Resource Competition and Forced Parallelization

Assignment 3, 5DV050, Spring 2013

Due on 4 June 2013 at 16:40

Abstract

One of the defining features of contemporary multi-core processors is that the cores share resources in the form of caches and memory buses. In this assignment, we study how this fact could impact the design of algorithms for perfectly parallel problems. We develop microbenchmarks to quantify the negative effects of resource competition on Abisko and also quantify the positive effects of the so-called forced parallelization of each individual task. Finally, we discuss how to used forced parallelization to improve the performance of a perfectly parallel graph problem.

1 Introduction

A perfectly parallel problem is one which has already been decomposed into a large number of independent tasks. Scheduling $N$ independent tasks on $p \ll N$ identical and independent processors should result in close to perfect (linear) speedup even if $p$ is a very large number. The only real issue is how to balance the load. Under the mild assumption that the tasks do not vary wildly in execution time, load balancing should not be a big issue.

However, the key assumption is that the processors are independent in the sense that they do not compete for shared resources. This assumption does not hold today if we consider a processor to be a single core of a multi-core processor as cores typically share one or more levels of cache as well as a memory bus.

To be more concrete, consider a cache with capacity $C$ that is shared by $q$ cores. If we execute a sequential task on one core while keeping all other cores idle, then the task can monopolize the entire cache. In contrast, if all $q$ cores simultaneously execute independent sequential tasks, then the effective cache capacity per task is going to be reduced from $C$ to only $C/q$ as each core can now effectively use only its fair share of the cache. The performance will degrade as a result of the resource competition and less than perfect speedup will be observed.

The performance and scalability of perfectly parallel problems that suffer from shared resource competition can in some cases be improved by the forced parallelization of each

\footnote{We use the term forced parallelization to mean the parallelization of a task for some other purpose than to expose concurrency to keep processors busy.}
individual task. Since there is already many more tasks than processors \( p \ll N \), the further parallelization of each task is not motivated by exposing more concurrency but rather by transforming resource competition into cooperation.

This assignment consists of two parts: First, we aim to quantify the negative effects of resource competition and the positive effects of forced parallelization on the Abisko system using microbenchmarks. Second, we attempt to design a parallel algorithm for a perfectly parallel graph problem and use forced parallelization to improve the performance.

### 1.1 Shared resources on Abisko

A processor in the Abisko system consists of twelve cores partitioned into two NUMA nodes. Cores on different NUMA nodes share no resources. The six cores of a NUMA node share a memory bus as well as a 6 MiB L3 cache. The six cores are further partitioned into three modules consisting of two cores each. Each module has its own 2 MiB L2 cache, and each core in a module has a small private L1 cache. The L3 cache is mostly exclusive with respect to the L2 caches, which means that cache lines present in an L2 cache are not present in the L3 cache and vice versa, and only evictions from an L2 cache cause allocations in the L3 cache. In other words, there is an aggregate total of \( 6 + 3 \times 2 = 12 \) MiB of L2 and L3 cache capacity per NUMA node and thus \( 8 \times 12 = 96 \) MiB of L2 and L3 cache per Abisko node.

In addition to caches and a memory bus, the two cores in a module share a Floating Point Unit (FPU). If only one core is using the FPU, then the full capacity of that FPU is allocated to the core. On the other hand, if both cores use the FPU simultaneously, then the FPU is dynamically shared and each core receives approximately one half of the FPU capacity. As with cache and memory bus sharing, there are little to no fairness guarantees and one core might receive more than its fair share of the FPU. In this assignment, however, we will not concern ourselves with the effects of competition for the FPU resources and focus exclusively on the memory hierarchy.

### 2 Part 1: Benchmarking

In this part, we develop two microbenchmarks designed to stress the shared cache and memory bus resources. The first benchmark is designed to maximize the utilization of the available bandwidth via regular and contiguous memory accesses. The second benchmark uses irregular random memory accesses and is therefore sensitive to the latency of the caches and memory.

The effects of resource competition is sensitive to the working set size\(^2\). In particular, if the working set size is so small that it fits in the L2 cache but not in the L1 cache, then there will be competition for the shared L2 cache but no competition for neither the L3 cache nor the memory bus. As the working set size increases, the problem will spill over into the L3 cache and eventually into the memory. These transitions can be observed in

\(^2\)The working set size refers to the amount of memory required to solve a problem.
the performance of the benchmarks, for example by plotting performance versus working set size. Therefore, the benchmarks will be designed to allow the working set size to be chosen very precisely from the small to the very large in small increments.

2.1 Microbenchmarks

Your task is to design, implement, and run two microbenchmarks. The first one should be bandwidth-bound while the second one should be latency-bound. To achieve this, the first benchmark should feature a regular and contiguous memory access pattern while the second one should feature random and unpredictable memory accesses. In both cases, the working set size must be precisely controllable.

2.1.1 Benchmark 1: Contiguous memory accesses

You should provide the details for this benchmark. Make sure that the benchmark has a configurable working set size, that the task granularity is moderate, and that the working set is used repeatedly (or else caching would have no effect).

2.1.2 Benchmark 2: Random memory accesses

You should provide the details for this benchmark. Make sure that the benchmark has a configurable working set size, that the task granularity is moderate, and that the working set is used repeatedly (or else caching would have no effect). One idea is to use a technique known as pointer chasing.

2.2 Implementation

The two benchmarks can share a large portion of their code such as the management of threads, the binding of threads to physical cores, the synchronization of threads, and the measuring of time. Use this to your advantage and only customize the very small portion of the code that is unique to each benchmark.

2.2.1 Functionality

Given a perfectly parallel problem with \( N \) tasks and \( p \ll N \) cores, we can demonstrate the negative effects of resource competition by executing the tasks sequentially on a single core (execution time \( T_1 \)) as well as (dynamically) balanced across all \( p \) cores (execution time \( T_p \)). If there was no resource competition, then we would expect near perfect speedup (\( T_1/T_p \approx p \)).

To demonstrate the positive effects of forced parallelization we need additional functionality. We can simulate the effects of a forcibly parallelized task as follows. Let \( W \) denote the problem size of one task. Suppose that we want to forcibly parallelize it over \( q \leq p \) cores, then we simulate this by running \( q \) smaller tasks of size \( W/q \) with barrier
synchronization before and after. In effect, we are thereby simulating the best case scenario of a perfectly parallelizable task with perfect load balancing and locality.

2.2.2 Thread affinity

To bind threads to physical cores you should use the hwloc library, which is available on Abisko. You first load the appropriate module: module load hwloc. The command pkg-config --cflags hwloc then provides a list of necessary compiler flags and the command pkg-config --libs hwloc provides a list of necessary linker flags. You only need to use a very small subset of the otherwise fairly large hwloc API. In particular, you need the data types

- **hwloc_topology_t**
  Represents the hardware topology of a node.

- **hwloc_obj_t**
  Represents an object, such as a core.

and the functions

- **hwloc_topology_init**
  Initializes a topology object.

- **hwloc_topology_load**
  Loads hardware topology information.

- **hwloc_get_obj_by_type**
  Get an indexed object, such as a core.

- **hwloc_set_cpubind**
  Bind the calling thread to a specified set of cores (obj->cpuset).

The documentation for the hwloc library is available online. Make sure to use the version of the manual that corresponds to the version installed on Abisko (see module help hwloc). The 48 cores on one node of Abisko are numbered consecutively from 0 to 47. The numbering of cores is fairly natural (see the output of hwloc-ls for details) and means in particular that the cores 0 – 11 belong to the same processor, cores 0 – 5 belong to the same NUMA node (i.e., they share an L3 cache and a memory bus), and cores 0 – 1 belong to the same module (i.e., they share an FPU and an L2 cache).

2.3 Analysis

With the benchmarks implemented, we are now ready to run them on Abisko and analyze the results. We are primarily interested in the performance[^] as a function of the working

[^]: Number of accessed bytes per second or number of accesses per second.
set size. The aim of the computational experiments will be to gather enough of data points scattered over a carefully chosen range of working set sizes to draw conclusions and quantify various effects. A rule of thumb is to choose working set sizes up to twice the aggregate cache capacity (12 MiB). Note that we do not have to run on more than one NUMA node since cores on different NUMA nodes share no resources.

The performance depends on the working set size. For example, in a bandwidth-bound computation the performance will be greater for small problems and decreases when the working set spills over to the next level of cache. One aim of the analysis is to correlate the features of the plots of performance versus working set size with the characteristics of the cache hierarchy. In particular, is it possible to observe any effects of the shared L2 caches, the shared L3 cache, and the exclusive organization of the L2 and L3 caches? Try to offer plausible explanations for all observed features.

Finally, quantify the negative effects of resource competition (in a plot versus working set size) and the positive effects of forced parallelization (again in a plot versus working set size).

3 Part 2: Algorithm design

The second part of this assignment concerns the application of forced parallelization to a realistic problem. Given a description of the problem, you will be asked to design an algorithm that solves it while taking resource competition into account. You should explain in detail how the proposed solution will mitigate some of the competition for shared resources. Note that you will not be asked to implement the proposed solution; A conceptual design suffices.

A weighted directed graph is a graph with directed edges with edge weights. Formally, such a graph can be represented by a tuple $G = (V, E, \omega)$, where $V$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. An edge $(u, v) \in E$ from $u$ to $v$ has vertex $u$ as its tail and vertex $v$ as its head. The function $\omega : V \times V \to \mathbb{R}$ maps pairs of vertices to a non-negative edge weight. If $(u, v) \in E$ then $\omega(u, v)$ is finite, otherwise $\omega(u, v) = \infty$.

A path is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that each adjacent pair of vertices are connected by an edge, i.e., $(v_k, v_{k+1}) \in E$ for $k = 1, 2, \ldots, n - 1$. The length of a path is defined as the sum of the edge weights along the path.

It is frequently useful to know the shortest path between two vertices. Given a starting vertex $s$, the problem of finding the shortest paths from $s$ to all other vertices in the graph is the classical single-source shortest paths problem (SSSP). Dijkstra’s algorithm solves the SSSP problem efficiently if the graph is dense, i.e., if the number of edges is close to the maximum. The algorithm maintains a set $S$ of vertices whose shortest paths from $s$ are known. Initially, $S = \{s\}$. The distances from $s$ to all vertices in $V \setminus S$ are computed and the vertex $r \in V \setminus S$ that is closest to $s$ is selected for addition into $S$. The details of the algorithm, using an adjacency matrix representation of the graph, is given in the C-like code in Figure 1.

The distance array $d[]$ and the visited[] bitmap are initialized in lines 10 and 11,
Figure 1: Dijkstra’s algorithm for the SSSP problem.

```c
// INPUT
// int n - Number of vertices
// int s - Source vertex
// OUTPUT
// double d[n] - Length of SP from s to v is d[v]

// Initialize.
for (int v = 0; v < n; ++v) {
    d[v] = DBL_MAX;
    visited[v] = 0;
}
d[s] = 0.0;

// Set s as the first vertex to expand.
int r = s;
double dr = 0.0;

// Perform n-1 iterations.
for (int i = 1; i < n; ++i) {
    // Expand new vertex r.
    visited[r] = 1;
    for (int v = 0; v < n; ++v) {
        if (!visited[v]) {
            double dv = dr + A[r][v]; // Length of SP s, ..., r, v.
            if (dv < d[v]) {
                d[v] = dv; // Found new SP from s to v.
            }
        }
    }
    // Find unvisited vertex r closest to s.
    dr = DBL_MAX;
    for (int v = 0; v < n; ++v) {
        if (!visited[v]) {
            if (d[v] < dr) {
                r = v;
                dr = d[v];
            }
        }
    }
}
```
respectively. The trivially known distance to the source is set in line 13. The source vertex is chosen as the first vertex \( r \) to expand in line 16 and the corresponding distance \( d_r \) is set in line 17. The loop starting in line 20 has \( n - 1 \) iterations and in each iteration the previously selected vertex \( r \) is included in \( S \), expanded, and finally a new vertex \( r \) is selected for inclusion. Vertex \( r \) is marked as visited (included in \( S \)) in line 22. The loop in lines 23–29 updates the distance array \( d[] \) with respect to the vertex \( r \). More precisely, if the shortest distance \( d[r] \) from \( s \) to \( r \) plus the weight \( A[r][v] \) of the edge from \( r \) to the unvisited vertex \( v \) is less than the currently recorded distance \( d[v] \) (line 26), then \( d[v] \) is updated to the new value \( d[v] \) (line 27). The loop in lines 33–40 finds an unvisited vertex \( r \) with minimum recorded distance \( d[r] \). Both of the inner loops in Dijkstra’s algorithm can be parallelized: The first one trivially and the second one with a min reduction. The outer loop cannot be parallelized so easily due to a loop-carried dependence on \( r \). The computational complexity of Dijkstra’s algorithm is \( O(n^2) \) and the memory complexity is \( O(n) \).

3.1 The perfectly parallel problem

In some cases we are interested not only in the shortest paths from \( s \) to the other vertices but the shortest paths between any pair of vertices. This is the so-called all-pairs shortest paths problem (APSP). We can use Dijkstra’s algorithm to solve also the APSP problem simply by running the algorithm \( n \) times with a different source each time\(^4\). If we have \( p \ll n \) processors, then we can expect close to perfect linear speedup for this perfectly parallel problem in the absence of resource competition. If we use the cores of a multi-core processor, however, the performance might degrade due to the presence of shared caches and memory buses. Your task is to design a (possibly) more efficient algorithm for the APSP on multi-core processors based on the scheme above. You will not be asked to actually implement and test the algorithm, merely provide a conceptual design. You should give a detailed explanation of how your proposed algorithm mitigates resource competition. Moreover, you should estimate the size of the improvement on Abisko (using the results of the microbenchmarks) and describe how the improvement depends on the size of the problem. This is very valuable information for users that would like to know if their particular application can benefit from the use of forced parallelization.

4 Summary

This section provides a summary/checklist of the major steps required to complete this assignment.

1. Implement a framework for the two benchmarks.
   - Bind threads to cores.

\(^4\)This solution is known as the source-partitioned algorithm.
• Allocate memory on the local NUMA node.
• Dynamic load balancing.

2. Design the two benchmarks.
• Contiguous memory accesses.
• Random memory accesses.
• Moderate task granularity.
• Finely configurable working set size.

3. Benchmark one NUMA node of the Abisko system.
• Job submission script.
• Range of working set sizes.
• Measurement noise.

• Understand Dijkstra’s algorithm.
• Understand the perfectly parallel problem (APSP).
• Identify source(s) of resource competition.
• Design parallel algorithm to reduce resource competition.
• Explain how the proposed algorithm reduces resource competition.
• Estimate size of improvement and associated range of problem sizes on Abisko.