ParallelProgramming

January 10, 2018

Contents

#+:TITLE Parallel Programming

1 Day 1

Cindy is working on simulations! Quantum physics and lotsa cool stuff Raison d'etre of parallel programming: Lots of data, hard simulations, lots of results

1.1 Parallel Programming

The use of *multiple processors* (CPUs, GPUS, devices, etc) for problem-solving.

Why is it useful?

- Each core can perform the same operations, so with many more cores, many more operations can be performed.
- Each core has its own memory

However…

- Parallelizing code requires significant architectural changes
- It doesn't *magically* make serial code faster
- Communication between units introduces significant overhead
- Total time is determined by the slowest core. It becomes necessary to *synchronize* workers.
- Not all algorithms can be parallelized!

One of the main problems to solve in parallel computation is managing *data transfer times*, i.e. the pipes between different cores. This introduces race conditions, locking, etc.

1.1.1 TODO Example of an Unparallelizable system

x1 = f(x1, x2, x3)

1.1.2 Hardware

In a common computer, most memory is `DRAM`, which has high capacity but high latency (60-200 cycles). However, there's a CPU cache that allows quick access to frequently-used memory (2-10 cycles depending on cache level).

Even simple changes in array access can change memory access efficiency! By manipulating array access it is possible to exploit the L caches.

1. Levels

(a) Core A single core has access to all three L levels, but is limited to performing serial/sequential operations.

(b) Chip/Socket

With multiple cores, each core has its own individual L1 and L2 cache, but the L3 cache is typically shared between these cores to allow *shared memory access*. This is the first option for multiprocessing.

(c) Motherboard

Multiple chips in a single motherboard are connected through the DRAM memory exclusively and, in a different way than with individual cores, have Non-Uniform Memory Access (NUMA).

(d) Computer Cluster Multiple motherboards in a rack are linked through *switches* to other racks, which may be organized in different ways depending on the chosen topology.

(e) Memory types

- Sequential processing (cores) – L Caches
- Shared Memory (chips and motherboards)
- Distributed Memory

1.1.3 OpenMP

Application Programming Interface – Manages memory and task assignment for *shared memory*

1.1.4 MPI

Message Passing Interface – Communication between nodes in a *distributed memory system*.

1.1.5 Performance metrics

Speedup – Performance gain in switching from serial to parallel: $s_p = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$
Efficiency – Speedup per core: $E_p = \frac{s_p}{p}$

Allows us to gauge whether parallelizing is actually worth it. It's not always a worthwile investment, and this isn't even considering communication!

1.1.6 Communication

- Latency: Cost of sending 0B – time related
- Bandwidth: Communication rate per second

Granularity: Level of parallelization

1.1.7 Important ideas

Parallelism depends on *processing* and *communication*

1.1.8 CACHE COHERENCE

1.2 OpenMP

API to write multithreaded programs. It:

- Contains compilation directives and runtime libraries
- Facilitates implemntation in Fortran, C, and C++
- Support for multiple software and hardware architectures
- Comes by default in gcc and clang

CHeck out the site at OpenMP.com

1. Fork and Join model A single-threaded application stops to perform a parallel action, complete it, and return to a single thread.

1.2.1 Practical Problem: calculating Pi

π can be estimated by calculating the integral $\int_0^1 \frac{4}{1+x^2}dx$ with the desired approximation accuracy through the Riemann sum $\sum_{i=0}^{n-1} f(x_1) \cdot \frac{1}{n}$
The magical OpenMP directive: `#pragma omp parallel`

1.2.2 Managing variable access

- Private
- Shared
- FirstPrivate: Variable created on beginning of parallel block
- LastPrivate: Variable created at end of parallel block

1.2.3 Thread scheduling

2 Day 2

2.1 Distributed memory

There's no longer free access to memory located at different nodes – data must be specifically requested.

2.2 MPI

Message Passing Interface for distributed memory systems– It's a IEEE *standard*, not any particular piece of code.

2.2.1 Calculating pi with distributed memory

We must explicitly send and receive the data managed by each node using `MPI_SEND` and `MPI_RECV`.

Parallel code *must* be wrapped in `MPI_Init(&argc, &argv)` and `MPI_Finalize`

Can define different rank sub-worlds

2.2.2 1-1 communication

`MPI_SEND(&total, num_vars=1, var_type=MPI_DOUBLE, to_rank=1, tag, MPI_COMM_WORLD)` `MPI_RECV(&sum, num_vars=1, var_type=MPI_DOUBLE, from_rank=0, tag, MPI_COMM_WORLD, status)`

Deadlocking

2.3 Task management