount of cache available; thus, not cache-oblivious
• New Solution: Divide the Problem
Matrix Transposition

• Definition: Converting an n X m matrix A into an m X n matrix B where element $A_{i,j}$ is equal to element $B_{j,i}$

• Naïve approach takes $O(mn)$ time and cache misses (doubly nested loops)

• Divide and conquer algorithm takes $O(mn)$ time with $O(1+mn/L)$ cache misses where $L$ is the cache line length

• Having a cache-oblivious algorithm for matrix transposition allows cache-oblivious fast Fourier transform
Transposition Memory View

16 x 16 Matrix

Cache Map

16 x 16 Matrix

Cache Map

16 x 16 Matrix

Cache Map
Divide and Conquer Memory View

16 x 16 Matrix  8 x 8 Matrix  8 x 4 Matrix

Overflow  Overflow  Perfect Fit
Funnelsort

• Cache-oblivious sorting algorithm
• $O(1+(n/L)(1+\log_Z n))$ cache misses
• Running time $O(n \log n)$ 😊
• Harder to implement than quicksort, but better on account of cache misses
Funnelsort Diagram

• Divide the input into $n^{1/3}$ contiguous blocks each of size $n^{2/3}$, then sort blocks recursively
• Combine the $n^{1/3}$ sorted blocks using an $n^{1/3}$ merger
• Merging done by accepting $k$ already-sorted sequences and merging recursively
• Only merge portions which fit into cache simultaneously
Distribution Sort

- Cache-oblivious
- $O(1+(n/L)(1+\log Z n))$ cache misses
- $O(n \log n)$ running time
- Related to bucket sort
- Partition array into $vn$ contiguous array of size $vn$ where $n$ is the number of elements in the array. Recursively sort each array
- Distribute the sorted subarrays into $q$ buckets $B_1, \ldots, B_q$ of size $n_1, \ldots, n_q$ respectively such that:
  - $\max\{x \mid x \in B_{i+1}\}$ for $i = 1, 2, \ldots, q-1$.
  - $n_i = 2vn$ for $i = 1, 2, \ldots, q$
- Recursively sort each bucket
- Copy the sorted buckets back to the original array
Assumptions Made in Model

- Memory management is optimal
- Exactly two levels of memory
- Automatic replacement within memory
- Fully associative memory and cache
- Need to demonstrate that the ideal-cache model is accurately simulated by stricter models
Optimal Memory Management

• The time used in an LRU algorithm is at most twice the number of cache misses as the ideal algorithm (latest next used)

• Therefore, while memory management is not optimal, it is sufficiently close that the assumption is not unreasonable
Memory Hierarchy Model
Operating System Memory Management

- Two assumptions handled by modern operating systems
  - Automatic Memory Replacement
  - Fully Associative Cache
Conclusions

• Two different methods of accelerating processing by accessing memory less frequently
  – Transferring large quantities at once (coarse grained memory management)
  – Always transferring quantities small enough to fit into cache (divide and conquer/cache oblivious)