CSCI596 Assignment 1—Complexity, Flop/s and Message Passing Interface

Due: September 12 (Wed), 2018, 11:59 pm

Part I. Computational Complexity and Flop/s Performance

In this part, you will perform small hands-on exercises to get a feel for computational complexity and flop/s (floating-point operations per second) performance of a computer. Submit your answers to the following two questions.

I-1. Measuring Computational Complexity

Download the data file, `MDtime.out`, from the course home page for assignment 1. In the two-column file, the left column is the number of atoms, N, simulated by the `md.c` program, whereas the right column is the corresponding running time, T, of the program in seconds. Make a log-log plot of T *vs*. N. Perform linear fit of $\log T$ *vs*. $\log N$, *i.e.*, $\log T = \alpha \log N + \beta$, where α and β are fitting parameters. Note that the coefficient α signifies the power with which the runtime scales as a function of problem size N: $T \propto N^{\alpha}$. ***Submit the plot and your fit of α.***

I-2. Theoretical Flop/s Performance

Suppose you have a quadcore processor (a processor equipped with 4 processing cores) operating at a clock speed of 3.0 GHz (*i.e*., clock ticks 3×10^9 times per second), in which each core can operate 1 multiplication and 1 addition operations per clock cycle (*e.g*., using a special fused multiply-add (FMA) circuit). Assume that each multiply or add operation is performed on vector registers, each holding 4 double-precision operands (Fig. 1).[*] ***What is the theoretical peak performance of your computer in Gflop/s (gigaflop/s or 10^9 flops)?***

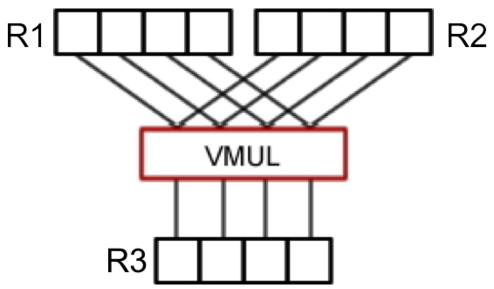

Fig. 1: Vector multiplier (VMUL) loads data from two vector registers, R1 and R3, each holding 4 double-precision numbers, concurrently performs 4 multiplications, and stores the results on vector register R3.

Part II. Implementing Your Own Global Summation with Message Passing Interface

In this part, you will write your own global summation program (equivalent to `MPI_Allreduce`) using `MPI_Send and MPI_Recv`. Your program should run on $P = 2^l$ processors ($l = 0, 1,...$). Each process contributes a partial value, and at the end, all the processes will have the globally summed value of these partial contributions.

Your program will use a communication structure called butterfly, which is structured as a series of pairwise exchanges (see the figure below where messages are denoted by arrows). This structure allows a global reduction among P processes to be performed in $\log_2 P$ steps.

[*] See FLOPS in Wikipedia [https://en.wikipedia.org/wiki/FLOPS].

```
    a000 + a001 + a010 + a011 + a100 + a101 + a110 + a111
=   ((a000 + a001) + (a010 + a011)) + ((a100 + a101) + (a110 + a111))
```

At each level *l*, a process exchanges messages with a partner whose rank differs only at the *l*-th bit position in the binary representation (Fig. 2).

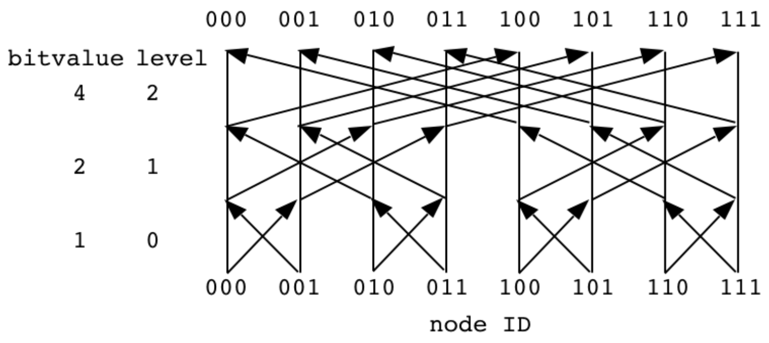

Fig. 2: Butterfly network used in hypercube algorithms.

HYPERCUBE TEMPLATE

We can use the following template to perform a global reduction using any associative operator `OP` (such as multiplication or maximum), `(a OP b) OP c = a OP (b OP c)`.

```
procedure hypercube(myid, input, logP, output)
begin
  mydone := input;
  for l := 0 to logP-1 do
  begin
    partner := myid XOR $2^l$;
    send mydone to partner;
    receive hisdone from partner;
    mydone = mydone OP hisdone
  end
  output := mydone
end
```

USE OF BITWISE LOGICAL XOR

Note that

```
0 XOR 0 = 1 XOR 1 = 0;
0 XOR 1 = 1 XOR 0 = 1.
```

so that `a XOR 1` flips the bit `a`, *i.e.*,

$$a \text{ XOR } 1 = \bar{a}$$
$$a \text{ XOR } 0 = a$$

where $\bar{a}$ is the complement of `a` ($\bar{a}$ = 1|0 for `a` = 0|1). In particular, `myid XOR` 2^l reverses the *l*-th bit of the rank of this process, `myid`:

$$\text{abcdefg XOR 0000100} = \text{abcd}\bar{\text{e}}\text{fg}$$

Note that the XOR operator is ^ (caret symbol) in the C programming language.

ASSIGNMENT

Complete the following program by implementing the function, `global_sum`, using `MPI_Send` and `MPI_Recv` functions and the hypercube template given above.

Submit the source code as well as the printout from a test run on 4 processors and that on 8 processors.

```
#include "mpi.h"
#include <stdio.h>

int nprocs;  /* Number of processors */
int myid;    /* My rank */

double global_sum(double partial) {
  /* Implement your own global summation here */
}

int main(int argc, char *argv[]) {
  double partial, sum, avg;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  partial = (double) myid;
  printf("Node %d has %le\n", myid, partial);

  sum = global_sum(partial);

  if (myid == 0) {
    avg = sum/nprocs;
    printf("Global average = %le\n", avg);
  }

  MPI_Finalize();

  return 0;
}
```
