Introduction to parallel Computing

P. B. Sunil Kumar

Department of Physics

IIT Madras, Chennai 600036

www.physics.iitm.ac.in/~sunil

What is high performance computing ?

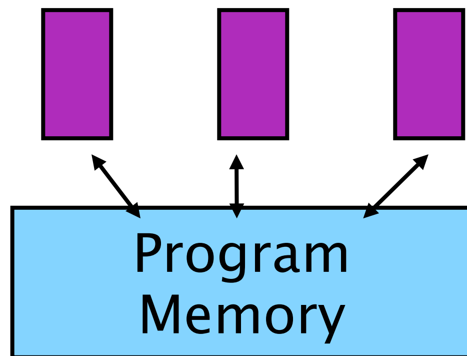
- Large volume numerical calculations : Set up facilites to run large number of codes in an efficient manner.
- Fast and efficient algorithms: Increase the speed of computing by optimizing the algorithm for a particular architecture.
- Highly scalable algorithms: Develop parallel codes that can scale linearly with number of processors .
- Modification of the computing model to allow for the use of recent advances in hardware: Cuda programming for GPU .
- Codes for complex problems: climate modeling , turbulence, protein folding, pattern and speech recognization, structural predictions ..

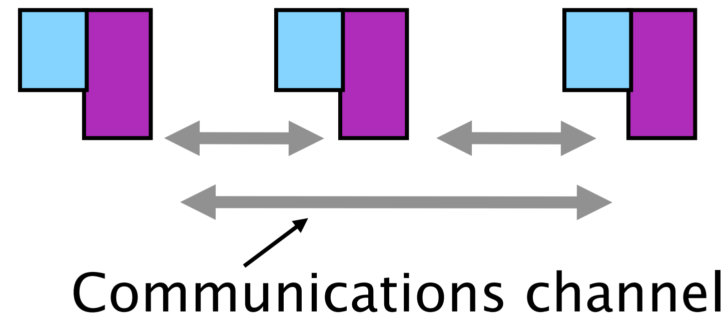
Sequential vs Parallel

- We are used to sequential programming – C, Java, C++, etc. E.g., Bubble Sort, Binary Search, Multiplication, FFT, BLAST, …
- Main idea – Specify the steps in perfect order
- Reality – We are used to parallelism a lot more than we think – as a concept; not for programming
- Methodology – Launch a set of tasks; communicate to make progress. E.g., Sorting 500 answer papers by – making 5 equal piles, have them sorted by 5 people, merge them together.

Main Paradigms: Shared vs Distributed Memory Programming

- Shared Memory – All tasks access the same memory, the same data. *pthreads*
- Distributed Memory – All memory is local. Data sharing is by explicitly transporting data from one task to another (*send-receive* pairs in MPI, e.g.)





- Tasks vs CPUs;
- SMPs vs Clusters

Simple Parallel Program – sorting numbers in a large array A

- Notionally **divide** A into **5** pieces [0..99;100..199;200..299;300..399;400..499].
- **Each part** is sorted by an **independent** sequential algorithm and **left** within its region.
- The resultant parts are merged by simply reordering among **adjacent** parts.

Work Break-down

- Parallel algorithm
- Prefer simple intuitive breakdowns
- Usually highly optimized sequential algorithms are not easily parallelizable
- Breaking work often involves some pre- or post- processing (much like divide and conquer)
- Fine vs large grain parallelism and relationship to communication

Sample OpenMP code

```
!$omp PARALLEL
!$OMP do
 do I=1,n
  y(I)=f(x(I))
Enddo
!$OMP end do
!$omp end PARALLEL
```

- ifort -openmp test.f
- export omp_NUM_THREADS=8
- ./a.out

```
#pragma omp PARALLEL for
for { I==1;I<=m;I++
  y(I)=f(x(I))
}
```

- icc -openmp test.f
- export omp_NUM_THREADS =8
- ./a.out

MPI: What do we need to think about…

- How many people are doing the work. (Degree of Parallelism)
- What is needed to begin the work. (Initialization)
- Who does what. (Work distribution)
- Access to work part. (Data/IO access)
- Whether they need info from each other to finish their own job. (Communication)
- When are they all done. (Synchronization)
- What needs to be done to collate the result.

MPI program structure

- All MPI programs must follow the same general structure. This structure is outlined as follows:

- include MPI header file
- variable declarations
- initialize the MPI environment

- ...do computation and MPI communication calls...

- close MPI communications

Hello world !!!

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {
  int myrank, size;
  MPI_Init(&argc, &argv);                    /* Initialize MPI        */
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* Get my rank
    */
  MPI_Comm_size(MPI_COMM_WORLD, &size);   /* Get the total
                                     number of processors */
  printf("Processor %d of %d: Hello World!\n", myrank, size);
  MPI_Finalize();                            /* Terminate MPI        */
 }
```

- Out put if run with "mpirun -np 4 ./a.out"
- Processor 2 of 4: Hello World!
- Processor 1 of 4: Hello World!
- Processor 3 of 4: Hello World!
- Processor 0 of 4: Hello World!

This program evaluates the trapezoidal rule estimate for an integral of F(x). In this case, F(x) is exp(x) evaluated for the interval of 0 to 1.

```
 *   This program requires the following call(s)
 *           MPI_Init,   MPI_Comm_rank,  MPI_Comm_size,  MPI_Abort,  MPI_Finalize

main (int argc, char **argv)
{ double  A = 0.0, B = 1.0;
MPI_Init(&argc, &argv);       /* Initialize MPI */
MPI_Comm_rank(MPI_COMM_WORLD, &whoami);/* Find out this processor number */
MPI_Comm_size(MPI_COMM_WORLD, &nworkers); /* Find out the number of processo
DH=(B-A)/nworkers;
if (whoami > NP)  MPI_Abort(MPI_COMM_WORLD, errcode);

T1=A+whoami*DH; T2=A+(whoami+1)*DH; H=(T2-T1)/50.0;

SUM = 0.0;
for (i = 0; i <=49; i++)   SUM = SUM+F(T1+H*i)+F(T1+H*(i+1));
TN = SUM*H;

MPI_Reduce (&TN, &res,1,MPI_DOUBLE, MPI_SUM, 1, MPI_COMM_WORLD);

if(whoami==2)printf("Integral = %10.6f\n",res);

MPI_Finalize(); }
```

An MPI program to simulate the Ising model in two dimensions.

This program demonstrate the usage of mpi grid topology and communication between grid elements.

Help from Ms. Prathyusha in the preparation of this code is acknowledged.

```
#include <math.h>
#include <mpi.h>
#include <string.h>
#define LENGTH 6        /* system+boundary layer size is LENGTH*LENGTH
#define BUFSIZE (LENGTH-2)*(LENGTH-2)       /* system size*/
#define TEMP   1.0         /* TEMP in units of interaction and k_B */
#define WARM   1000
#define MCS    1250

int    total_energy( int [][LENGTH], int [] , int []);
int    total_mag( int [][LENGTH]); int coordinates[2];

double ran3( long int *); float  fpow (float , float); float  mag_analytic(float);
/* total number of processes "nworkers", identity of each process "rank" */
int          nworkers, rank, NP;
int spin[LENGTH][LENGTH];  int nbr1[LENGTH];  int nbr2[LENGTH];

/* structure defining the variables of the grid topology */
typedef struct GRID_INFO_TYPE{
int p;
MPI_Comm comm;   MPI_Comm row_comm;  MPI_Comm col_comm;
int q;  int my_row;  int my_col;  int my_rank;
}GRID_INFO_TYPE;
```

```
void   print_config (GRID_INFO_TYPE* grid, int [][LENGTH] , int );

/* set up the grid-topology */
void Setup_grid(GRID_INFO_TYPE* grid);

void initialize(GRID_INFO_TYPE* grid, int [][LENGTH], int [] , int []);

void mcmove(GRID_INFO_TYPE* grid, int[][LENGTH] , int [] , int [] );

/* communicate the state of the spins at the boundaries to neighbors */
void boundary(GRID_INFO_TYPE* grid, int[][LENGTH] , int [] , int [] );
```

```
main (int argc, char **argv)
{
/* Allocate memory for the  structure grid */
GRID_INFO_TYPE *grid = (GRID_INFO_TYPE *)malloc(sizeof(GRID_INFO_TYPE));
   /* Initialize MPI */
MPI_Init(&argc, &argv);
   /* Find out this process number */
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   /* Find out the number of processes */
MPI_Comm_size(MPI_COMM_WORLD, &nworkers);

Setup_grid(grid);

  int itime;
  int i;
  int big_energy,E;
  int big_mag,M;
  double E_per_spin;
  double M_per_spin;
  NP=sqrt(nworkers);
  iseed=iseed*(rank+1);
    itime = 0;
    big_energy = 0;
    big_mag = 0;
```

```
/* get started */
   initialize(grid,spin, nbr1, nbr2);
         boundary(grid,spin, nbr1, nbr2);
/* warm up system */
    for (i = 1 ; i <= WARM; i++)
    {   itime = i;
        mcmove(grid,spin, nbr1, nbr2);
        boundary(grid,spin, nbr1, nbr2);  }
/* do Monte Carlo steps and collect stuff for averaging */
    for (i = (WARM + 1) ; i <= MCS; i++)
    {       itime = i;
            mcmove(grid,spin, nbr1, nbr2);
            if(i!=MCS)boundary(grid,spin, nbr1, nbr2);
              big_mag = big_mag + total_mag(spin);
              big_energy = big_energy + total_energy(spin,nbr1,nbr2); }
  printf("Mag  %f rank %d time %d\n", 1.0*big_mag/(MCS-WARM), rank, MCS-WARM);
    MPI_Reduce (&big_mag, &M,1,MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
/*process 0 adds total magentization from each process to find the net magnetization.
*/
    MPI_Reduce (&big_energy, &E,1,MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if(rank==0)
    { M_per_spin = (float)M/((MCS - WARM)*(nworkers*(LENGTH-2)*(LENGTH-2)));
     E_per_spin = (float)E/((MCS - WARM)*nworkers*((LENGTH-2)*(LENGTH-2)));}
   print_config(grid, spin, itime );
  MPI_Finalize();  /* exit all MPI functions */ }
```

```
void Setup_grid(GRID_INFO_TYPE* grid){
int dimensions[2];
int periods[2];
int varying_coords[2];
MPI_Comm   localname;

MPI_Comm_size(MPI_COMM_WORLD, (&grid->p)); /* get the total number of
processes */
grid->q=(int)sqrt((double) grid->p);  /* square grid  */
dimensions[0]=dimensions[1]=grid->q;   /* dimension of the grid */
periods[0]=periods[1]=1;  /*  periodic if 1 and non-periodic if 0 */

MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods, 1, &(grid->comm)
create a grid communicator from the "Comm_world" , having dimension 2, reorder
process to processor mapping */

MPI_Comm_rank(grid->comm, &(grid->my_rank));
MPI_Cart_coords(grid->comm, grid->my_rank,2,coordinates);
grid->my_row=coordinates[0];
grid->my_col=coordinates[1];
varying_coords[0]=0;varying_coords[1]=1;
MPI_Cart_sub(grid->comm,varying_coords,&(grid->row_comm));
varying_coords[0]=1;varying_coords[1]=0;
MPI_Cart_sub(grid->comm,varying_coords,&(grid->col_comm));
}
```

```
void initialize(GRID_INFO_TYPE* grid,int spin[][LENGTH], int nbr1[], int nbr2[])
{
 int i, ix, iy,m,n,mt,nt,tag=50; float sd;

  char message[100];

     for (iy = 0 ; iy <LENGTH; iy++) /* start with random spins */
     for (ix = 0 ; ix <LENGTH; ix++)
     { sd=ran3(&iseed);
       if( sd>0.5 )spin[ix][iy] = 1;
       if( sd <= 0.5 )spin[ix][iy] = -1; }

     for (i = 1 ; i < LENGTH-1 ; i++)  /* set up neighbor list */
     {
       nbr1[i] = i - 1;
       nbr2[i] = i + 1;
      }
}
```

```
void print_config(GRID_INFO_TYPE *grid, int spin[][LENGTH] , int itime )
   int ix , iy, i,buf[(LENGTH-2)],buf1[LENGTH-2],tag=147,n,mp,np;
   char fname[30],message[50];
   float cx[1];
   FILE* fpr;
   MPI_Status status;
   if(grid->my_rank!=0)/* if rank is not zero pack all spins into a 1-d array */
   {  i=0;
        for (iy =1  ; iy < LENGTH-1; iy++)    {
        for (ix = 1 ; ix < LENGTH-1; ix++)
            {  buf[i]=spin[iy][ix];  i++; } }
         MPI_Send(buf,BUFSIZE,MPI_INT,0,tag,MPI_COMM_WORLD); /* send it to
process 0 */ }
   else  /* if the rank is zero */
  { fpr=fopen("config","w");  /* open the file */
     for (iy =1  ; iy < LENGTH-1; iy++) /* print the own configuration*/
     { for (ix = 1 ; ix < LENGTH-1; ix++)
       {   fprintf(fpr,"%d \t", spin[iy][ix]); } }
     for (n=1 ; n<nworkers;n++)
     { MPI_Recv(buf1,BUFSIZE,
MPI_INT,n,tag,MPI_COMM_WORLD,MPI_STATUS_IGNORE); /* receive from all other
processes */
        for(i=0; i<(LENGTH-2);i++)
        { for (ix = i*(LENGTH-2) ; ix < (i+1)*(LENGTH-2); ix++)
           { fprintf(fpr,"%d \t", buf1[ix]); }  } }   fclose(fpr);} }
```

```
void mcmove(GRID_INFO_TYPE* grid, int spin[][LENGTH], int nbr1[] , int nbr2[])
{
  int *U1,*D1,*R1,*L1;
/* arrays to recieve the spin configuration from neighbors */
     U1=(int *)malloc(LENGTH*sizeof(int));
     D1=(int *)malloc(LENGTH*sizeof(int));
     R1=(int *)malloc(LENGTH*sizeof(int));
     L1=(int *)malloc(LENGTH*sizeof(int));
   MPI_Status status;
  MPI_Cart_shift(grid->comm,0,-1,&m,&n);
  MPI_Recv(U1,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD,&status); /* receive U1
buffer with dimension LENGTH of type MPI_INT from process "n" with "tag" */
  MPI_Cart_shift(grid->comm,0,1,&m,&n);
  MPI_Recv(D1,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD,&status);
  MPI_Cart_shift(grid->comm,1,-1,&m,&n);
  MPI_Recv(L1,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD,&status);
  MPI_Cart_shift(grid->comm,1,1,&m,&n);
  MPI_Recv(R1,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD,&status);
  prob1 = exp(-8.0/TEMP);
  prob2 = exp(-4.0/TEMP);
  for (i = 1 ; i < LENGTH-1; i++)
  {  spin[i][0]=U1[i]; spin[i][LENGTH-1]=D1[i]; spin[0][i]=R1[i];  spin[LENGTH-1][i]=L1[i];  }
            free(U1); free(D1); free(L1); free(R1);
```

DO THE MC SPIN FLIP MOVES

```
void boundary(GRID_INFO_TYPE* grid, int spin[][LENGTH], int nbr1[] , int nbr2[])
{     U2=(int *)malloc(LENGTH*sizeof(int));     D2=(int *)malloc(LENGTH*sizeof(int));
    R2=(int *)malloc(LENGTH*sizeof(int));  L2=(int *)malloc(LENGTH*sizeof(int));
/* put  the spins at the left, right, up and down boundaries into separate arrays */
    for (i = 1 ; i < LENGTH-1 ; i++)
 {  U2[i]=spin[i][1];  D2[i]=spin[i][LENGTH-2];  L2[i]=spin[1][i];  R2[i]=spin[LENGTH-2][i]; }

/*send the boundary arrays to appropriate neighbor process
Remeber 0 is up and down and  1 is left and right.  */

      MPI_Cart_shift(grid->comm,0,-1,&m,&n); /* find the neighbor process down the
current one */
      MPI_Send(U2,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD); /* send U2 buffer
with dimension LENGTH of type MPI_INT to  process "n" with "tag" */
      MPI_Cart_shift(grid->comm,0,1,&m,&n);
      MPI_Send(D2,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD);
      MPI_Cart_shift(grid->comm,1,1,&m,&n);
      MPI_Send(R2,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD);
      MPI_Cart_shift(grid->comm,1,-1,&m,&n);
      MPI_Send(L2,LENGTH, MPI_INT,n,tag,MPI_COMM_WORLD);

           free(U2);             free(D2);       free(L2);   free(R2);
}
```

Introduction to Parallel Computing. Michael Skuhersky vex@mit.edu. What is Parallel Computing? â— Wikipedia says: â€œParallel computing is a form of computation in which many calculations are carried out simultaneouslyâ€. â— Speed measured in FLOPS. What is Parallel Computing? (cont.) How can this be useful?Â Main Granularity Paradigms. â— Granularity: the ratio of computation to communication. â— 3 approaches to parallelism, depending on what kind of problem you need to solve. Embarrassingly Parallel. â— No effort required to separate tasks â— Tasks do not depend on, or communicate. with, each other. â— Examples: Mandlebrot, Folding@home, Password brute-forcing, Bitcoin Mining! Introduction to Parallel Computing. Table of Contents. Abstract. Overview. What is Parallel Computing? Why Use Parallel Computing? Concepts and Terminology. von Neumann Computer Architecture.Â Overview. What is Parallel Computing? Traditionally, software has been written for serial computation: To be run on a single computer having a single Central Processing Unit (CPU) â€¢ Parallel computing: use of multiple processors or computers working together on a common task. â€“ Each processor works on its section of the problem â€“ Processors can exchange information. Grid of Problem to be solved. y. CPU #1 works on this area of the problem exchange.

Parallel computing is a type of computation where many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time. There are several different forms of parallel computing: bit-level, instruction-level, data, and task parallelism. Parallelism has long been employed in high-performance computing, but has gained broader interest due to the physical constraints preventing frequency scaling. As An Introduction to Parallel Computing. Edgar Gabriel Department of Computer Science. University of Houston gabriel@cs.uh.edu. Short course on Parallel Computing Edgar Gabriel. Why Parallel Computing? â€¢ To solve larger problems â€“ many applications need significantly more memory than a regular PC can provide/handle. â€¢ To solve problems faster â€“ despite of many advances in computer hardware technology, many applications are running slower and slower â€¢ e.g. databases having to handle more and more data â€¢ e.g. large simulations working on even more accurate solutions. Short course on Parallel Compu... â€¢ Parallel computing: use of multiple processors or computers working together on a common task. â€“ Each processor works on its section of the problem â€“ Processors can exchange information. Grid of Problem to be solved. y. CPU #1 works on this area of the problem exchange. Introduction to Parallel Computing. Addison Wesley, ISBN: 0-201-64865-2, 2003. Ananth Grama, Purdue University, W. Lafayette, IN 47906 (ayg@cs.purdue.edu) Anshul Gupta, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598 (anshul@watson.ibm.com) George Karypis, University of Minnesota, Minneapolis, MN 55455 (karypis@cs.umn.edu) Vipin Kumar, University of Minnesota, Minneapolis, MN 55455 (kumar@cs.umn.edu) Here are the transparencies accompanying each of the chapters. Introduction to Parallel Computing. Author: Blaise Barney, Lawrence Livermore National Laboratory. UCRL-MI-133316.Â In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem: A problem is broken into discrete parts that can be solved concurrently. Each part is further broken down to a series of instructions.