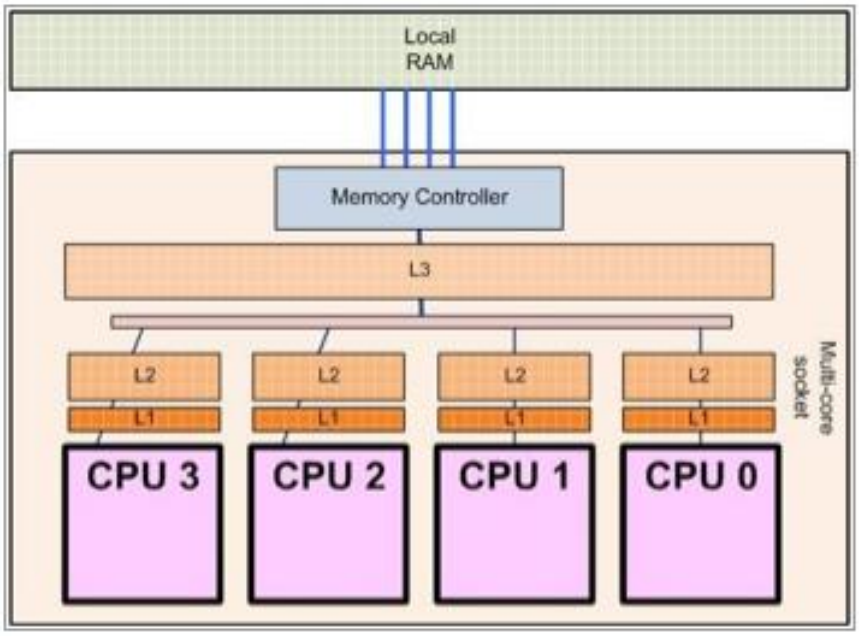
CPS 5401 Fall 2014 Answers
Shirley Moore, Instructor
Test 1 November 19, 2014

Please show all your work.

1. Draw a sketch of the extended von Neumann architecture for a 4-core multicore processor with three levels of cache and only the last level shared. Please label all the parts of your drawing.



2. Suppose a 2 GHz processor has a floating point unit (FPU) with a five-stage pipeline and that there is sufficient redundancy so that the FPU resources can be allocated as follows:

1) FDIV (divide) instructions require all FPU resources, so no other FP instruction can be in the pipeline while the FDIV is executing.
2) One FMUL can be executed simultaneously with one or two FADD instructions. (i.e., all three can occupy the same stage of the pipeline simultaneously).
3) Three FADD instructions can be executed simultaneously.

a. Assuming a perfect balance of FMUL and FADD instructions and no pipeline stalls, what would be the FLOPS rate of the FPU?

The best balance for this pipeline is either one FMUL and 2 FADD and 3 FADD instructions. After the first five cycles to warm up the pipeline, three operations will finish execution every cycle. Thus, the asymptotic rate will be 3 flops/cycle x 2 Gcycles/sec = 6 GFLOPS/s.

b. Assume that after every ten FMUL/FADD pairs, there is an FDIV instruction. Now what would be the FLOPS rate of the FPU?

   The pipeline will need to be flushed before and after execution of an FDIV instruction. Execution of ten FMUL/FADD pairs (20ops) will require 5+9 = 14 cycles (5 cycles to fill the pipeline and produce the first results and then one op per cycle for the next 9 cycles), and execution of one FDIV instruction will require 5 cycles. Thus, the overall rate will by 21 ops/19 cycles, or 1.1 ops/cycle or 2.2 GFLOPS/s.

c. Assuming 64-bit floating point numbers and 256-bit vector registers and operations, how would your answer to part a change?

   Assuming that the vector operations are pipelined, the FLOPS rate in part a will be multiplied by 4, giving a rate of 24 GFLOPS/s.

3. Consider the pseudocode below for matrix-vector multiply. Assume the data consist of 8-byte floating point numbers.

```
for i = 1 to 1024
  y[i] = 0.0
  for j = 1 to 1024
    y[i] = y[i] + a[i,j] * x[j]
  endfor
endfor
```

Assume a fully associative 128KB cache with 64-byte cache lines and least recently used (LRU) replacement policy.

a. What would be the approximate cache hit rate assuming row major order for storing the array?

   The only reuse of array a is reuse of the remainder of a cache line after it has been loaded. Since the cache cannot simultaneously hold a row of array a and the vector x, the cache lines for x will be evicted before they are reused. Thus, since there are 8 doubles in each cache line, the cache hit rate will be 7/8.

b. What would be the approximate cache hit rate assuming column major order for storing the array?

   With column major order, a new cache line will be loaded every time an element of a is accessed. Only cache lines for x will have any reuse. Thus, the cache hit rate will be $7/8 \times (2^{10}/(2^{20}+2^{10})) \approx 7/2^{13} \approx 0$.

c. Assuming column major order for storing the array, rewrite the code to achieve a better cache hit rate. (You can put your answer to the right of the above code).

```
for i = 1 to 1024
   y[i] = 0.0
endfor
for j = 1 to 1024
   for i = 1 to 1024
      y[i] = y[i] + a[i,j] * x[j]
   endfor
endfor
```

d. If we had a set-associative cache rather than a fully associative cache, what would we need to do to the data layout to achieve similar cache hit rates? Explain.

We don't really know in what order the arrays are laid out in memory, but generally large powers of two for array sizes are bad, since they set up a resonance with mapping to cache sets. For example, for the code in part c, if we had a 2-way set associative cache, it's possible that corresponding elements of x, y, and a row of a could all map to the same cache set and cause extra conflict cache misses. The way we can prevent this is to pad arrays so that their sizes are not large powers of two.

4. a. Compare and contrast shared memory and distributed parallel architectures (that is, describe similarities and differences).

Each core in a shared memory node can access any memory location on that node, whereas cores or processes on different nodes in a distributed memory architecture only access memory on their own node. Both architectures can support parallel programming, but only shared memory architectures support threaded parallel programming models.

b. We have discussed the OpenMP and MPI parallel programming paradigms. On which of the architectures in part a can each of these be used? Explain.

MPI can be used on either distributed or shared memory architectures, since MPI uses message passing which can be implemented with either shared or distributed memory.
OpenMP can only be used on shared memory architectures since OpenMP uses reads and writes to shared memory for communication.

5. Assume a ~~multicore~~ processor that runs at 3GHz has a pipelined floating point unit capable of executing a 256-bit wide vector multiply-add instruction each cycle, and that the measured maximum effective memory bandwidth is 48 GB/sec.

a. Sketch the roofline model for this processor, including the ceiling for non-vectorizable code.

The roofline model has a horizontal roofline at 3 x 2 x 4 = 24 GFLOPS, and a slanted memory bandwidth roofline with a slope of 48/8 = 6 G double words/sec. For non-vectorizable code, there is a horizontal ceiling at 6 GFLOPS. The machine balance point, assuming vectorization, is at an operational intensity of 4 FLOPS/double word.

b. What would be the maximum FLOPS rate achieved by the matrix-vector multiply code from problem 3a, assuming perfect overlap of computation and communication with memory? Show this on your roofline model sketch.

Assuming the elements of y are kept in a register during a single execution of the outer loop, the inner loop has an operational intensity of 2 FLOPS/2 dw = 1. Thus, the code is memory bound and the maximum performance of 6 GFLOPS is given by the memory bandwidth roofline.

6. Is the MPI code below safe from deadlock? Explain why or why not. If not, rewrite the code so that it is safe from deadlock.

```
destrank = (myrank + 1) % numtasks;
srcrank = (myrank – 1 + numtasks) % numtasks;
MPI_Send(sendbuf, 10, MPI_INT, destrank, 1, MPI_COMM_WORLD);
MPI_Recv(recvbuf, 10, MPI_INT, srcrank, 1, MPI_COMM_WORLD, &status);
```

Each process is sending to the process with rank one higher than itself and receiving from the process with rank one lower, with wraparound. The code is not safe from deadlock since the sends are blocking and may not be buffered. Hence, there could be a cycle of each process i waiting on the next process (i+1)%numtasks to post its receive before process i can complete its send. There are several solution to the problem. One is to have even numbered processes send first and receive second and odd numbered processes receive first and send second, as shown below.

```
destrank = (myrank + 1) % numtasks;
srcrank = (myrank – 1 + numtasks) % numtasks;
if ((myrank %2) == 0) {
    MPI_Send(sendbuf, 10, MPI_INT, destrank, 1 MPI_COMM_WORLD);
    MPI_Recv(recvbuf, 10, MPI_INT, srcrank, 1, MPI_COMM_WORLD, &status);
} else {
    MPI_Recv(recvbuf, 10, MPI_INT, srcrank, 1, MPI_COMM_WORLD, &status);
    MPI_Send(sendbuf, 10, MPI_INT, destrank, 1 MPI_COMM_WORLD);
}
```

7. Consider the OpenMP code below. Is it safe from race conditions? Explain why or why not. If not, rewrite the code so that it does not have a race condition. Assume arrays a and b are already initialized.

```
sum = 0;
#pragma omp parallel for
for (i=0; i<n; i++) {
   sum += sqrt(a[i])*b[i];
}
```

The code is not safe from race conditions since sum is a shared variables and updates are not atomic. The best solution is to use the reduction clause, as shown below, which tells the OpenMP runtime to aggregate the updates in a safe manner.

```
sum = 0;
#pragma omp parallel for reduction (+: sum)
for (i=0; i<n; i++) {
  sum += sqrt(a[i])*b[i];
}
```

8. Assume that vectors a and b both of length n are to be distributed from the root process with rank 0 to p processes (which includes the root) in 1D block decomposition. You may assume that n is a multiple of p. Then the processes call a routine named Compute with arguments n, alocal, blocal, clocal, and an MPI communicator, with the local results being placed in clocal. Finally, the results are gathered back to the root process into vector c of length n. Write code to carry out these steps, using MPI collective calls for the communication. The prototypes for MPI_Bcast, MPI_Scatter, and MPI_Gather are given below. Assume that the storage for a, b, and c has already been allocated on the root process, but that the storage for alocal, blocal, and clocal has not been allocated on any process.

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int
root, MPI_Comm comm )

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm)

int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype
sendtype,void *recvbuf, int recvcount, MPI_Datatype recvtype,
int root, MPI_Comm comm)
```

Sorry I was not clear that I meant the vectors are to be distributed with a block distribution. If you interpreted the question to mean that the entire vectors a and b would be distributed to each process, please let me know and I will regrade.

Assuming block distribution of the vectors and that the root process has rank 0:

```
nlocal = n/p;
alocal = malloc(nlocal * sizeof(double));
blocal = malloc(nlocal * sizeof(double));
clocal = malloc(nlocal * sizeof(double));
MPI_Scatter(a, nlocal, MPI_DOUBLE, alocal, nlocal, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
MPI_Scatter(b, nlocal, MPI_DOUBLE, blocal, nlocal, MPI_DOUBLE, 0,
            MPI_COMM_WORLD);
Compute(n, alocal, blocal, clocal, MPI_COMM_WORLD);
MPI_Gather(clocal, nlocal, MPI_DOUBLE, c, nlocal, MPI_DOUBLE, 0,
           MPI_COMM_WORLD);
```

9. a. Given that one-third of the serial execution time of a code is for a portion that cannot be parallelized, compute the maximum speedup possible using an unlimited number of processors.

$S = \frac{T_s}{T_p}$ where Ts is the serial execution time and Tp is the parallel execution time.

Normalizing the serial execution time to 1,

$$\lim_{p\to\infty} \frac{1}{\frac{1}{3} + \frac{2/3}{p}} = 3$$

b. Assume same as part a except that you are limited to 64 processors. Now what is the maximum speedup?

$$S = \frac{1}{\frac{1}{3} + \frac{2/3}{64}} = \frac{1}{\frac{1}{3} + \frac{1}{96}} \approx 2.91$$

c. Assume same as part b except that communication overhead is sqrt(p). Now what is the maximum speedup?

Since we don't have absolute times, we will assume that the overhead a multiplicative factor of sqrt(p).

$$S = \frac{1}{\frac{1}{3} + \frac{2/3}{64} \cdot 8} = \frac{1}{\frac{1}{3} + \frac{1}{12}} = \frac{12}{5} = 2.4$$