Memory Management Algorithms on Distributed Systems

Katie Becker and David Rodgers
CS425
April 15, 2005
Table of Contents

1. Introduction
2. Coarse Grained Memory
   2.1. Bottlenecks
   2.2. Simulations
   2.3. SSCRAP
   2.4. Introduction to Parallel Resource Optimal Computation
   2.5. Simulating
      2.5.1. Simultaneous Execution On a Single Processor
      2.5.2. Communication Between Processors
      2.5.3. Analysis of Simulation
   2.6. Sample Simulated Algorithms
      2.6.1. Test Problems
      2.6.2. Sorting
      2.6.3. Randomized Permutations
      2.6.4. List Ranking
      2.6.5. Results
3. Cache-Oblivious Algorithms
   3.1. Introduction to Cache
   3.2. Memory Models
   3.3. Problem Sets
   3.4. Sample Cache-Aware Algorithm: Review of Blocked
   3.5. Matrix Multiplication
   3.6. Sample Cache-Oblivious Algorithms
      3.6.1. Matrix Multiplication
      3.6.2. Matrix Transposition Using Rec-Transpose
      3.6.3. Fast Fourier Transform
      3.6.4. Funnelsort
      3.6.5. Distribution Sort
   3.7. Theoretical Justifications for Ideal Model
4. Conclusion
5. References
1. Introduction

This paper analyzes two papers on external memory management; one on coarse grained memory, the other on cache-oblivious memory. Each decrease the requisite overall time using a radically different strategy; coarse grained memory decreases the amount of time required to transfer data between machines by transferring large blocks at one time to avoid latency effects while the cache-oblivious strategies decrease the amount of time required to transfer data within a single machine by guaranteeing that the sum of the blocks transferred are smaller than the amount of space available in the cache.

2. Coarse Grained Memory

Very large problem sets require very large quantities of memory; frequently, more memory than can be present on a single system. In order to gain access to the data, the data is spread out amongst various systems, which is then retrieved as it is needed. The system is usually designed to transfer either large or small quantities of memory, instead of variable amounts, at a time. The first paper analyzes which option is optimal between systems using both proofs and simulations.

2.1. Bottlenecks

Bandwidth, latency, and computation speed are the primary bottlenecks associated with coarse grained memory management. Bandwidth is the amount of data that can be transferred (sent or received) simultaneously; latency is the delay between when data is sent and when it is received. The transfer of coarse grained memory, or transferring large quantities of memory at once by using a high amount of bandwidth, is one strategy of getting information where it needs to be when it is needed, because the speed of the transfer of information is always limited by the speed of light; however, transferring large quantities of information is limited only by the number of bits sent and received at the same time. This means that latency can only be improved so far, depending on the distance between systems, but bandwidth can be increased dependent only on the amount of required physical space. Additionally, the cost to increase bandwidth is orders of magnitude less than the cost to increase latency. Therefore, increasing bandwidth is cheaper, easier, and has a higher limit to implement.
However, data can not be sent before it is available, so a third bottleneck is the speed of computation. No amount of memory management can improve the bottleneck of computation speed, so this paper does not consider methods of reducing the computation time. Coarse grained memory is best applied with the limitation comes not from bandwidth but from latency and the bandwidth can be expanded. The system which is waiting on the data can then obtain a large quantity of data at once and perform the calculations on it.

2.2. Simulations

In order to determine the actual efficacy of transferring large amounts of data at a time, a simulation is set up. While most parallel systems are designed to create the illusion of a single processor from multiple processors, these simulations are designed to create the illusion of multiple processors on a single processor. Three elements can increase the processing time: Bandwidth, latency, and computation. Fortunately, latency can be treated as bandwidth overhead and memory management does not impact computation speed, meaning that only bandwidth effects need to be evaluated within the context of this paper.

2.3. SSCRAP

SSCRAP is a scalable simulator used within this paper used to simulate many processors on a single processor. The additional processors are simulated by creating one thread for each processor which then runs at a slower speed than the actual, complete system would, but with identical end results. SSCRAP is also capable of accurately simulating systems with more data than fits into the address space of a single processor by making few additions to SSCRAP for the purpose of this evaluation, which allowed it to simulate bandwidth effects; namely, adding a handle to the data to be transferred as a pointer and mapping that pointer to the actual data. SSCRAP could also be extended with an environment variable allowing the programmer to fix the number of simultaneously-executing processors. This adds the additional advantage of being able to modify the number of systems being used on a single problem, allowing the user to determine the optimal number of processors to use on a single problem instance.
2.4. Introduction to Parallel Resource Optimal Computation

Parallel Resource Optimal Computation (PRO) is a method of defining an algorithm defined using a 5-tuple, algorithmic size $P$, a problem instance of size $n$, a sequential algorithm $A$ which solves the problem in time $TA(n)$ within space $SA(n)$, each of which is relevant since any system only has a finite amount of time and space available to it. This overall model can be broken down into several submodels, as follows.

- **PRO architecture** allows the simulation of a homogeneous system composed of $p$ processors, each of which has a memory size $M(n) = O((SA(n))/p)$ and a granularity $Grain(n)$.

- **PRO execution** allows PRO to simulate the execution of a program by assuming the program does as much computation as is necessary between possible messages then sends no more than one message. The data sent at once is limited by $M(n)$, since it is not possible to send information not available in the memory. This is called a superstep. The number of supersteps has an upper bound of $\lambda(n) = o((TA(n))/(p^2))$. The duration of each superstep has an upper bound of the maximum computation time on any of the single processors.

- **PRO cost** assumes a linear speedup for ease of computation. The sum of all running times is always $T(n) = O(TA(n))$.

With this information available, different PRO algorithms solving the same problem can be compared with a function of $Grain(n)$. Since this is a memory analysis, the effect of random access memory must also be evaluated; this is simulated in the parameter $R(n)$, where $R_{i,j}(n)$ is the maximum amount of memory available to the whole system on processor $j$ during superstep $i$. This may vary depending on how much memory is available in a given superstep, so both parameters are present. $R(n)$ is the sum of $R_{i,j}(n)$ over all $i$ and $j$, allowing the simulation to evaluate random-access penalties. An algorithm is RAM-aware if $O(TA(n)) = O(T(n)) = R(n)$. 

2.5. Simulating

2.5.1. Simultaneous Execution On a Single Processor

In order to cause a single processor to behave as though it is many processors, some sacrifices must be made; namely, that the speed of computation can be slowed down considerably. This is acceptable for a simulation designed to determine the optimal bandwidth and number of processors. This simulation uses one thread to simulate each processor (so that there are an equal number of threads in the simulation as processors in real life) and a delay between when a pointer is to be copied and when it is actually copied to simulate the latency and bandwidth delays. By combining the latency and bandwidth effects, the simulator can ignore real-world latency effects and treat them only as bandwidth overhead.

2.5.2. Communication Between Processors

SSCRAP assumes that there is a maximum of one send and one receive operation performed after a given superstep. These operations do not block and are not buffered, allowing the simulation to continue its operations until the end of the superstep. In order to simulate bandwidth effects, the memory stores a pointer to a file, which then stores what the memory in the real-world system would hold. The delay in transfer acts not to slow down the simulation, but as a contributor to the bandwidth effects.

2.5.3. Analysis of Simulation

The simulation can perform on a variety of systems but for purposes of this paper, a reasonably modern desktop PC was used, comprised of 2 2.0 GHz Pentium 4 processors, 1 GB RAM, a bus speed of 99 MHz, a 20 GB hard drive (with 2 GB dedicated to swap) with a maximum read bandwidth of 55 MB/sec and a maximum write bandwidth of 70 MB/sec, running GNU/Linux kernel 2.4.18 using gcc 3.2 as its C++ compiler. Reading the whole of the RAM takes at least 18.6 seconds and exchanging the whole of the RAM onto the hard drive takes at least 33.2 seconds. The system runs only the SSCRAP simulation, preventing there from being a large variance between process times of the same algorithm on the same data. The results are from running the simulation between 7 and 10 times and taking the average of the Wall times.
2.6. Sample Simulated Algorithms

2.6.1. Test Problems
Several test problems are given to the simulator in order to determine the efficiency of the system being simulated: sorting, permutation generation at random, and list ranking.

2.6.2. Sorting
Sorting is an excellent sample problem because it has a computational non-linear lower bound but this bound does not apply to memory accesses or communication. This allows the computational overhead to be calculated in advance allowing the only variables to be memory access and communication. Quicksort is chosen since it is a standard method of sorting data, done on doubles. The maximum amount of data the system could hold was 134 million objects, due to the amount of memory available, though a 32-bit system could actually address as much as 537 million objects. As expected, the simulation ran slower than the system being simulated might, due to the use of swap space, but the simulation remained accurate even as the swap space was used entirely.

2.6.3. Randomized Permutations
Randomized permutations make for another excellent sample problem since the vast majority of the processing time is due to randomly accessing memory. This causes most accesses to be cache misses, emphasizing the transfer time between main memory and the cache. Unfortunately, since the access is done randomly, numbers must be generated randomly, which has a high computation overhead. This deemphasizes the transfer time between main memory and cache while the system determines what memory to access next. These tests were done using random permutations of long ints with linear congruent generators running on each of the simulated SSCRAP processors with the maximum data physically available to the machine running the simulation was 268 million objects.

2.6.4. List Ranking
The objective of list ranking is to calculate the distance from the beginning to the end of a collection of lists for every list in the collection. This is trivial to calculate, since it is only done by starting at the beginning of each list and counting how many items it takes to traverse until the
end, but has a highly random access pattern, since the lists are not guaranteed to be stored in any order. The maximum number of objects the system could hold was 67 million, since two longs (one for the data, one for the location of the next item in the list) were needed. This problem must access a disparate group of memory in rapid succession in order to simulate the multiple machines, causing it to be the most demanding simulated problem. The results were still accurate, indicating that the simulation is correct.

2.6.5. Results

The SSCRAP simulation was found to be accurate in the context of distributed memory systems for known problems, indicating that it may be used to accurately simulate unknown problems. The simulations determined that the main bottleneck was not computation speed, but bandwidth (particularly regarding how much disk space was available outside of the system--one of the cheapest elements to expand).

3. Cache-Oblivious Algorithms

The previous method transferred large blocks of data in order to avoid delays due to latency. The following strategy increases speed by transferring smaller quantities, avoiding cache thrashing. A cache oblivious algorithm is one that runs at the same speed regardless of hardware parameters, such as cache line length and cache size.

The design of cache oblivious algorithms is dependent on the design of an ideal cache model. This cache has only two levels which can then be used to simulate any cache which has additional levels. These additional levels can be simulated by having the second level take a variable quantity of time to move to main cache.

The cache-oblivious algorithms are designed in such a way that they use cache efficiently despite being unaware of it. Block matrix multiplication is an example of a cache aware algorithm; too large a block and the cache will cycle the same value repeatedly, too small a block and there will have to be more memory fetches.


3.1. Introduction to Cache

Computation can only be done on data present in a register which makes its way through the processor's cache. In the model used in this paper, all caches are two-level, so effectively there is only a cache/register component and a main memory component. Initially, the cache is empty and the main memory has all of the input data; this input data may already be present in the actual main memory or in a farther removed component, but for simulation purposes, it is considered to be in the main memory with a variable access time. Any time data must be moved from the main memory into the cache, a cache miss occurs, causing the overall computation time using that piece of data to take longer. When data is already present in the cache, it can be used immediately. An ideal cache is one where the required data is always located in the cache.

3.2. Memory Models

This paper considers the only thing important for memory to be the speed at which data can be acquired from memory. The ideal situation is instantaneously, but this is not feasible in reality. In order to better simulate reality, two formulae are described, one for work complexity $W(n)$ as a function of the problem size $n$, and the time it takes to perform the calculations on data using random access memory, and cache complexity $Q(n; Z, L)$, the number of cache misses as a function of the cache size $Z$, the cache line length $L$, and the problem size $n$.

3.3. Problem Sets

The cache-oblivious strategies can only be useful if they have problems on which to apply. For this paper, the problems selected to design cache-oblivious strategies for were matrix multiplication, rectangular matrix transpose, fast Fourier transform, and sorting, of which all except sorting chosen for the ease of which a cache oblivious algorithm can be designed.
3.4. **Sample Cache-Aware Algorithm: Review of Blocked Matrix Multiplication**

The blocked algorithm given in class is a standard method of performing matrix multiplication. This is a cache aware algorithm; if a block is too large, the system thrashes, moving data in and out of the cache repeatedly.

3.5. **Sample Cache-Oblivious Algorithms**

3.5.1. **Matrix Multiplication**

Some method must be created to guarantee that regardless of the size of the cache, the cache is not overflowed by the quantity of data put inside of it. The algorithm designed in this case is one that recursively divides the matrices into patches half the size of the original matrices. Once all three matrices (conventionally A, B, and C) can fit into the cache simultaneously, the multiplication is performed identically to the patched algorithm, but with the guarantee that cache misses will occur only once for any given data component.

3.5.2. **Matrix Transposition Using Rec-Transpose**

Matrix transposition is converting an $n \times m$ matrix $A$ into an $m \times n$ matrix $B$ where the element $A_{i,j}$ corresponds to the element $B_{j,i}$. Both matrices are stored in the same format; that is, if matrix $A$ is stored with contiguous rows, then matrix $B$ is stored with contiguous rows. The obvious solution takes $O(mn)$ time and cache misses using doubly-nested loops.

The algorithm presented here takes $O(mn)$ time and $O(1+mn/L)$ cache misses, each of which is optimal. This is another divide and conquer algorithm, similar to the one used for matrix multiplication. In this case, the matrices are divided in half along whichever axis of matrix $A$ is greater (dividing the rows if they are equal). This is repeated until the submatrices fit entirely into the cache, at which point the transposition is done on the submatrices. This decreases the number of cache misses, since a larger quantity of matrix-contiguous values are moved into memory than in the naive approach. By relocating the data into the cache only once, cache misses are minimized.
3.5.3. Fast Fourier Transform

In the previous section, a cache-oblivious method of performing matrix transposition was developed. Fast Fourier Transform (FFT) is dependent on having a method to perform matrix transposition, so a cache-oblivious version of FFT can now be developed. This algorithm uses the six-step version of the Cooley-Tuckey FFT algorithm. By replacing the cache-sensitive transposition algorithm with the above cache-oblivious one, the FFT algorithm also becomes cache-oblivious.

3.5.4. Funnelsort

A majority of general-purpose sorting algorithms, such as mergesort, are cache-aware. Funnelsort provides an example of a cache-oblivious sorting algorithm with $O(1+(n/L)(1+\log Z n))$ cache misses that performs in $O(n \lg n)$ time. The funnelsort algorithm is as follows:

1. Divide the input into $n^{1/3}$ contiguous blocks of size $n^{2/3}$ and sort these blocks recursively.
2. Combine the $n^{1/3}$ sorted blocks using an $n^{1/3}$ merger.

This is similar to mergesort except in how merging is done; funnelsort merges the blocks by accepting $k$ sorted sequences and recursively merging them. By sorting only quantities of data which fit into cache simultaneously, there are fewer cache misses that would occur due to thrashing.

3.5.5. Distribution Sort

Distribution sort is another sorting algorithm which may be adapted to be cache-oblivious with a cache complexity of $O(1+(n/L)(1+\log Z n))$ that performs in $O(n \lg n)$ time. This method uses a bucket-splitting method, where data of approximately the same range is moved to the same bucket and then sorted in more detail. The funnelsort algorithm is as follows:

1. Partition the array into $\sqrt{n}$ contiguous arrays of size $\sqrt{n}$ where $n$ is the number of elements in the array. Recursively sort each subarray.
2. Distribute the sorted subarrays into q buckets $B_1, ..., B_q$ of size $n_1, ..., n_q$ respectively such that:
   a. $\max \{ x \mid x \text{ is an element of } B_i \} \leq \{ x \mid x \text{ is an element of } B_{i+1} \}$ for $i = 1, 2, ..., q-1$.
   b. $n_i \leq 2\sqrt{n}$ for $i = 1, 2, ..., q$
2. Recursively sort each bucket.
3. Copy the sorted buckets back to the original array

Once again, by preventing there from being more being sorted simultaneously than can fit into the cache at once, the number of cache misses is minimized.

3.6. Theoretical Justifications for Ideal Model
The model makes four key assumptions: The memory management is optimal, that there are exactly two levels of memory, that there is automatic replacement within memory, and that there is full associativity between memory and cache. It must be shown that the ideal-cache model can be simulated by stricter models. Several proofs appear showing that all of these are reasonable assumptions.

4. Conclusion
This paper analyzed two papers on external memory management; one on coarse grained memory, the other on cache-oblivious memory. Coarse grained memory management deals with very large quantities of memory -- frequently, more memory than can be present on a single system. The simulations performed determined that the main bottleneck was not computation speed, but bandwidth, particularly regarding how much disk space was available outside of the system.

On the other hand, cache-oblivious strategies deal with a single machine. These strategies decrease the amount of time required to transfer data by guaranteeing that the sum of the blocks transferred is smaller than the amount of space available in the cache.
References
