2-D Potts Model Simulation using Metropolis Heatbath in HPJava

Yuhong Wen, Bryan Carpenter

September 22, 1997

1 Introduction

The initial 2-D Potts Model Simulation using Metropolis Heatbath has been implemented in Fortran 90 and HPF in NPAC. We will introduce its implementatin in HPJava here.

The code sets up a 2-D Q-State Potts model, and evolves it using the classic Metropolis heatbath algorithm. In this model each site may take on one of Q states, and each site interacts with its neighbour if both states are equal.

2 Implementation

In terms of communications in the parallelised HPF code, the main cost comes from the use of the CSHIFT() intrinsic functions in directions where the data has been distributed, since the 2-D array of sites is block distributed in one direction. Data transfer is near-neighbour between sites.

In HPJava implementation, we use the new feature of communication `writeHalo`, which only copies the necessary neighbour elements between the processors, instead of moving the whole data along the sepecific dimention of the array. This will dramaticly reduce the communication cost and memory copy operations when updating the energy states.

Another difference in the HPJava implementation is that we can provide an efficient and parallelized implementation of the random number generator locally in each processor in HPJava. The locally controlled operations and local temporary values will particularly be of benefit to spin physics codes such as this Potts model.

[This example does not respect the standard variable usage rules in the case of random number state `xsubi`*.* `xsubi` *is not a distributed array, but it has different values on different processes. Whether this should be allowed is an open question. Incidentally, we do not endorse the specific random number generator or mode of initialization for accurate simulations!]*

```
class Potts {
  int wlo [[]] = {1, 1}, whi [[]]]] = {1, 1} ;        // ghost width for 'writeHalo'
  int mode [[]] = {CYCL, CYCL} ;          // boundary conds for 'writeHalo'

  void main(int argc, char **argv) {

    int nprocs = AdlibNProcs() ;

    Procs1 p = new Procs1(nprocs) ;

    on(p) {
        const int w [2] = {1, 1} ;      // ghost area for 'writeHalo'
        int mode[] = {CYCL, CYCL} ;     // boundary conds for 'writeHalo'

        int m = 64;
        int n = 64;

        Range x = new Range(m) ;
        Range y = new BlockRange(n, p.dim(0)) ;

        int q = 4;

        float cup = -4.0;

        int latt[[,]] = new int [[x, y]] ;
        int latt_old[[,]] = new int [[x, y]] ;

        char result1[[,]] = new char [[x, y]] ;
        char result2[[,]] = new char [[x, y]] ;
        char result3[[,]] = new char [[x, y]] ;

        int itmp[[,]] = new int [[x, y]] ;


        int nmeasure = 20;
        int niter = 10;

        int xsubi [3] = new int [3] ;

        float t1, t2;

//  Set up a random number generator.
        setup_random( xsubi, 12345 );

//  Initialize the latt array.
        over(x) {
            over(y) {
                latt[[x, y]] = int( erand48(xsubi) * q ) ;
            }
        }

        for (int imeasure = 1; imeasure<=nmeasure; imeasure++) {

//  Use ghost area to get all four neighbor values.
            Adlib.copy(latt_old, latt) ;
            Adlib.writeHalo(latt_old, w, w, mode) ;

            over (x) {
                over (y) {
                    int latt_val = latt[[x, y]] ;
                    itmp[[x, y]] = latt_val == latt_old[[x >> 1, y]] +
                                   latt_val == latt_old[[x << 1, y]] +
                                   latt_val == latt_old[[x, y >> 1]] +
                                   latt_val == latt_old[[x, y << 1]] ;
                }
            }

            Adlib.gprintf("\t# Bonds = %d\n",sum( itmp ));
```

```
            t1 = (float)clock()/CLOCKS_PER_SEC;
            for (int iter=1; iter<=niter; iter++) {

              for (int parity = 0; parity < 2; parity ++) {

                    Adlib.copy(latt_old, latt) ;
                    Adlib.writeHalo(latt_old, w, w, mode) ;

                    over (x) {
                      over (y | (parity + x) % 2 : : 2) {
                        int latt_val = latt[[x, y]] ;
                        int itmp_val = (latt[[x, y]] +
                                       int(erand48(xsubi) * q - 1) + 1) % q ;
                        float  p  = (latt_val == latt_old[[x >> 1, y]] +
                                     latt_val == latt_old[[x << 1, y]] +
                                     latt_val == latt_old[[x, y >> 1]] +
                                     latt_val == latt_old[[x, y << 1]] -
                                     itmp_val == latt_old[[x >> 1, y]] -
                                     itmp_val == latt_old[[x << 1, y]] -
                                     itmp_val == latt_old[[x, y >> 1]] -
                                     itmp_val == latt_old[[x, y << 1]]) * cup ;
                        if (erand48(xsubi) < exp( p ) )
                                latt[[x, y]] = itmp_val ;
                      }
                    }

               }

            }
            t2 = (float)clock()/CLOCKS_PER_SEC;
            Adlib.gprintf("Time: %f\n", t2 - t1);
        }

//  Output final value of 'latt'
        FILE* ifile = Adlib.gfopen("out.ppm", "w") ;

        Adlib.gfprintf(ifile, "P6\n");
        Adlib.gfprintf(ifile, "# 2d Potts Model\n");
        Adlib.gfprintf(ifile, "%12d%12d\n",m,n);
        Adlib.gfprintf(ifile, "%8d\n",127);

        over(x) {
            over(y) {
                switch (latt[[x, y]]) {
                  case 0 :
                        result1[[x, y]] = 127;
                        result2[[x, y]] = 0;
                        result3[[x, y]] = 0;
                        break;
                  case 1 :
                        result1[[x, y]] = 0;
                        result2[[x, y]] = 127;
                        result3[[x, y]] = 0;
                        break;
                  case 2 :
                        result1[[x, y]] = 0;
                        result2[[x, y]] = 0;
                        result3[[x, y]] = 127;
                        break;
                  case 3 :
                        result1[[x, y]] = 127;
                        result2[[x, y]] = 127;
                        result3[[x, y]] = 0;
                        break;
                  case 4 :
                        result1[[x, y]] = 127;
                        result2[[x, y]] = 0;
```

```
                        result3[[x, y]] = 127;
                        break;
                    case 5 :
                        result1[[x, y]] = 0;
                        result2[[x, y]] = 127;
                        result3[[x, y]] = 127;
                        break;
                    case 6 :
                        result1[[x, y]] = 127;
                        result2[[x, y]] = 127;
                        result3[[x, y]] = 127;
                        break;
                    default :
                        result1[[x, y]] = 0;
                        result2[[x, y]] = 0;
                        result3[[x, y]] = 0;
                        break;
                }
            }
        }

        Adlib.afprintf(ifile, "%c%c%c",result1,result2,result3);

        Adlib.gfclose(ifile);

    }

  }
}


void setup_random(int xsubi [] , int iseed) {

  over(id) {
    int procs = id ;
    for (int i=0; i<3; i++) {
        xsubi[i] = (procs * 4096 + 889 + iseed) % 24025;
        procs *= 3;
    }
  }
}
```

3 Performance

The most time consuming part in the kernel loop when updating the energy state of each site has been reduced by using the ghost area communication technology `writeHalo` in the HPJava implementation. This will result in much better performance compared with the HPF and F90 version.

(To be tested)