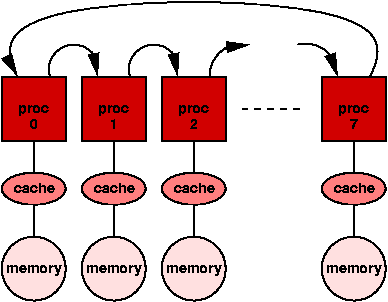
### The roundRobin program

* This is our second parallel program. It illustrates basic interprocessor communication. According to our SPMD model it is also a single program (SP) that is executed by every processor.
* Processor 0 composes a message and sends it to processor 1. Processor 1 receives the message and sends it on to 2. Processor 2 receives it and sends it to 3, and so forth. Finally the last processor sends it back to 0 who then compares it with the original.



In C our roundRobin program is:

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#define ARRAY\_SIZE 10000 /\* message size \*/

int main( int argc, char \*argv[] ) {

/\* Processor 0 packs a send buffer and sends it off to processor 1. \*/

/\* Processor 1 receives this package and sends it off to \*/

/\* processor 2. This continues until processor 0 finally receives the \*/

/\* package and compares it with the original. \*/

/\* \*/

/\* Remember, it is convenient to reference the typical processor in the \*/

/\* first person, using "I" and "me" and "my." In other words, \*/

/\* read through this program as if you were one of the processors. \*/

/\* My ID number and the total number of processors: \*/

int myrank, numProcs;

/\* Variables for send destination ID and receive source ID: \*/

int dest, src;

/\* Let's not use tagging. We're sending just one message. Set it to 1: \*/

int tag = 1;

/\* Allocate the array msg[] which will serve as both send and \*/

/\* receive buffer: \*/

double msg[ARRAY\_SIZE];

/\* But if I am processor 0, I want a separate receive buffer for \*/

/\* comparing with the original: \*/

double check[ARRAY\_SIZE];

/\* A couple of integer indices: \*/

int i,proc;

/\* And finally I need to declare an MPI status variable \*/

/\* for checking the status of the MPI\_Recv function: \*/

MPI\_Status status;

/\* Initialize the MPI API: \*/

**MPI\_Init(&argc, &argv);**

/\* Request my ID number: \*/

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);**

/\* I'll also ask how many other processors are out there: \*/

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcs);**

/\* Okay. The preparations have been made. \*/

/\* If I'm processor 0, then I should print the number of \*/

/\* processors in this run. \*/

if (myrank == 0) {

printf( "The number of processors in this run is %d.\n", numProcs );

}

/\* If I'm processor 0, then I better get busy and pack my send buffer: \*/

if (myrank == 0) {

for (i = 0; i < ARRAY\_SIZE; i++) {

/\* I'll assign the array index as the value of array element \*/

msg[i] = i;

/\* Also I better prepare my receive buffer: \*/

check[i] = 0;

}

}

/\* Begin the round-robin: \*/

for (proc = 0; proc < numProcs - 1; proc++) {

/\* If I'm processor proc, then I want to send \*/

/\* to processor proc + 1: \*/

if (myrank == proc) {

dest = proc + 1;

**MPI\_Send( msg,**

**ARRAY\_SIZE,**

**MPI\_DOUBLE,**

**dest,**

**tag,**

**MPI\_COMM\_WORLD );**

printf( "Processor %d has sent the message.\n", myrank );

}

/\* Else if I'm processor proc + 1, then I want to \*/

/\* receive from processor proc: \*/

else if (myrank == proc + 1) {

src = proc;

**MPI\_Recv( msg,**

**ARRAY\_SIZE,**

**MPI\_DOUBLE,**

**src,**

**tag,**

**MPI\_COMM\_WORLD,**

**&status );**

printf( "Processor %d has received the message.\n", myrank );\

}

}

/\* And if I happen to be the last processor \*/

/\* then I want to send to processor 0: \*/

if (myrank == numProcs - 1) {

dest = 0;

**MPI\_Send( msg,**

**ARRAY\_SIZE,**

**MPI\_DOUBLE,**

**dest,**

**tag,**

**MPI\_COMM\_WORLD );**

printf( "Processor %d has sent the message.\n", myrank );

}

/\* Finally if I'm processor 0 then I want to receive \*/

/\* from the last processor and compare this with what \*/

/\* I'd originally sent: \*/

if (myrank == 0) {

src = numProcs - 1;

**MPI\_Recv( check,**

**ARRAY\_SIZE,**

**MPI\_DOUBLE,**

**src,**

**tag,**

**MPI\_COMM\_WORLD,**

**&status);**

for (i = 0; i < ARRAY\_SIZE; i++) {

if (msg[i] != check[i]) {

printf( "error in element %d of the message.\n", i );

}

}

printf( "Processor %d has received and checked the message. "

"Program terminating normally.\n", myrank );

}

/\* Close the MIP API: \*/

**MPI\_Finalize();**

return 0;

}

You can expect to see something like this:

The number of processors in this run is 4.

Processor 0 has sent the message.

Processor 0 has received and checked the message. Program terminating normally.

Processor 2 has received the message.

Processor 2 has sent the message.

. . .

[fela@epsilon ~]$ mpicc roundRobin.c -o roundRobin

[fela@epsilon ~]$ ls

1 hello hello.c implementacja roundRobin

a.out helloc hello.cc nodes1 roundRobin.c

[fela@epsilon ~]$ mpirun -np 4 roundRobin

The number of processors in this run is 4.

Processor 0 has sent the message.

Processor 1 has received the message.

Processor 1 has sent the message.

Processor 2 has received the message.

Processor 2 has sent the message.

Processor 3 has received the message.

Processor 3 has sent the message.

Processor 0 has received and checked the message. Program terminating normally.

### Serial program to compute PI: serialPi

The serial version of our program, written in C, is displayed below:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

const double pi = 3.1415926535897932385;

int main( int argc, char \*argv[] ) {

/\* Arguments required for executing serialPi: \*/

/\* 0 -> serialPi \*/

/\* 1 -> numberOfIntervals \*/

int numberOfIntervals, interval;

double intervalWidth, intervalMidPoint, area = 0.0;

/\* The number of intervals is a command line argument. \*/

numberOfIntervals = atoi( argv[1] );

/\* Compute the interval width. \*/

intervalWidth = 1.0 / numberOfIntervals;

/\* Now compute the area. \*/

for ( interval = 0; interval < numberOfIntervals; interval++ ) {

intervalMidPoint = (interval + 0.5) \* intervalWidth;

area += 4.0 / ( 1.0 + intervalMidPoint\*intervalMidPoint );

}

area \*= intervalWidth;

/\* Print the results. \*/

printf( "The computed value of the integral is %.15f\n", area );

printf( "The exact value of the integral is %.15f\n", pi );

return 0;

}

The parallel version of our program, written in C, is shown below:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <mpi.h>

const double pi = 3.1415926535897932385;

int main( int argc, char \*argv[] ) {

/\* This is a parallelization of serialPi.c. \*/

/\* The for-loop is partitioned among the processors. \*/

/\* Function MPI\_Reduce sums the processors' results for processor 0. \*/

/\* Processor 0 outputs. \*/

/\* \*/

/\* Remember, it is convenient to reference the typical processor in the \*/

/\* first person, using "I" and "me" and "my." In other words, \*/

/\* read through this program as if you were one of the processors. \*/

/\* Arguments required for executing argv[0]: \*/

/\* 0 -> serialPi \*/

/\* 1 -> numberOfIntervals \*/

/\* My ID number and the total number of processors: \*/

int myrank, numProcs;

/\* Variables for the computation. \*/

int numberOfIntervals, interval;

double intervalWidth, intervalMidPoint, totalArea, myArea = 0.0;

/\* For completeness I'll declare an MPI status variable, \*/

/\* although I don't plan to use it. \*/

**MPI\_Status status;**

/\* Initialize the MPI API: \*/

**MPI\_Init(&argc, &argv);**

/\* Request my ID number: \*/

**MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);**

/\* I'll also ask how many other processors are out there: \*/

**MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcs);**

/\* Okay. The preparations have been made. \*/

/\* The number of intervals is a command line argument. \*/

numberOfIntervals = atoi( argv[1] );

/\* Compute the interval width. \*/

intervalWidth = 1.0 / numberOfIntervals;

/\* Now I'll compute my area. \*/

for ( interval = myrank; interval < numberOfIntervals; interval += numProcs ) {

intervalMidPoint = (interval + 0.5) \* intervalWidth;

myArea += 4.0 / ( 1.0 + intervalMidPoint\*intervalMidPoint );

}

/\* Okay. Now I'll submit my results to the MPI API. \*/

/\* If I'm processor 0, I'll receive the total area. \*/

/\* Otherwise, I'm done. \*/

**MPI\_Reduce( &myArea,**

**&totalArea,**

**1,**

**MPI\_DOUBLE,**

**MPI\_SUM,**

**0,**

**MPI\_COMM\_WORLD );**

/\* If I'm processor 0, I need to normalize the total area and output. \*/

if (myrank == 0) {

totalArea \*= intervalWidth;

printf( "The computed value of the integral is %.15f\n", totalArea );

printf( "The exact value of the integral is %.15f\n", pi );

}

/\* Close the MIP API: \*/

**MPI\_Finalize();**

return 0;

}

Example 1

#include <math.h>

**#include "mpi.h"**

int main(int argc, char \*argv[])

{

int n, rank, size, i;

double PI25DT = 3.141592653589793238462643;

double mypi, pi, h, sum, x;

**MPI::Init(argc, argv);**

**size = MPI::COMM\_WORLD.Get\_size();**

**rank = MPI::COMM\_WORLD.Get\_rank();**

while (1) {

if (rank == 0) {

cout << "Enter the number of intervals: (0 quits)"

<< endl;

cin >> n;

}

**MPI::COMM\_WORLD.Bcast(&n, 1, MPI::INT, 0);**

if (n==0)

break;

else {

h = 1.0 / (double) n;

sum = 0.0;

for (i = rank + 1; i <= n; i += size) {

x = h \* ((double)i - 0.5);

sum += (4.0 / (1.0 + x\*x));

}

mypi = h \* sum;

**MPI::COMM\_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,**

**MPI::SUM, 0);**

if (rank == 0)

cout << "pi is approximately " << pi

<< ", Error is " << fabs(pi - PI25DT)

<< endl;

}

}

**MPI::Finalize();**

return 0;

}

**Example 2**

/\* compute pi using Monte Carlo method \*/

#include <math.h>

#include "mpi.h"

#include "mpe.h"

#define CHUNKSIZE 1000

/\* We'd like a value that gives the maximum value returned by the function

random, but no such value is \*portable\*. RAND\_MAX is available on many

systems but is not always the correct value for random (it isn't for

Solaris). The value ((unsigned(1)<<31)-1) is common but not guaranteed \*/

#define INT\_MAX 1000000000

/\* message tags \*/

#define REQUEST 1

#define REPLY 2

int main( int argc, char \*argv[] )

{

int iter;

int in, out, i, iters, max, ix, iy, ranks[1], done, temp;

double x, y, Pi, error, epsilon;

int numprocs, myid, server, totalin, totalout, workerid;

int rands[CHUNKSIZE], request;

MPI\_Comm world, workers;

MPI\_Group world\_group, worker\_group;

MPI\_Status status;

**MPI\_Init(&argc,&argv);**

world = MPI\_COMM\_WORLD;

**MPI\_Comm\_size(world,&numprocs);**

**MPI\_Comm\_rank(world,&myid);**

server = numprocs-1; /\* last proc is server \*/

if (myid == 0)

sscanf( argv[1], "%lf", &epsilon );

**MPI\_Bcast( &epsilon, 1, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD );**

**MPI\_Comm\_group( world, &world\_group );**

ranks[0] = server;

**MPI\_Group\_excl( world\_group, 1, ranks, &worker\_group );**

**MPI\_Comm\_create( world, worker\_group, &workers );**

**MPI\_Group\_free(&worker\_group);**

if (myid == server) { /\* I am the rand server \*/

do {

**MPI\_Recv(&request, 1, MPI\_INT, MPI\_ANY\_SOURCE, REQUEST,**

**world, &status);**

if (request) {

for (i = 0; i < CHUNKSIZE; ) {

rands[i] = random();

if (rands[i] <= INT\_MAX) i++;

}

**MPI\_Send(rands, CHUNKSIZE, MPI\_INT,**

**status.MPI\_SOURCE, REPLY, world);**

}

}

while( request>0 );

}

else { /\* I am a worker process \*/

request = 1;

done = in = out = 0;

max = INT\_MAX; /\* max int, for normalization \*/

**MPI\_Send( &request, 1, MPI\_INT, server, REQUEST, world );**

**MPI\_Comm\_rank( workers, &workerid );**

iter = 0;

while (!done) {

iter++;

request = 1;

**MPI\_Recv( rands, CHUNKSIZE, MPI\_INT, server, REPLY,**

**world, &status );**

for (i=0; i<CHUNKSIZE-1; ) {

x = (((double) rands[i++])/max) \* 2 - 1;

y = (((double) rands[i++])/max) \* 2 - 1;

if (x\*x + y\*y < 1.0)

in++;

else

out++;

}

**MPI\_Allreduce(&in, &totalin, 1, MPI\_INT, MPI\_SUM,**

**workers);**

**MPI\_Allreduce(&out, &totalout, 1, MPI\_INT, MPI\_SUM,**

**workers);**

Pi = (4.0\*totalin)/(totalin + totalout);

error = fabs( Pi-3.141592653589793238462643);

done = (error < epsilon || (totalin+totalout) > 1000000);

request = (done) ? 0 : 1;

if (myid == 0) {

printf( "\rpi = %23.20f", Pi );

**MPI\_Send( &request, 1, MPI\_INT, server, REQUEST,**

**world );**

}

else {

if (request)

**MPI\_Send(&request, 1, MPI\_INT, server, REQUEST,**

**world);**

}

}

**MPI\_Comm\_free(&workers);**

}

if (myid == 0) {

printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",

totalin+totalout, totalin, totalout );

getchar();

}

**MPI\_Finalize();**

}

**Example**

Broadcast 100 ints from process 0 to every process in the group.

MPI\_Comm comm;

int array[100];

int root=0;

...

MPI\_Bcast( array, 100, MPI\_INT, root, comm);

**Scatter**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | A B C D E |  |  |  |  |  |  |
| After operation: | | | | | | | | |
| A |  | A B C D E  B |  | C |  | D |  | E |

**Example**

The reverse of Example [Examples using MPI\_GATHER, MPI\_GATHERV](http://www.mpi-forum.org/docs/mpi-11-html/node70.html#Node70) . Scatter sets of 100 ints from the root to each process in the group. See figure [8](http://www.mpi-forum.org/docs/mpi-11-html/node72.html#Figure8) .

MPI\_Comm comm;

int gsize,\*sendbuf;

int root, rbuf[100];

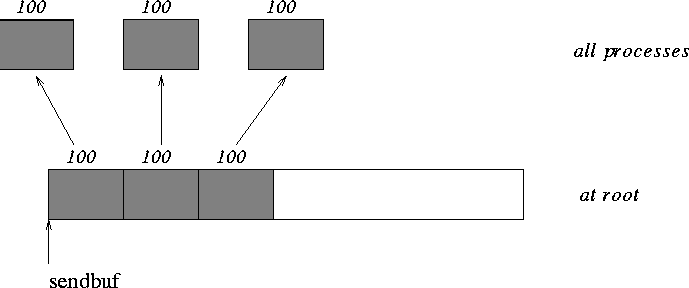
...

MPI\_Comm\_size( comm, &gsize);

sendbuf = (int \*)malloc(gsize\*100\*sizeof(int));

...

MPI\_Scatter( sendbuf, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm);



**Figure 8:** The root process scatters sets of 100 ints to each process in the group.

**Gather**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A |  | B |  | C |  | D |  | E |
| After operation: | | | | | | | | |
| A |  | A B C D E  B |  | C |  | D |  | E |

**Example**

Gather 100 ints from every process in group to root.

MPI\_Comm comm;

int gsize,sendarray[100];

int root, \*rbuf;

...

MPI\_Comm\_size( comm, &gsize);

rbuf = (int \*)malloc(gsize\*100\*sizeof(int));

MPI\_Gather( sendarray, 100, MPI\_INT, rbuf, 100, MPI\_INT, root, comm);

**Example**

Now have each process send 100 ints to root, but place each set (of 100) stride ints apart at receiving end. Use MPI\_GATHERV and the displs argument to achieve this effect. Assume . See figure [4](http://www.mpi-forum.org/docs/mpi-11-html/node70.html#Figure4) .

MPI\_Comm comm;

int gsize,sendarray[100];

int root, \*rbuf, stride;

int \*displs,i,\*rcounts;

...

MPI\_Comm\_size( comm, &gsize);

rbuf = (int \*)malloc(gsize\*stride\*sizeof(int));

displs = (int \*)malloc(gsize\*sizeof(int));

rcounts = (int \*)malloc(gsize\*sizeof(int));

for (i=0; i<gsize; ++i) {

displs[i] = i\*stride;

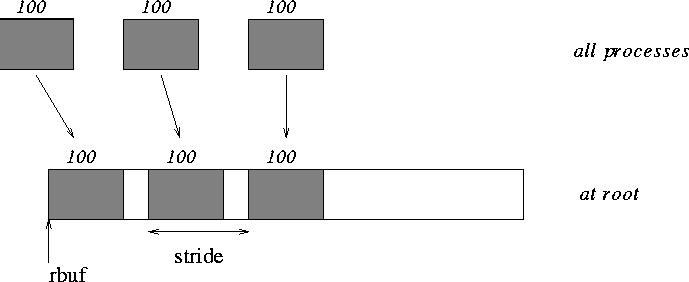
rcounts[i] = 100;

}

MPI\_Gatherv( sendarray, 100, MPI\_INT, rbuf, rcounts, displs, MPI\_INT,

root, comm);

Note that the program is erroneous if *stride < 100*.



**Figure 4:** The root process gathers 100 int’s from each process in the group, each set is placed stride ints apart.

**Example**

Each process has an array of 30 doubles, in C. For each of the 30 locations, compute the value and rank of the process containing the largest value.

...

/\* each process has an array of 30 double: ain[30]

\*/

double ain[30], aout[30];

int ind[30];

struct {

double val;

int rank;

} in[30], out[30];

int i, myrank, root;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);

for (i=0; i<30; ++i) {

in[i].val = ain[i];

in[i].rank = myrank;

}

MPI\_Reduce( in, out, 30, MPI\_DOUBLE\_INT, MPI\_MAXLOC, root, comm );

/\* At this point, the answer resides on process root

\*/

if (myrank == root) {

/\* read ranks out

\*/

for (i=0; i<30; ++i) {

aout[i] = out[i].val;

ind[i] = out[i].rank;

}

}

### Example of User-defined Reduce

**Example** Compute the product of an array of complex numbers, in C.

typedef struct {

double real,imag;

} Complex;

/\* the user-defined function

\*/

void myProd( Complex \*in, Complex \*inout, int \*len, MPI\_Datatype \*dptr )

{

int i;

Complex c;

for (i=0; i< \*len; ++i) {

c.real = inout->real\*in->real -

inout->imag\*in->imag;

c.imag = inout->real\*in->imag +

inout->imag\*in->real;

\*inout = c;

in++; inout++;

}

}

/\* and, to call it...

\*/

...

/\* each process has an array of 100 Complexes

\*/

Complex a[100], answer[100];

MPI\_Op myOp;

MPI\_Datatype ctype;

/\* explain to MPI how type Complex is defined

\*/

MPI\_Type\_contiguous( 2, MPI\_DOUBLE, &ctype );

MPI\_Type\_commit( &ctype );

/\* create the complex-product user-op

\*/

MPI\_Op\_create( myProd, True, &myOp );

MPI\_Reduce( a, answer, 100, ctype, myOp, root, comm );

/\* At this point, the answer, which consists of 100 Complexes,

\* resides on process root

\*/

# The Two-dimensional Diffusion Equation

# http://www.iu.edu/~rac/hpc/mpi\_tutorial/index.html

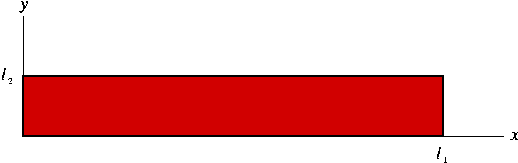
### The equation

The next example is more interesting than parallelPi. Our objective is to solve a two-dimensional diffusion PDE:



where u(x,y,t) is the temperature of the surface being modeled, and D is the so-called diffusion coefficient.

We will choose as the surface a rectangular plate:



### The exact solution of the diffusion equation

It will be comforting as we test our code to know an exact solution of the diffusion equation.

By examination of the diffusion equation, one may readily write down an exact solution:

simpleSoln

which is consistent with Dirichlet boundary conditions of u = 0 along the edges of the surface, and with the initial condition:

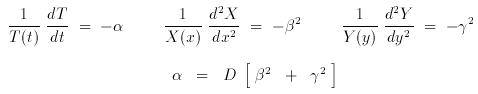
simpInitCond

However, it is easier to interpret numerical results for a uniform initial condition. The entire plate is at the same initial temperature, and at time t = 0 is immersed into a zero-temperature bath. To obtain an exact solution consistent with a uniform initial condition we use the method of separation of variables. Again, the Dirichlet boundary conditions we choose is temperature u equals 0 along the edges of the surface.

We write the solution as a product of three single-variable functions: u(x,y,t) = X(x) Y(y) T(t). The diffusion equation then looks like this:

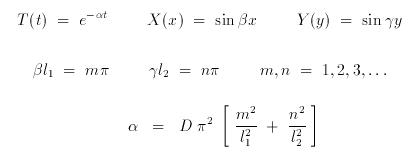
sepDiffEqn

If this relationship is to hold for arbitrary (x,y,z), then each term must be constant. This leads to three single-variable ordinary differential equations plus a relationship for the constants:



We chose the sign and form of these constants to make it easy to satisfy the thermodynamic law of entropy coupled with our Dirichlet boundary conditions. The temperature of the surface must vanish at the edges and decrease with time.

The solution of the ODE for T is a decreasing exponential function. To satisfy the boundary conditions the solutions of the X and Y ODEs are sine functions:



Therefore, the general solution of the diffusion equation for the rectangular plate given the zero-temperature boundary conditions is:

genSoln

where the expansion coefficients A are to be determined by the initial conditions, the temperature of the rectangular plate at t = 0, u(x,y,0) = f(x,y).

Using the orthogonality of the sine function:

orthog

We obtain an expression for the expansion coefficients:

expCoeffs

Now let's assume the simplist initial condition, a uniform temperature on the plate (except for the boundary of course), f(x,y) = Uo. Then it is easy to show that:

A

And at last the exact solution for u = 0 boundary conditions and given a uniform initial temperature Uo is:

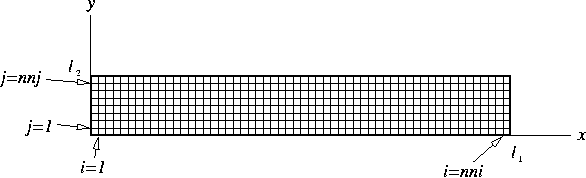
soln

This is of course an infinite sum, but with an exponential term that decreases with increasing indices m,n. In choosing a cutoff for m,n be aware that for small time t many terms may be needed to achieve satisfactory accuracy.

# Numerical Solution of the Diffusion Equation

### The grid

We superimpose a computational grid upon the plate we are modeling:



which shows the numbering of the x and y grid indices i and j, respectively. The number of interior grid points in the x-direction is nni, in the y-direction nnj. The boundaries are i = 0, i = nni + 1, j = 0, and j = nnj + 1.

Clearly the x-direction and y-direction step sizes are:

gridStepSize

We'll use a friendly notation for the temperature:

unij

where we've introduced the time index n and the time step size dt.

### The finite-difference equation (FDE)

We'll use the Forward Time Centered Space (FTCS) finite-difference approximation to the diffusion equation:

finDiff

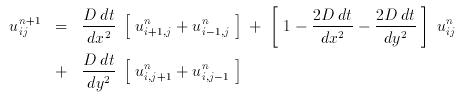
where i = 1, 2, ... , nni and j = 1, 2, ... , nnj. Also n = 0, 1, 2, ... .

In practice you will not want to use the unsophisticated FTCS representation because of its instability. The instability is avoided, however, if the time step size dt obeys the so-called Courant condition:

courant

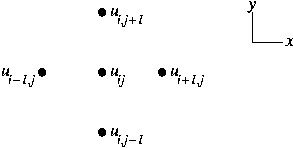
The disadvantage of this condition is that for complicated geometries for which dx and dy must be small, dt becomes very small, and many iterations are required to evolve the system though time.

The finite-difference equation may be rearranged:



Using the temperature at time step n, the temperature at time step n + 1 is computed.

Later when considering parallelization we will be especially interested in the stencil. The stencil is the set of grid points needed to compute the single ij grid point at the next time step. For the above finite-difference representation of the two-dimensional diffusion equation, we have a 5-point stencil that must be computed to obtain u\_ij at the next time step:



For computing u\_ij adjacent to a boundary, one of these stencil points will be a boundary point.

# Serial Numerical Solution of the Diffusion Equation

### The serialDiffusion program

As we'd mentioned earlier, if you are beginning a parallel project from scratch, writing the serial code is typically the first step.

In C, our serialDiffusion program is shown below:

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

const double pi = 3.1415926535897932385;

int main( int argc, char \*argv[] )

{

/\* Arguments required for executing argv[0]: \*/

/\* 0 -> serialDiffusion \*/

/\* 1 -> nni \*/

/\* 2 -> nnj \*/

/\* 3 -> numTimeSteps \*/

double \*\*uo; /\* solution at previous time step \*/

double \*\*u; /\* solution at current time step \*/

double mu, nu, c; /\* parameters of the FDE \*/

double ell1,ell2; /\* x,y grid dimensions, 0 <= x <= ell1, etc. \*/

double dx,dy; /\* x,y grid spacing, or spacial step size \*/

int nni,nnj; /\* x,y number of interior grid points; excl. bdy. pts.\*/

int numTimeSteps; /\* when to end simulation \*/

double dt; /\* time step size \*/

double dcoeff; /\* diffusion coefficient D \*/

double u\_0; /\* uniform initial conditions \*/

double actual; /\* the true solution \*/

int i,j,m,n,timeStep;

/\* Get the number of interior grid points and \*/

/\* number of time steps from the command line. \*/

nni = atoi( argv[1] );

nnj = atoi( argv[2] );

numTimeSteps = atoi( argv[3] );

/\* Allocate memory for the solution arrays; include boundaries. \*/

/\* As declared above, u is a pointer to a pointer to a double. \*/

/\* Allocate memory so u points to the first element of an \*/

/\* (nni+2)-long array of pointers to doubles. Likewise for uo. \*/

u = (double \*\*) malloc( (size\_t) ( (nni+2) \* sizeof(double\*) ) );

uo = (double \*\*) malloc( (size\_t) ( (nni+2) \* sizeof(double\*) ) );

/\* u[0] is a pointer to a double. \*/

/\* Allocate memory so u[0] points to the first element of an \*/

/\* (nni+2)\*(nnj+2)-long array of doubles. Likewise for uo[0]. \*/

u[0] = (double \*) malloc( (size\_t) ( ((nni+2)\*(nnj+2)) \* sizeof(double) ) );

uo[0] = (double \*) malloc( (size\_t) ( ((nni+2)\*(nnj+2)) \* sizeof(double) ) );

/\* Now set pointer u[1] to point to \*(u[0] + (nnj+2)). \*/

/\* And set pointer u[2] to point to \*(u[0] + 2\*(nnj+2)) \*/

/\* or \*(u[1] + (nnj+2)). \*/

/\* And so forth. Likewise for uo[1], uo[2], etc. \*/

for ( i = 1; i < nni+2; i++ )

{

u[i] = u[i-1] + (nnj+2);

uo[i] = uo[i-1] + (nnj+2);

}

/\* We have now dynamically allocated memory for the arrays \*/

/\* u[nni+2][nnj+2] and uo[nni+2][nnj+2] which are large \*/

/\* enough to include the boundary points. \*/

/\* Set the diffusion coefficient, grid dimensions, \*/

/\* and the uniform initial conditions. \*/

dcoeff = 1.0;

ell1 = 7.0;

ell2 = 1.0;

u\_0 = 1.0;

/\* Compute dx and dy, and set dt so that the Courant condition \*/

/\* is satisfied. \*/

dx = ell1 / ( nni + 1 );

dy = ell2 / ( nnj + 1 );

dt = 0.49 / ( dcoeff\*( 1.0/(dx\*dx) + 1.0/(dy\*dy) ) );

/\* Set uniform initial conditions. \*/

for ( i = 1; i <= nni; i++ )

{

for ( j = 1; j <= nnj; j++ )

{

u[i][j] = u\_0;

}

}

/\* Set boundary conditions. \*/

/\* Assume Dirichlet boundary conditions equal to 0. \*/

/\* First the y = 0 boundary. \*/

for ( i = 0; i <= nni+1; i++ )

{

u[i][0] = 0.0;

}

/\* Second the y = ell2 boundary. \*/

for ( i = 0; i <= nni+1; i++ )

{

u[i][nnj+1] = 0.0;

}

/\* Third the x = 0 boundary. \*/

for ( j = 1; j <= nnj; j++ )

{

u[0][j] = 0.0;

}

/\* Finally the x = ell1 boundary. \*/

for ( j = 1; j <= nnj; j++ )

{

u[nni+1][j] = 0.0;

}

/\* Solve the 2-D diffusion equation by the explicit FTCS method. \*/

/\* Pre-compute the FDE coefficients. \*/

mu = dcoeff\*dt / (dx\*dx);

nu = dcoeff\*dt / (dy\*dy);

c = 1.0 - 2.0\*( mu + nu );

for ( timeStep = 1; timeStep <= numTimeSteps; timeStep++ )

{

/\* Update previous time step data. \*/

for ( i = 0; i <= nni+1; i++ )

{

for ( j = 0; j <= nnj+1; j++ )

{

uo[i][j] = u[i][j];

}

}

/\* Now compute the stencil for each i,j pair. \*/

for ( i = 1; i <= nni; i++ )

{

for ( j = 1; j <= nnj; j++ )

{

u[i][j] = mu\*(uo[i+1][j]+uo[i-1][j]) + c\*uo[i][j]

+ nu\*(uo[i][j+1]+uo[i][j-1]);

}

}

/\* Now proceed to the next time step. \*/

}

/\* Okay, the numerical computation of u is completed. \*/

/\* Let's look now at a specific grid point. \*/

i = (int) ell1 / (20\*dx);

j = (int) ell2 / (100\*dy);

/\* Compute the actual temperature at this point. \*/

actual = 0.0;

for ( m = 1; m < 1000; m += 2 )

{

for ( n = 1; n < 1000; n += 2 )

{

actual += 1.0/(m\*n) \* exp( - dcoeff \* pi\*pi \*

( m\*m/(ell1\*ell1) + n\*n/(ell2\*ell2) ) \*

numTimeSteps\*dt )

\* sin( m\*pi\*i\*dx/ell1 ) \* sin( n\*pi\*j\*dy/ell2 );

}

}

actual \*= 16.0\*u\_0 / (pi\*pi);

/\* Compare the actual with the numerical result at this point. \*/

printf( "After %d time steps some results are:\n", numTimeSteps );

printf( "actual u[%d][%d] = %f\n", i, j, actual );

printf( "computed u[%d][%d] = %f\n", i, j, u[i][j] );

/\* And let's look at another specific grid point. \*/

i = (int) ell1 / (8\*dx);

j = (int) ell2 / (100\*dy);

/\* Compute the actual temperature at this point. \*/

actual = 0.0;

for ( m = 1; m < 1000; m += 2 )

{

for ( n = 1; n < 1000; n += 2 )

{

actual += 1.0/(m\*n) \* exp( - dcoeff \* pi\*pi \*

( m\*m/(ell1\*ell1) + n\*n/(ell2\*ell2) ) \*

numTimeSteps\*dt )

\* sin( m\*pi\*i\*dx/ell1 ) \* sin( n\*pi\*j\*dy/ell2 );

}

}

actual \*= 16.0\*u\_0 / (pi\*pi);

/\* Compare the actual with the numerical result at this point. \*/

printf( "After %d time steps some results are:\n", numTimeSteps );

printf( "actual u[%d][%d] = %f\n", i, j, actual );

printf( "computed u[%d][%d] = %f\n", i, j, u[i][j] );

return 0;

}

Note: If you didn't copy source files from HPC's tutorial directory, then copy the above source code into a file called serialDiffusion.c inside your ~/MPI\_Tutorial/SerialDiffusion/ directory.

To compile the code, change to the ~/MPI\_Tutorial/SerialDiffusion/ directory and then type [since this is not a MPI program, we are not going to use the Makefile we've been using all along]:

[agopu@bc81 agopu]$ cd ~/MPI\_Tutorial/SerialDiffusion/

[agopu@bh2 SerialDiffusion]$ gcc serialDiffusion.c -o serialDiffusion -lm

Note: Our serialDiffusion program uses functions like sin () and exp (). That's whay we've used the -lm flag in the above call to gcc ...to link with the Math library.

To run this program on a interactive node you got through qsub -I, do:

[agopu@bc81 SerialDiffusion]$ ./serialDiffusion 8000 1000 100

### The results of serialDiffusion

The output of serialDiffusion 8000 1000 100 is:

After 100 time steps some results are:

actual u[400][10] = 0.874737

computed u[400][10] = 0.873978

After 100 time steps some results are:

actual u[1000][10] = 0.874806

computed u[1000][10] = 0.873978

The computed and the actual differ by one part in a thousand -- Good enough.

Although these results are for a point very near the zero-temperature boundary, after 100 time steps the temperature at the point has dropped by only 13% from the initial condition Uo = 1.0. Not much has happened during our simulation. In fact if we look at an interior point, nothing has happened:

After 100 time steps some results are:

actual u[1000][500] = 1.000023

computed u[1000][500] = 1.000000

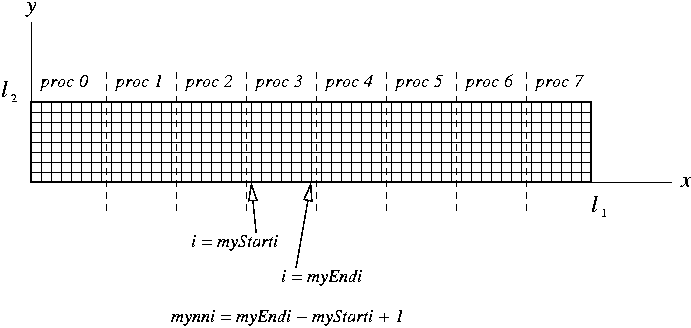
This is a consequence of the Courant condition. Our time step size is so small, in order to ensure stability, that we need to compute many steps to see some significant evolution in the system. After 100 steps only the points near the zero-temperature boundary have experienced a temperature drop. Points further in the interior have not yet been affected by the zero-temperature bath.

You can see the advantage of beginning with a uniform initial condition Uo = 1. Understanding the behavior of our model is much easier than if the initial condition were some trigonometric functional form.

# Parallelizing the Numerical Solution of the Diffusion Equation

### Domain decomposition

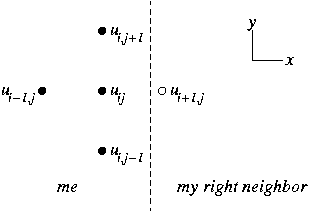
In our problem the easiest way to allocate grid points among the processors is to decompose along the x-axis:



At each time step, each processor computes only it's portion of the grid.

### Ghost points

However, when a processor computes a grid point that lies adjacent to the edge of its portion, one of the stencil points is actually owned by the neighboring processor. This point is one of the processor's "ghost points."



Ghost points are points the processor needs to compute all its stencils. However, the processor never computes the ghost points themselves. Ghost points are computed during the prior time step by neighboring processors.

To compute this stencil, the processor must have received the temperature at this ghost point from the neighbor at the conclusion of the prior time step. Ergo, processors must share their ghost points between themselves after each time step. Interprocessor communication is required at the conclusion of each time step.

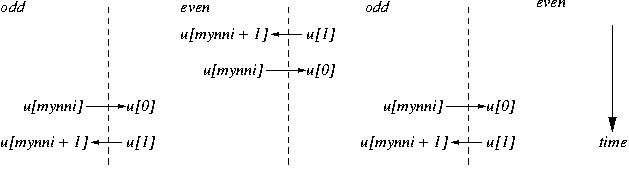
A processors ghost points are u[0][j], which are received from its left neighbor's u[mynni][j], and u[mynni + 1][j], which are received from its right neighbor's u[1][j], for all j.

### Interprocessor exchange of ghost points

We will use the MPI\_Send and MPI\_Recv functions to transfer ghost points between the processors. These are blocking functions; they do not return until the communication is complete (in many installations). So we will need to take care that a deadlock situation does not arise.

A deadlock arises in a perfectly symmetric situation when, for instance, two processors are sending to each other, and each processor's call to MPI\_Send never returns so that it can call MPI\_Recv. Better to program asymmetrically, whereby one processor first calls MPI\_Send and the other first calls MPI\_Recv.

To prevent deadlocks we'll authorize odd-numbered processors to MPI\_Send first and MPI\_Recv second, while even-numbered processors will MPI\_Recv first and MPI\_Send second.



# parallelDiffusion

### parallelDiffusion.c

The C version of parallelDiffusion is as follows:

/\* WARNING: There is a bug in this program! \*/

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <mpi.h>

const double pi = 3.1415926535897932385;

int main( int argc, char \*argv[] )

{

/\* Arguments required for executing argv[0]: \*/

/\* 0 -> serialDiffusion \*/

/\* 1 -> nni \*/

/\* 2 -> nnj \*/

/\* 3 -> numTimeSteps \*/

double \*\*uo; /\* solution at previous time step \*/

double \*\*u; /\* solution at current time step \*/

double mu, nu, c; /\* parameters of the FDE \*/

double ell1,ell2; /\* x,y grid dimensions, 0 <= x <= ell1, etc. \*/

double dx,dy; /\* x,y grid spacing, or spacial step size \*/

int nni,nnj; /\* x,y number of interior grid points; excl. bdy. pts.\*/

int numTimeSteps; /\* when to end simulation \*/

double dt; /\* time step size \*/

double dcoeff; /\* diffusion coefficient \*/

double u\_0; /\* uniform initial conditions \*/

double actual; /\* the true solution \*/

int i,j,k,m,n,timeStep;

int myStarti,myEndi,mynni; /\* my start, end, and number of grid points \*/

/\* in x-direction \*/

int leftProc,rightProc; /\* my neighboring processes, left and right \*/

int myrank,numProcs;

/\* Declare an MPI status variable. \*/

MPI\_Status status;

/\* Initialize the MPI API: \*/

MPI\_Init(&argc, &argv);

/\* Request my ID number: \*/

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank);

/\* I'll also ask how many other processors are out there: \*/

MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcs);

/\* Okay. The preparations have been made. \*/

/\* Set the diffusion coefficient, grid dimensions, \*/

/\* and the uniform initial condition. \*/

dcoeff = 1.0;

ell1 = 7.0;

ell2 = 1.0;

u\_0 = 1.0;

/\* Get the number of interior grid points and \*/

/\* number of time steps from the command line. \*/

nni = atoi( argv[1] );

nnj = atoi( argv[2] );

numTimeSteps = atoi( argv[3] );

/\* Compute number of x-direction grid points allocated to me, mynni. \*/

/\* Compute my starting and ending x-direction grid points, myStarti and \*/

/\* myEndi. \*/

mynni = nni / numProcs;

if ( myrank < (nni % numProcs) )

{

mynni++;

myStarti = myrank\*mynni + 1;

}

else

{

myStarti = myrank\*mynni + 1 + (nni % numProcs);

}

myEndi = myStarti + mynni - 1;

/\* Allocate my memory for my portion of the solution arrays. \*/

/\* Include boundaries. \*/

/\* As declared above, u is a pointer to a pointer to a double. \*/

/\* Allocate my memory so u points to the first element of a \*/

/\* (mynni+2)-long array of pointers to doubles. Likewise for uo. \*/

u = (double \*\*) malloc( (size\_t) ( (mynni+2) \* sizeof(double\*) ) );

uo = (double \*\*) malloc( (size\_t) ( (mynni+2) \* sizeof(double\*) ) );

/\* u[0] is a pointer to a double. \*/

/\* Allocate my memory so u[0] points to the first element of a \*/

/\* (mynni+2)\*(nnj+2)-long array of doubles. Likewise for uo[0]. \*/

u[0] =

(double \*) malloc( (size\_t) ( ((mynni+2)\*(nnj+2)) \* sizeof(double) ) );

uo[0] =

(double \*) malloc( (size\_t) ( ((mynni+2)\*(nnj+2)) \* sizeof(double) ) );

/\* Now set pointer u[1] to point to \*(u[0] + (nnj+2)). \*/

/\* And set pointer u[2] to point to \*(u[0] + 2\*(nnj+2)) \*/

/\* or \*(u[1] + (nnj+2)). \*/

/\* And so forth. Likewise for uo[1], uo[2], etc. \*/

for ( i = 1; i < mynni+2; i++ )

{

u[i] = u[i-1] + (nnj+2);

uo[i] = uo[i-1] + (nnj+2);

}

/\* We have now dynamically allocated my memory for the arrays \*/

/\* u[mynni+2][nnj+2] and uo[mynni+2][nnj+2] which are large \*/

/\* enough to include the boundary points. \*/

/\* Set the ranks of the processors to the left and right of me. \*/

/\* These are the processors I will communicate with. \*/

rightProc = myrank + 1;

if(rightProc == numProcs) rightProc = MPI\_PROC\_NULL;

leftProc = myrank - 1;

if(leftProc == -1) leftProc = MPI\_PROC\_NULL;

/\* Compute dx and dy, and set dt so that the Courant condition \*/

/\* is satisfied. \*/

dx = ell1 / ( nni + 1 );

dy = ell2 / ( nnj + 1 );

dt = 0.49 / ( dcoeff\*( 1.0/(dx\*dx) + 1.0/(dy\*dy) ) );

/\* Set uniform initial conditions on my portion of the grid. \*/

for ( i = 1; i <= mynni; i++ )

{

for ( j = 1; j <= nnj; j++ )

{

u[i][j] = u\_0;

}

}

/\* Set boundary conditions for my portion of the grid. \*/

/\* Assume Dirichlet boundary conditions equal to 0. \*/

/\* First the y = 0 boundary. \*/

for ( i = 0; i <= mynni+1; i++ )

{

u[i][0] = 0.0;

}

/\* Second the y = ell2 boundary. \*/

for ( i = 0; i <= mynni+1; i++ )

{

u[i][nnj+1] = 0.0;

}

/\* Third the x = 0 boundary. \*/

if ( myrank == 0 )

{

for ( j = 1; j <= nnj; j++ )

{

u[0][j] = 0.0;

}

}

/\* Finally the x = ell1 boundary. \*/

if ( myrank == numProcs-1 )

{

for ( j = 1; j <= nnj; j++ )

{

u[mynni+1][j] = 0.0;

}

}

/\* Solve the 2-D diffusion equation by the explicit FTCS method. \*/

mu = dcoeff\*dt / (dx\*dx);

nu = dcoeff\*dt / (dy\*dy);

c = 1.0 - 2.0\*( mu + nu );

for ( timeStep = 1; timeStep <= numTimeSteps; timeStep++ )

{

/\* Update previous time step data. \*/

for ( i = 0; i <= mynni+1; i++ )

{

for ( j = 0; j <= nnj+1; j++ )

{

uo[i][j] = u[i][j];

}

}

/\* Now compute the stencil for each ij pair that I own. \*/

for ( i = 1; i <= mynni; i++ )

{

for ( j = 1; j <= nnj; j++ )

{

u[i][j] = mu\*(uo[i+1][j]+uo[i-1][j]) + c\*uo[i][j]

+ nu\*(uo[i][j+1]+uo[i][j-1]);

}

}

/\* Now send/receive ghost points to/from my left and right neighbors. \*/

/\* Be careful that no deadlock situations arise! \*/

/\* To avoid deadlocks, we let all odd-ranked processors send first \*/

/\* and receive second. \*/

if ( (myrank % 2) == 1 )

{

MPI\_Send( u[1], nnj+2, MPI\_DOUBLE, leftProc, 0, MPI\_COMM\_WORLD );

MPI\_Recv( u[0], nnj+2, MPI\_DOUBLE, leftProc, 0, MPI\_COMM\_WORLD,

&status );

MPI\_Send( u[mynni], nnj+2, MPI\_DOUBLE, rightProc, 0,

MPI\_COMM\_WORLD );

MPI\_Recv( u[mynni+1], nnj+2, MPI\_DOUBLE, rightProc, 0,

MPI\_COMM\_WORLD, &status );

}

/\* And we let all even-ranked processors receive first and \*/

/\* and send second. \*/

else

{

MPI\_Recv( u[mynni+1], nnj+2, MPI\_DOUBLE, rightProc, 0,

MPI\_COMM\_WORLD, &status );

MPI\_Send( u[mynni], nnj+2, MPI\_DOUBLE, rightProc, 0,

MPI\_COMM\_WORLD );

MPI\_Recv( u[0], nnj+2, MPI\_DOUBLE, leftProc, 0, MPI\_COMM\_WORLD,

&status );

MPI\_Send( u[1], nnj+2, MPI\_DOUBLE, leftProc, 0, MPI\_COMM\_WORLD );

}

/\* Now proceed to the next time step. \*/

}

/\* Now that I am done we should consolidate our results somewhere. \*/

/\* But this is just a test program, so let's not bother. Rather, \*/

/\* let's just compare our results at a representative point with \*/

/\* the actual solution. \*/

/\* Let's look now at a specific grid point. \*/

i = (int) ell1 / (20\*dx);

j = (int) ell2 / (100\*dy);

/\* If I own this grid point, then let me do the work. \*/

if ( (i>=myStarti) && (i<=myEndi) )

{

/\* Compute the actual temperature at this point. \*/

actual = 0.0;

for ( m = 1; m < 1000; m += 2 )

{

for ( n = 1; n < 1000; n += 2 )

{

actual += 1.0/(m\*n) \* exp( - dcoeff \* pi\*pi \*

( m\*m/(ell1\*ell1) + n\*n/(ell2\*ell2) )\*

numTimeSteps\*dt )

\* sin( m\*pi\*i\*dx/ell1 ) \* sin( n\*pi\*j\*dy/ell2 );

}

}

actual \*= 16.0\*u\_0 / (pi\*pi);

/\* Compare the actual with the numerical result at this point. \*/

printf( "After %d time steps some results are:\n", numTimeSteps );

printf( "actual u[%d][%d] = %f\n", i, j, actual );

printf( "computed u[%d][%d] = %f\n", i, j, u[i-myStarti+1][j] );

}

/\* Let's look now at another specific grid point. \*/

i = (int) ell1 / (8\*dx);

j = (int) ell2 / (100\*dy);

/\* If I own this grid point, then let me do the work. \*/

if ( (i>=myStarti) && (i<=myEndi) )

{

/\* Compute the actual temperature at this point. \*/

actual = 0.0;

for ( m = 1; m < 1000; m += 2 )

{

for ( n = 1; n < 1000; n += 2 )

{

actual += 1.0/(m\*n) \* exp( - dcoeff \* pi\*pi \*

( m\*m/(ell1\*ell1) + n\*n/(ell2\*ell2) )\*

numTimeSteps\*dt )

\* sin( m\*pi\*i\*dx/ell1 ) \* sin( n\*pi\*j\*dy/ell2 );

}

}

actual \*= 16.0\*u\_0 / (pi\*pi);

/\* Compare the actual with the numerical result at this point. \*/

printf( "After %d time steps some results are:\n", numTimeSteps );

printf( "actual u[%d][%d] = %f\n", i, j, actual );

printf( "computed u[%d][%d] = %f\n", i, j, u[i-myStarti+1][j] );

}

/\* Close the MIP API: \*/

MPI\_Finalize();

return 0;

}