Levi Overcast
DSA F21
October 1

# Parallel & Distributed Computing
# Homework 1

1. Define parallel programming and how it is different from concurrent programming.

   A program is said to be parallel if multiple *instructions* are executed at the same time.

   On the other hand, a program is said to be concurrent if multiple *tasks* are *in progress* at the same time.

   Consider a program that adds vectors. On modern processors under the x86 architecture, a single 512-bit ZMM register can pack sixteen 32-bit floating point numbers and perform a vectorized addition on them with a single instruction. This is an example of *parallelism without concurrency.*

   Imagine a data server that can multitask client processes. Suppose two clients both want write access to the same data at the same time. Obviously, the server cannot deliver write access to both processes in parallel. The server might implement a mutex lock to solve this problem. This is an example of *concurrency without parallelism.*

2. Exercise 1.3 from Chapter 1 in Pacheco.

   **1.3** Try to write pseudo-code for the tree-structured global sum illustrated in Figure 1.1. Assume the number of cores is a power of two (1, 2, 4, 8, ...).

```
// suppose the IDs are {0..num_cores}
int num_cores;
core_list cores = create_cores(num_cores);
list<int> powers = {1..2^lg(num_cores)};
for (p : powers) {
    /* Cores which are 0 mod 2*p receive
     * from the core which is this.ID+p. */
    cores.recv(p);
    /* Cores which are p mod 2*p will send
     * to the core which is this.ID-p. */
    cores.send(p);
    cores.add();
}
return cores[0].sum;
```

3. Optional: Chapter 1: 1.4.

**1.4** Implement this algorithm in pseudo-code using the bitwise exclusive or and the left-shift operator.

As above, the routines that do all the work here are `cores.send` and `cores.receive`, which in this case accept bit vectors that identify the cores which do the respective operations.

```
rvector = svector = 0; /* global */

private {
    core_id; /* bit vector, a power of 2 */
    sum;
}

assign_ids(); /* tell each core what its id is */
compute_id_send(); /* tell each core to compute the id that it will send to */
/* lowest power p of 2 that its id is not remainder 0, then subtract the modulus. */

int bit;
list<int> powers = {1..2^(lg(num_cores)-1)};
for (p : powers) {
    rvector = 0;
    for (bit = 1; bit < num_cores; bit <<= 2*p) /* mask, selecting for */
        rvector = rvector ^ bit;                /* 0s mod 2, then 0s mod 4, etc. */
    svector = rvector << p; /* offset to calculate the senders. */

    cores.sync();
    cores.receive(); /* Each core will & its id with the global rvector. */
    cores.send(); /* Similarly. */
    sum += cores.add(); /* 0 if not receiving. */
}

/* end parallel */
return cores[0].sum;
```

4. Exercises 2.6, 2.7, 2.10 from Chapter 2.

**2.6** Suppose that a vector processor has a memory system in which it takes 10 cycles to load a single 64-bit word from memory. How many memory banks are needed so that a stream of loads can, on average, require only one cycle per load?

Given that memory latency is 10 clock cycles, we need at least 10 memory banks to achieve an amortized throughput of 1 word per clock cycle in the long run. 16 banks is preferable since it is simpler and more efficient to implement a whole number of bits for bank addressing.

Given that the processor is a vector processor, a vector register can chain the load/store operations so that the results of load operations may be stored before all the load operations are completed. The result is that after the first ten cycles, the result of the first load operation will store, and from then on one store operation will complete per clock cycle. The average throughput limits to 1.

**2.7** Discuss the differences in how a GPU and a vector processor might execute the following code:

```
sum = 0.0;
for (i = 0; i < n; i++) {
    y[i] += a*x[i];
    sum += z[i]*z[i];
}
```

The term *GPU* designates a broad class of implementations, ranging from multi-threaded SIMD model to approaching a MIMD model in modern systems. The qualifications for breadth apply doubly for the term *vector processor*, which is used to describe systems as diverse as the `CRAY-2` supercomputer and Intel's AVX SIMD extension.

That said, I may attempt to speak in general terms.

If we suppose our vector processor is composed of a single core, then the biggest difference between the two will be that the GPU uses thread-level parallelism without instruction-level parallelism, and the vector processor uses instruction-level parallelism without thread-level parallelism.

Many of the other differences are implementation-dependent. A modern vector processor would typically have multiple cores, though not as many as the GPU, but with a higher clock rate.

The `CRAY-2` had eight vector registers composed of 64 64-bit elements, along with 64-bit scalar registers. And each background processor had a set of functional units which were capable of operating in parallel.[1] The execution of the above loop might have involved loading vector registers with $x, y$ and $z$, loading scalar registers with $a$ and $sum$, computing the addition and multiplication operations in parallel in functional units spread over the background processors and chaining ('tailgating' in the manual) the results in the vector registers before storing.

By contrast, on a modern GPU, the execution of the loop might begin by carving $x, y, z$ into blocks of threads, performing all the scalar multiplications on $x$ and the dot-product multiplications on $z$ first, since GPUs are optimized for executing one instruction on a datastream, then executing the add operations and finally the store operations on $y$ and $sum$.

**2.10** Suppose a program must execute $10^{12}$ instructions in order to solve a particular problem. Suppose further that a single processor system can solve the problem in $10^6$ seconds (about 11.6 days). So, on average, the single processor system executes $10^6$ or a million instructions per second. Now suppose that the program has been parallelized for execution on a distributed-memory system. Suppose also that if the parallel program uses $p$ processors, each processor will execute $10^{12}/p$ instructions and each processor must send $10^9(p-1)$ messages. Finally, suppose that there is no additional overhead in executing the parallel program. That is, the program will complete after each processor has executed all of its instructions and sent all of its messages, and there won't be any delays due to things such as waiting for messages.

[1]The manual is available for free download. `http://www.mirrorservice.org/sites/www.bitsavers.org/pdf/cray/CRAY-2/HR-0200-0D_CRAY-2_Computer_Systems_Functional_Description_Jun89.pdf`

(a.) Suppose it takes $10^{-9}$ seconds to send a message. How long will it take the program to run with 1000 processors, if each processor is as fast as the single processor on which the serial program was run?

$$\begin{aligned}\text{Total Time} &= 1000 \text{ processors} \times (10^{12}/p \times 10^{-6}\text{sec/instr} + 10^9(1000-1) \times 10^{-9}\text{sec/msg}) \\ &= 10^6 \text{ seconds spent calculating} + 999 \text{ seconds spent messaging} \\ &= 10^6 \text{ seconds.}\end{aligned}$$

(b.) Suppose it takes $10^{-3}$ seconds to send a message. How long will it take the program to run with 1000 processors?

$$\begin{aligned}\text{Total Time} &= 1000 \text{ processors} \times (10^{12}/p \times 10^{-6}\text{sec/instr} + 10^9(1000-1) \times 10^{-3}\text{sec/msg}) \\ &= 10^6 \text{ seconds spent calculating} + 10^6 \times 999 \text{ seconds spent messaging} \\ &= 10^9 \text{ seconds (31 years).}\end{aligned}$$