Module 8: "Performance Issues"

Lecture 15: "Locality and Communication Optimizations"

- Artifactual comm.
- Capacity problem
- Temporal locality
- Spatial locality
- 2D to 4D conversion
- Transfer granularity
- Worse: false sharing
- Communication cost
- Contention
- Hot-spots
- Overlap
- Summary

[From Chapter 3 of Culler, Singh, Gupta]

Previous Next

Artifactual comm.

- Communication caused by artifacts of extended memory hierarchy
  - Data accesses not satisfied in the cache or local memory cause communication
  - Inherent communication is caused by data transfers determined by the program
  - Artifactual communication is caused by poor allocation of data across distributed memories, unnecessary data in a transfer, unnecessary transfers due to system-dependent transfer granularity, redundant communication of data, finite replication capacity (in cache or memory)
- Inherent communication assumes infinite capacity and perfect knowledge of what should be transferred

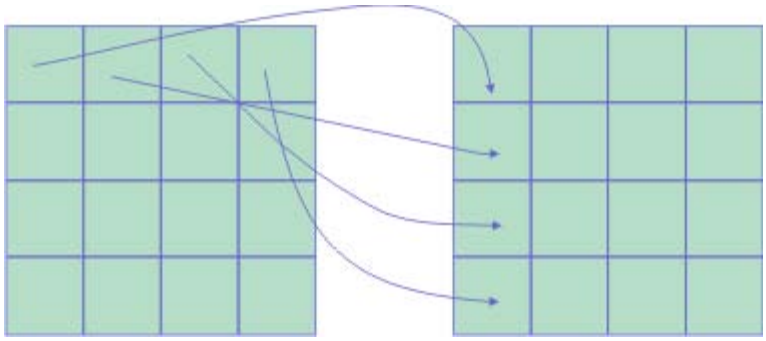
Capacity problem

- Most probable reason for artifactual communication
  - Due to finite capacity of cache, local memory or remote memory
  - May view a multiprocessor as a three-level memory hierarchy for this purpose: local cache, local memory, remote memory
  - Communication due to cold or compulsory misses and inherent communication are independent of capacity
  - Capacity and conflict misses generate communication resulting from finite capacity
  - Generated traffic may be local or remote depending on the allocation of pages
  - General technique: exploit spatial and temporal locality to use the cache properly

Temporal locality

- Maximize reuse of data
  - Schedule tasks that access same data in close succession
  - Many linear algebra kernels use blocking of matrices to improve temporal (and spatial) locality
  - Example: Transpose phase in Fast Fourier Transform (FFT); to improve locality, the algorithm carries out blocked transpose i.e. transposes a block of data at a time

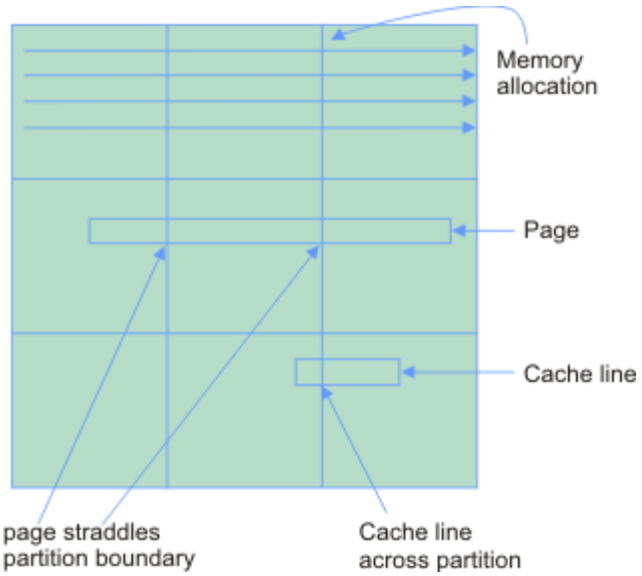
Block transpose



Previous Next

Spatial locality

- Consider a square block decomposition of grid solver and a C-like row major layout i.e. A[i][j] and A[i][j+1] have contiguous memory locations



The same page is local to a processor while remote to others; same applies to straddling cache lines. Ideally, I want to have all pages within a partition local to a single processor. Standard trick is to covert the 2D array to 4D.

2D to 4D conversion

- Essentially you need to change the way memory is allocated
  - The matrix A needs to be allocated in such a way that the elements falling within a partition are contiguous
  - The first two dimensions of the new 4D matrix are block row and column indices i.e. for the partition assigned to processor P_6 these are 1 and 2 respectively (assuming 16 processors)
  - The next two dimensions hold the data elements within that partition
  - Thus the 4D array may be declared as float B[vP][vP][N/vP][N/vP]
  - The element B[3][2][5][10] corresponds to the element in 10^{th} column, 5^{th} row of the partition of P_{14}
  - Now all elements within a partition have contiguous addresses
- Clearly, naming requires some support from hw and OS
  - Need to make sure that the accessed virtual address gets translated to the correct physical address

Transfer granularity

- How much data do you transfer in one communication?
  - For message passing it is explicit in the program
  - For shared memory this is really under the control of the cache coherence protocol: there is a fixed size for which transactions are defined (normally the block size of the outermost level of cache hierarchy)
- In shared memory you have to be careful
  - Since the minimum transfer size is a cache line you may end up transferring extra data e.g., in grid solver the elements of the left and right neighbors for a square block

decomposition (you need only one element, but must transfer the whole cache line): no good solution

Previous Next

Worse: false sharing

- If the algorithm is designed so poorly that
  - Two processors write to two different words within a cache line at the same time
  - The cache line keeps on moving between two processors
  - The processors are not really accessing or updating the same element, but whatever they are updating happen to fall within a cache line: not a true sharing, but false sharing
  - For shared memory programs false sharing can easily degrade performance by a lot
  - Easy to avoid: just pad up to the end of the cache line before starting the allocation of the data for the next processor (wastes memory, but improves performance)

Communication cost

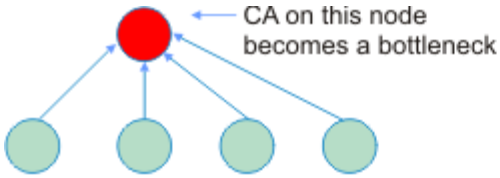
- Given the total volume of communication (in bytes, say) the goal is to reduce the end-to-end latency
- Simple model:

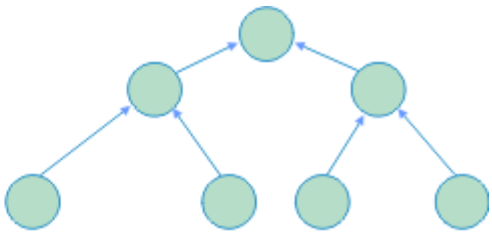
  T = f*(o + L + (n / m) / B + tc – overlap) where
  f = frequency of messages
  o = overhead per message (at receiver and sender)
  L = network delay per message (really the router delay)
  n = total volume of communication in bytes
  m = total number of messages
  B = node-to-network bandwidth
  t_c = contention-induced average latency per message
  overlap = how much communication time is overlapped with useful computation

- The goal is to reduce T
  - Reduce o by communicating less: restructure algorithm to reduce m i.e. communicate larger messages (easy for message passing, but need extra support in memory controller for shared memory e.g., block transfer)
  - Reduce L = number of average hops*time per hop
  - Number of hops can be reduced by mapping the algorithm on the topology properly e.g., nearest neighbor communication is well-suited for a ring (just left/right) or a mesh (grid solver example); however, L is not very important because today routers are really fast (routing delay is ~10 ns); o and t_c are the dominant parts in T
  - Reduce t_c by not creating hot-spots in the system: restructure algorithm to make sure a particular node does not get flooded with messages; distribute uniformly

Contention

- It is very easy to ignore contention effects when designing algorithms
  - Can severely degrade performance by creating hot-spots
- Location hot-spot:
  - Consider accumulating a global variable; the accumulation takes place on a single node i.e. all nodes access the variable allocated on that particular node whenever it tries to increment it





Scalable tree accumulation

Previous Next

Hot-spots

- Avoid location hot-spot by either staggering accesses to the same location or by designing the algorithm to exploit a tree structured communication
- Module hot-spot
  - Normally happens when a particular node saturates handling too many messages (need not be to same memory location) within a short amount of time
  - Normal solution again is to design the algorithm in such a way that these messages are staggered over time
- Rule of thumb: design communication pattern such that it is not bursty; want to distribute it uniformly over time

Overlap

- Increase overlap between communication and computation
  - Not much to do at algorithm level unless the programming model and/or OS provide some primitives to carry out prefetching, block data transfer, non-blocking receive etc.
  - Normally, these techniques increase bandwidth demand because you end up communicating the same amount of data, but in a shorter amount of time (execution time hopefully goes down if you can exploit overlap)

Summary

- Comparison of sequential and parallel execution
  - Sequential execution time = busy useful time + local data access time
  - Parallel execution time = busy useful time + busy overhead (extra work) + local data access time + remote data access time + synchronization time
  - Busy useful time in parallel execution is ideally equal to sequential busy useful time / number of processors
  - Local data access time in parallel execution is also less compared to that in sequential execution because ideally each processor accesses less than 1/P th of the local data (some data now become remote)
- Parallel programs introduce three overhead terms: busy overhead (extra work), remote data access time, and synchronization time
  - Goal of a good parallel program is to minimize these three terms
  - Goal of a good parallel computer architecture is to provide sufficient support to let programmers optimize these three terms (and this is the focus of the rest of the course)

Previous Next

Exercise : 1

These problems should be tried after module 08 is completed.

1. [10 points] Suppose you are given a program that does a fixed amount of work, and some fraction s of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on P processors. Derive a formula for execution time on P processors and establish an upper bound on the achievable speedup.

2. [40 points] Suppose you want to transfer n bytes from a source node S to a destination node D and there are H links between S and D. Therefore, notice that there are H+1 routers in the path (including the ones in S and D). Suppose W is the node-to-network bandwidth at each router. So at S you require n/W time to copy the message into the router buffer. Similarly, to copy the message from the buffer of router in S to the buffer of the next router on the path, you require another n/W time. Assuming a store-and-forward protocol total time spent doing these copy operations would be (H+2)n/W and the data will end up in some memory buffer in D. On top of this, at each router we spend R amount of time to figure out the exit port. So the total time taken to transfer n bytes from S to D in a store-and-forward protocol is (H+2)n/W+(H+1)R. On the other hand, if you assume a cut-through protocol the critical path would just be n/W+(H+1)R. Here we assume the best possible scenario where the header routing delay at each node is exposed and only the startup n/W delay at S is exposed. The rest is pipelined. Now suppose that you are asked to compare the performance of these two routing protocols on an 8x8 grid. Compute the maximum, minimum, and average latency to transfer an n byte message in this topology for both the protocols. Assume the following values: W=3.2 GB/s and R=10 ns. Compute for n=64 and 256. Note that for each protocol you will have three answers (maximum, minimum, average) for each value of n. Here GB means 10^9 bytes and not 2^30 bytes.

3. [20 points] Consider a simple computation on an nxn double matrix (each element is 8 bytes) where each element A[i][j] is modified as follows. A[i][j] = A[i][j] - (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/4. Suppose you assign one matrix element to one processor (i.e. you have n^2 processors). Compute the total amount of data communication between processors.

4. [30 points] Consider a machine running at 10^8 instructions per second on some workload with the following mix: 50% ALU instructions, 20% load instructions, 10% store instructions, and 20% branch instructions. Suppose the instruction cache miss rate is 1%, the writeback data cache miss rate is 5%, and the cache line size is 32 bytes. Assume that a store miss requires two cache line transfers, one to load the newly updated line and one to replace the dirty line at a later point in time. If the machine provides a 250 MB/s bus, how many processors can it accommodate at peak bus bandwidth?

Solution of Exercise : 1

1. [10 points] Suppose you are given a program that does a fixed amount of work, and some fraction s of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on P processors. Derive a formula for execution time on P processors and establish an upper bound on the achievable speedup.

Solution: Execution time on P processors, T(P) = sT(1) + (1-s)T(1)/P. Speedup = 1/(s + (1-s)/P). Upper bound is achieved when P approaches infinity. So maximum speedup = 1/s. As expected, the upper bound on achievable speedup is inversely proportional to the sequential fraction.

2. [40 points] Suppose you want to transfer n bytes from a source node S to a destination node D and there are H links between S and D. Therefore, notice that there are H+1 routers in the path (including the ones in S and D). Suppose W is the node-to-network bandwidth at each router. So at S you require n/W time to copy the message into the router buffer. Similarly, to copy the message from the buffer of router in S to the buffer of the next router on the path, you require another n/W time. Assuming a store-and-forward protocol total time spent doing these copy operations would be (H+2)n/W and the data will end up in some memory buffer in D. On top of this, at each router we spend R amount of time to figure out the exit port. So the total time taken to transfer n bytes from S to D in a store-and-forward protocol is (H+2)n/W+(H+1)R. On the other hand, if you assume a cut-through protocol the critical path would just be n/W+(H+1)R. Here we assume the best possible scenario where the header routing delay at each node is exposed and only the startup n/W delay at S is exposed. The rest is pipelined. Now suppose that you are asked to compare the performance of these two routing protocols on an 8x8 grid. Compute the maximum, minimum, and average latency to transfer an n byte message in this topology for both the protocols. Assume the following values: W=3.2 GB/s and R=10 ns. Compute for n=64 and 256. Note that for each protocol you will have three answers (maximum, minimum, average) for each value of n. Here GB means 10^9 bytes and not 2^30 bytes.

Solution: The basic problem is to compute the maximum, minimum, and average values of H. The rest is just about substituting the values of the parameters. The maximum value of H is 14 while the minimum is 1. To compute the average, you need to consider all possible messages, compute H for them, and then take the average. Consider S=(x0, y0) and D=(x1, y1). So H = |x0-x1| + |y0-y1|. Therefore, average H = (sum over all x0, x1, y0, y1 |x0-x1| + |y0-y1|)/(64*63), where each of x0, x1, y0, y1 varies from 0 to 7. Clearly, this is same as (sum over x0, x1 |x0-x1| + sum over y0, y1 |y0-y1|)/63, which in turn is equal to 2*(sum over x0, x1 |x0-x1|)/63 = 2*(sum over x0=0 to 7, x1=0 to x0 (x0-x1)+ sum over x0=0 to 7, x1=x0+1 to 7 (x1-x0))/63 = 16/3.

3. [20 points] Consider a simple computation on an nxn double matrix (each element is 8 bytes) where each element A[i][j] is modified as follows. A[i][j] =

A[i][j] - (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1])/4. Suppose you assign one matrix element to one processor (i.e. you have n^2 processors). Compute the total amount of data communication between processors.

Solution: Each processor requires the four neighbors i.e. 32 bytes. So total amount of data communicated is 32n^2.

4. [30 points] Consider a machine running at 10^8 instructions per second on some workload with the following mix: 50% ALU instructions, 20% load instructions, 10% store instructions, and 20% branch instructions. Suppose the instruction cache miss rate is 1%, the writeback data cache miss rate is 5%, and the cache line size is 32 bytes. Assume that a store miss requires two cache line transfers, one to load the newly updated line and one to replace the dirty line at a later point in time. If the machine provides a 250 MB/s bus, how many processors can it accommodate at peak bus bandwidth?

Solution: Let us compute the bandwidth requirement of the processor per second. Instruction cache misses 10^6 times transferring 32 bytes on each miss. Out of 20*10^6 loads 10^6 miss in the cache transferring 32 bytes on each miss. Out of 10^7 stores 5*10^5 miss in the cache transferring 64 bytes on each miss. Thus, total amount of data transferred per second is 96*10^6 bytes. Thus at most two processors can be supported on a 250 MB/s bus.

Previous Next