Parallel Numerical Solution of PDEs with Message Passing

Ge Baolai
SHARCNET
The University of Western Ontario

January, 2008

1 Overview

This tutorial aims to give an introduction to the design of parallel numerical procedures for solving partial differential equations using finite difference method. In particular we consider a simple, two dimensional parabolic equation (diffusion equation) on a unit square. The example is simple, but the intended numerical procedures are sophisticated enough to serve the purpose of a walk-through demonstration of how to implement numerical solution of partial differential equations in a parallel paradigm.

The problem we are trying to solve is considered a toy example. Consider the finite difference method on a uniform grid. The computational domain is decomposed into a number of sub-domains, in which the computational tasks are performed in parallel. We examine the explicit and implicit schemes and discuss the corresponding approaches to parallel processing in the framework of message passing.

This tutorial contains the following sections

1. Overview
2. Two dimensional diffusion equation
3. Numerical procedures
4. Implementation

For the ease of comprehension we attempt to present the algorithms in pseudo code in a high level, language independent manner. Readers can easily translate the code into their favourite language.

Due to the mathematical content, this tutorial is best viewed in PDF (http://www.sharcnet.ca/Documents/tutorials/heat2d/main.pdf) format. It is also available in HTML (http://www.sharcnet.ca/Documents/tutorials/heat2d/html/) and MathML (http://www.sharcnet.ca/Documents/tutorials/heat2d/main.xml).

2 Two Dimensional Diffusion Equation

Let's consider a simple, two dimensional parabolic equation

$$u_t = a(u_{xx} + u_{yy}), \quad 0 < x < 1, \ 0 < y < 1 \tag{1}$$

on a unit square, with initial condition $u(x, y, 0) = \phi(x, y)$ and boundary conditions $u(x, y, t) = 0$ along the four sides. The goal is to solve for $u(x, y, t)$ for $t > 0$.

Though an analytic solution can be obtained due to the simple form of the equation, in general the initial boundary value problem will need to be solved numerically. There are a few different numerical approaches that can be used. We use the finite difference method, which is simple to understand and easy to implement. As well it is easy to have the convergence and stability analysis if one would like to further pursue the details.

The solution of u using finit difference method on a uniform grid is plotted for three different times in Figure 1-3.

3 Numerical Procedures

We consider two typical schemes – explicit and implicit schemes. As we will see, both schemes have their pros and cons in the context of parallel processing. Interested readers may refer to the works on numerical solution of partial differential equations, for example, [2] and [3] for details.

3.1 Explicit Scheme

We use finite difference approximation on an $N_x \times N_y$ grid (Figure 4). Denote

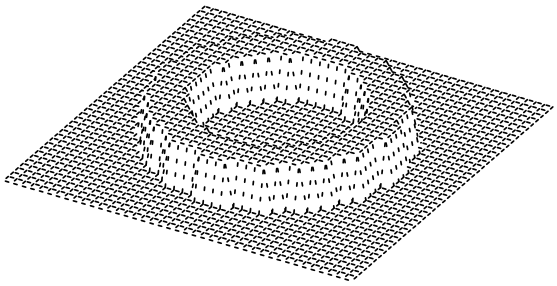

Figure 1: Initial value of u at $t = t_0$.

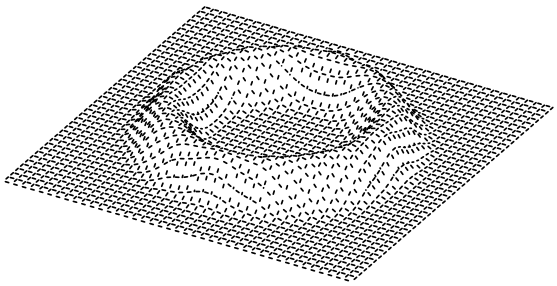

Figure 2: Solution of u at $t = t_4$.

by $v_{i,j}^k$ the finite difference approximation of u at grid point (i, j) at time t_k.

First we consider the explicit scheme

$$\frac{v_{i,j}^{k+1} - v_{i,j}^k}{\Delta t} = a \left(\frac{v_{i+1,j}^k - 2v_{i,j}^k + v_{i-1,j}^k}{\Delta x^2} + \frac{v_{i,j+1}^k - 2v_{i,j}^k + v_{i,j-1}^k}{\Delta y^2} \right)$$

or

$$v_{i,j}^{k+1} = v_{i,j}^k + a\Delta t \left(\frac{v_{i+1,j}^k - 2v_{i,j}^k + v_{i-1,j}^k}{\Delta x^2} + \frac{v_{i,j+1}^k - 2v_{i,j}^k + v_{i,j-1}^k}{\Delta y^2} \right). \qquad (2)$$

Eq. (2) says that the value of v at next time step t_{k+1} can be obtained solely from the values of v in the neighborhood of (i, j) at time step t_k. It reveals that the values of v can be calculated independently of each other, thus, the calculations can be performed in parallel.

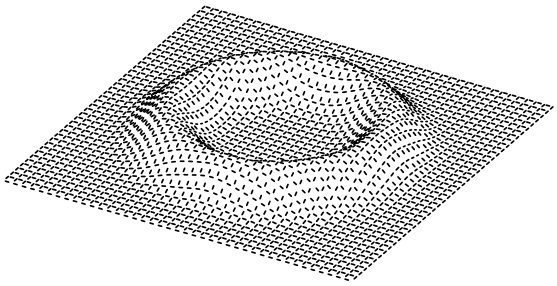

Figure 3: Solution of u at $t = t_9$.

Technical Points 1 *The explicit scheme is naturally suitable for parallel processing as the value of v at each grid point can be updated independently of others. A serious drawback of the explicit schemes, however, is the restriction in the time step size. The convergence and stability of the scheme require that the following be satisfied*

$$\Delta t \leq \frac{1}{2a} \frac{\Delta x^2 \Delta y^2}{\Delta x^2 + \Delta y^2}.$$

For example, if we set $\Delta x = 0.05$, $\Delta y = 0.05$, then the maximum value of Δt that we can have is 0.000625. *So if we are to solve for $u(x, y, t)$ in (1) for $t = 0$ to $t = 1$, we will have to use at least* 1600 *time steps.*

A pseudo code for the explicit scheme (2) is given in Algorithm 1.

ALGORITHM 1 *Explicit scheme: Serial version.*

```
set initial condition on uold(i,j), for all i,j;
for n=0,t=0.0; n<nt; n++,t+=dt, do
   for i=1,nx, j=1,ny, do
      uxx := (uold(i+1,j) - 2*uold(i,j) + uold(i-1,j))/dx2;
      uyy := (uold(i,j+1) - 2*uold(i,j) + uold(i,j-1))/dy2;
      u(i,j) := dt*a*(uxx + uyy) + uold(i,j);
   end
   swap u and uold;
end
```

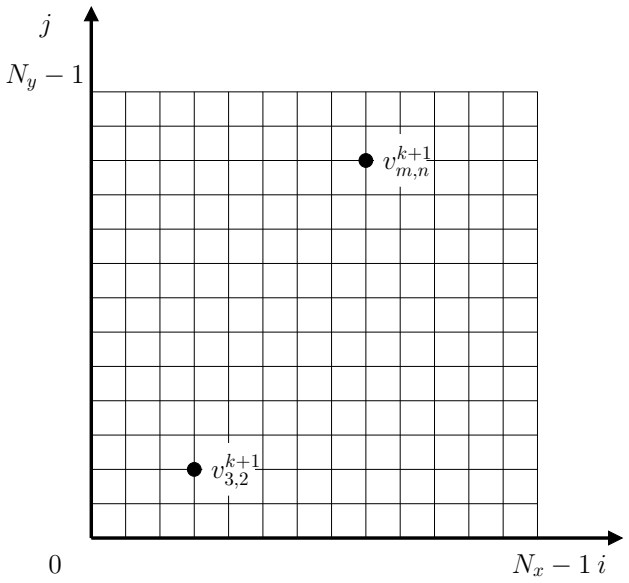

Figure 4: Finite difference grid. With explicit scheme, $v_{3,2}^{k+1}$ and $v_{m,n}^{k+1}$ can be calculated independently.

Here obvious notations are used: `dt` for Δt, `dx` and `dy` for Δx and Δy, `nx` and `ny` for N_x and N_y, etc. We use two arrays `u` and `uold` to store the new $v_{i,j}^{k+1}$ and old $v_{i,j}^{k}$ respectively. The outer loop is for time integration and the inner loop is for the calculation of the new $v_{i,j}^{k+1}$ values over all grid points $1 \leq i \leq N_x$ and $1 \leq j \leq N_x$. At each time step, we swap arrays `u` and `uold` so that `uold` holds the most recent updates ready for the next time step.

Technical Points 2 *To date neither C nor C++ is able to create a dynamic, two dimensional array (and higher dimensional array too) via a single function call. For instance, to create a double precision, two dimensional $n_1 \times n_2$ array* `A` *on demand, one would typically do the following:*

```
double **A;

A = (double **)malloc(sizeof(double*)*n1);
for (i = 0; i < n1; i++)
    A[i] = (double *)malloc(sizeof(double)*n2);
```

That is we first create a one dimensional array of length n_1, with each element being to hold an address, and then for each array element `A[i]` *we allocate a double precision one dimensional array of n_2 elements and have* `A[i]` *point to this array.*

While this does not seem to bring too much inconvenience as it is a one time effort and one can reference to `A[i][j]` *without extra work, there might be a potential problem. With language standard up to date, it is not guaranteed that the memory allocated for* `A` *is consecutive through a sequence of calls to* `malloc()`*. It may become a problem when one attempts to take a shortcut and map the two dimensional array* `A` *to an one dimensional array. For instance, in a call to MPI [4] function* `MPI_Send()`

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)
```

to send a request to a peer process, if one passes the address of `A[0][0]` *to the send buffer,*

```
MPI_Send(&A[0][0], n1*n2, MPI_DOUBLE, i, TAG_REQ, MPI_COMM_WORLD)
```

hoping that the array elements are picked up consecutively, then there might have unexpected data mismatch issues.

A remedy for this is to allocate an one dimensional array `B` *of size $n_1 \times n_2$ elements and have* `A[i]` *point to the appropriate chucks of* `B`*. The following code show how this works*

```
    B = (double *)calloc(n1*n2, sizeof(double));
    assert(NULL != B);
    A = (double **)malloc(sizeof(double *)*n1);
    assert(A);

    A[0] = B;
    for (i = 1; i <= n1; i++)
        A[i] = A[i-1] + n2;
```

Another point we try to make here is the use of pointers to reference to the storages that hold the old and updated values of v for performance. In C/C++ the use of pointers for shortcuts can be found everywhere. In Fortran, pointers can be used in the same way, but the syntax is not as simple as in C/C++. The following code segment demonstrates the use of pointers for array swapping in Fortran.

```
   real, dimension(:,:), allocatable, target :: w_u, w_uold
   real, dimension(:,:), pointer :: u, uold, tmp
   ... ...
   ! Allocate spaces for u and uold
   allocate(w_u(nx,ny))
   allocate(w_uold(nx,ny)
```

```
! Use pointers to reference storages
uold => w_uold
u => w_u

! Reference to u and uold as for regular arrays
while (t < tend) do
   t = t + dt
   do j = 1, nx
      do i = 1, nx
         uxx = (uold(i+1,j) - 2*uold(i,j) + uold(i-1,j))/dx2
         uyy = (uold(i,j+1) - 2*uold(i,j) + uold(i,j-1))/dy2
         u(i,j) = dt*a*(uxx + uyy) + uold(i,j)
      end do
   end do

   ! Swap u and uold
   tmp => uold
   uold => u
   u => tmp
end do
```

In the above sample code, we first create two work spaces held by `w_u` *and* `w_uold`*. Then we use pointers* `u` *and* `uold` *to reference the addresses of* `w_u` *and* `w_uold` *respectively. With* `u` *and* `uold` *array indexing is done in a regular way as if* `u` *and* `uold` *were regular arrays. Note that unlike in C/C++, pointer assignment in Fortran takes a different form.*

In order to be able to perform the calculations in parallel, we divide the finite difference grid in vertical direction into N subgrids as shown in Figure 5. We assign each subgrid to one processor, and have the updates of $v_{i,j}^{k+1}$ in Eq. (2) done on that processor. Note that at t_1 and onwards, the value of v at one of each five-point stencil on the boundary where the original grid is divided is in fact updated in the neighboring subgrid. This requires that value be acquired from the neighbor.

In the paradigm of message passing, each process P_ℓ, $0 \leq \ell \leq N-1$ holds the portion of data for subgrid ℓ (as shown in Figure 5) it works on. In this example, before any processor is able to update all the values from v^k to v_{k+1}, it must acquire the boundary data v^k calculated by the neighboring processors. Likewise each processor must also send those updated v's along the dividing boundary of subgrid to its neighboring processors in order for them to calculate the new values of v for the next time step. Such communications – data synchronization – among the processors must take place at each time step to replicate the "synchronous" nature of the explicit scheme.

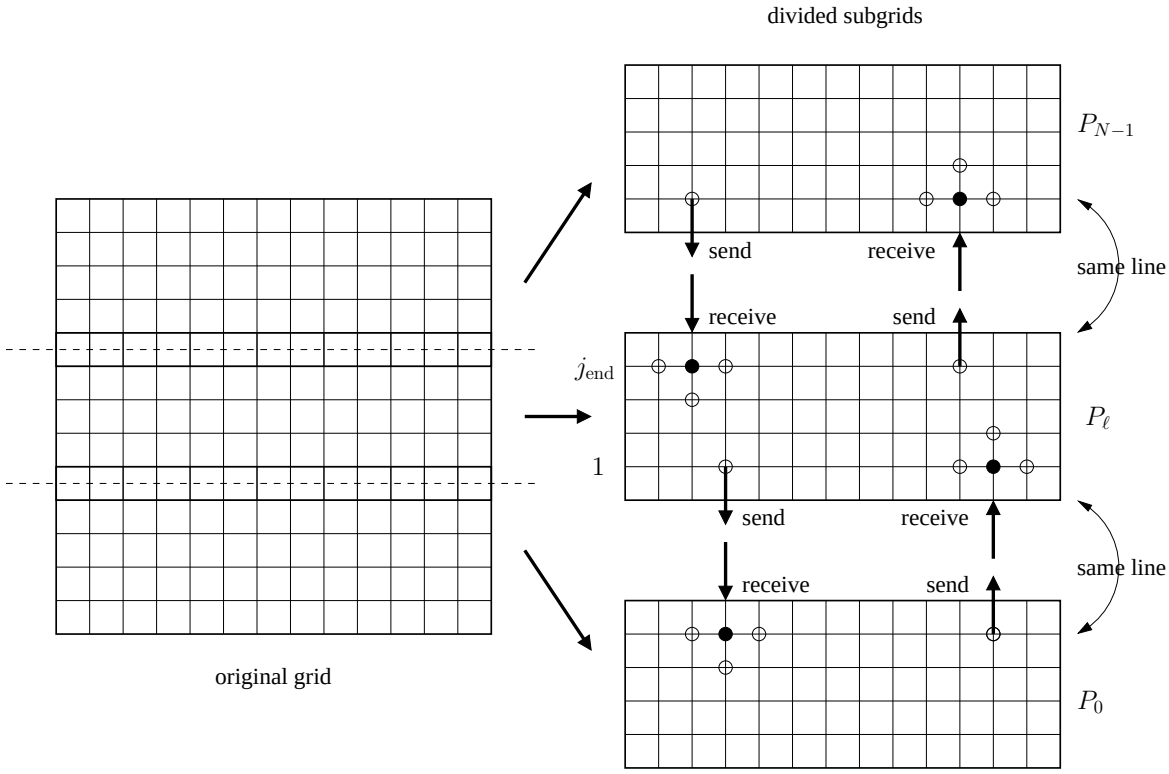

Figure 5: Finite difference grid divided into subgrids.

ALGORITHM 2 *Explicit scheme: Parallel version.*

```
set initial condition on uold(i,j), for all i,j;
if (0 == myid) send chunks of uold to others;
for n=0,t=0.0; n<nt; n=n+1,t=t+dt; do
   for i=1 to nx; j=1 to jend, do
      uxx := (uold(i+1,j) - 2*uold(i,j) + uold(i-1,j))/dx2;
      uyy := (uold(i,j+1) - 2*uold(i,j) + uold(i,j-1))/dy2;
      u(i,j) := dt*a*(uxx + uyy) + uold(i,j);
   end
   exchange boundary data;
   swap u and uold;
end
if (0 != myid) then
```

```
   send u to process 0;
else
   recv u from process 1 to np-1 and place in u;
end if
```

The parallel algorithm is illustrated in pseudo code in Algorithm 2. It resembles most of the structure of its serial counterpart except for three major differences. First an ID `myid` is used to necessarily identify the process itself. The value of this ID is assigned by the system. Second the inner loop now uses local index for j, $1 \leq j \leq j_{\text{end}}$ for the storage arrays `u` and `uold` are local. Third two if branches are used. The first one is to populate the parameters on all processors, though this can be done in an alternative way, e.g. having each processor read the parameter by itself. The second if branch at the end simply lets the process with ID 0 – also known as the root process – to collect all the values of v and place them in the global arrays it holds.

Technical Points 3 *In the example we have shown the domain partition in one dimension – the unit square is divided into a number of rectangles in y-direction. We can also have two dimensional partition – dividing the domain in both directions into a grid of subdomains. It would be interesting to compare the two partitions as both are natural to think of. The following example shown in Figure 7 illustrates the communication costs for one- and two-dimensional partitions using* 16 *processors. The number in each sub domain indicates the number of communications that take place for the data exchange at each time steps. We look at both the number of communications and the estimate amount of data to be transferred in each partition. Denote by $M_{4\times4}$ the number of messages to be sent and by $D_{4\times4}$ the amount of data (units) in total to be transferred for the 4×4 partition. Assume, for the matter of simplicity, that the number of data points in each direction are equal, denoted by n. Then we have*

$$\begin{aligned} M_{4\times4} &= 2 \times 4 + 3 \times 8 + 4 \times 4 \\ &= 48, \end{aligned}$$

$$\begin{aligned} D_{4\times4} &= M_{4\times4} \cdot \frac{n}{4} \\ &= 12n. \end{aligned}$$

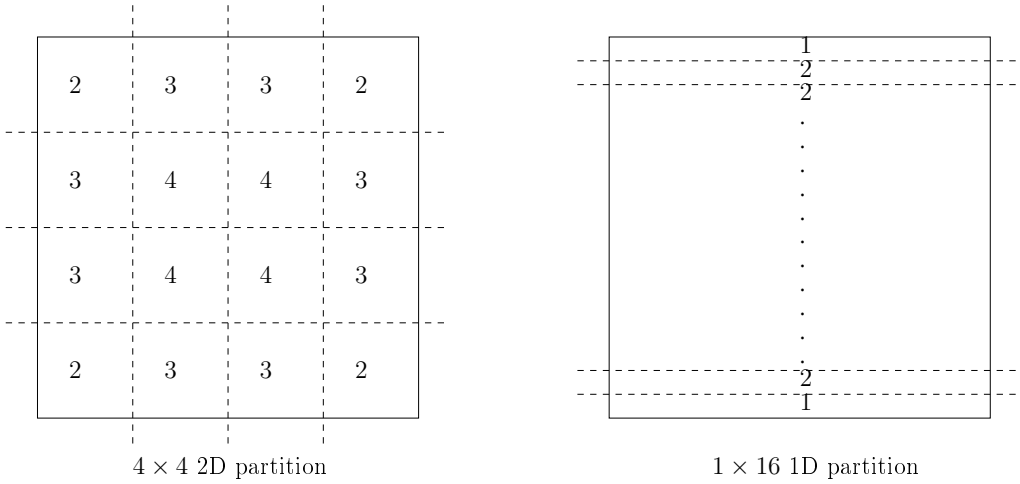

Figure 6: 1D and 2D partitions for 16 processors. The numbers indicate the number of communications with neighboring processors.

For 1×16 *partition, we have*

$$\begin{aligned} M_{1\times 16} &= (16-2)\times 2 + 2 \\ &= 30, \end{aligned}$$

$$\begin{aligned} D_{1\times 16} &= M_{1\times 16} \cdot \frac{n}{4} \\ &= 30n. \end{aligned}$$

This implies that 2D 4×4 *partition would requires more communications, but relatively low bandwidth, while 1D* 1×16 *partition requires less communication but high bandwidth.*

For general cases, assume that the $n_x \times n_y$ *grid is partitioned to* $P \times Q$ *subgrids which can be mapped to* $P \times Q$ *processors. The estimated number of messages and the total amount of data for 2D* $P \times Q$ *partiton and 1D* $1 \times PQ$ *partition are given below, respectively*

$$M_{P\times Q} = 4PQ - 2(P+Q),$$
$$D_{P\times Q} = 2(n_y P + n_x Q) - 2(n_x + n_y),$$
$$M_{1\times PQ} = 2(PQ-1),$$
$$D_{1\times PQ} = 2n_x(PQ-1).$$

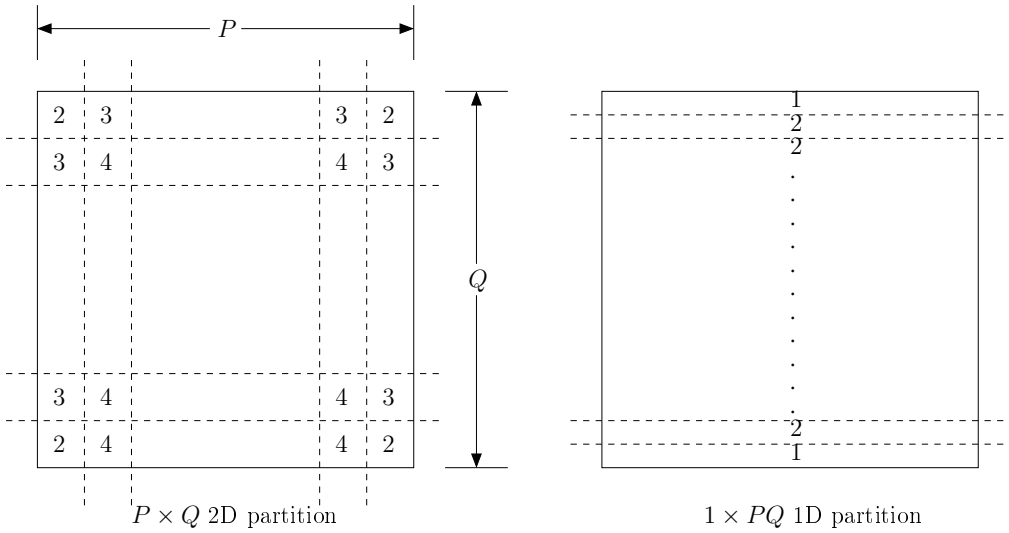

Figure 7: 1D and 2D partitions for $P \times Q$ 16 processors. The numbers indicate the number of communications with neighboring processors.

It can be seen that the number of messages to send/receive for 2D partitioning is always more than 1D, that is, $M_{P\times Q} > M_{1\times PQ}$, and for the total amount of data to be transferred, unless $n_x Q < n_y$, i.e. we have a "slender" grid in y-direction, 1D partition results in larger amount of data to be exchanged.

3.2 Implicit Scheme

As we have seen the explicit scheme is naturally ready for parallel processing, but it suffers from the drawback of numerical stability. Implicit schemes in general give a remedy for that. Consider the implicit, θ scheme that applies to our example

$$\begin{aligned}\frac{v_{i,j}^{k+1} - v_{i,j}^{k}}{\Delta t} &= \theta a \left(\frac{v_{i+1,j}^{k+1} - 2v_{i,j}^{k+1} + v_{i-1,j}^{k+1}}{\Delta x^2} + \frac{v_{i,j+1}^{k+1} - 2v_{i,j}^{k+1} + v_{i,j-1}^{k+1}}{\Delta y^2} \right) \\ &+ (1-\theta) a \left(\frac{v_{i+1,j}^{k} - 2v_{i,j}^{k} + v_{i-1,j}^{k}}{\Delta x^2} + \frac{v_{i,j+1}^{k} - 2v_{i,j}^{k} + v_{i,j-1}^{k}}{\Delta y^2} \right) \quad (3)\end{aligned}$$

or

$$[1+2(\alpha+\beta)]v_{i,j}^{k+1} - \alpha v_{i+1,j}^{k+1} - \alpha v_{i-1,j}^{k+1} - \beta v_{i,j+1}^{k+1} - \beta v_{i,j-1}^{k+1} = b_{i,j} \tag{4}$$

where

$$\alpha = \frac{\theta \Delta t a}{\Delta x^2}, \tag{5}$$

$$\beta = \frac{\theta \Delta t a}{\Delta y^2}, \tag{6}$$

$$b_{i,j} = v_{i,j}^k + (1-\theta)a\Delta t \left(\frac{v_{i+1,j}^k - 2v_{i,j}^k + v_{i-1,j}^k}{\Delta x^2} + \frac{v_{i,j+1}^k - 2v_{i,j}^k + v_{i,j-1}^k}{\Delta y^2} \right) \tag{7}$$

In particular, for $\theta = 1/2$, we have the second order in time, Crank-Nicholson scheme.

With implicit scheme (3), we need to solve a linear system (4) at each time step. Our focus now shifts to the numerical solution of linear systems, and we will show in the following how one can achieve parallelism again when solving the linear system.

For the purpose for demo only, we consider two schemes: Jacobi and SOR iterations for solving linear system (4). In Jacobi iteration, we rewrite (4) as

$$v_{i,j}^{k+1} = \frac{1}{1+2(\alpha+\beta)} \left(b_{i,j} - \alpha v_{i-1,j}^{k+1} - \alpha v_{i+1,j}^{k+1} - \beta v_{i,j+1}^{k+1} - \beta v_{i,j-1}^{k+1} \right),$$

which says that the value of $v_{i,j}^{k+1}$ can be obtained from the values at the neighboring points. This leads to the Jacobi iteration

ALGORITHM 3 *Jacobi iteration.*

$$v_{i,j}^{(m+1)} = \frac{1}{1+2(\alpha+\beta)} \left(b_{i,j} - \alpha v_{i+1,j}^{(m)} - \beta v_{i,j+1}^{(m)} - \alpha v_{i-1,j}^{(m)} - \beta v_{i,j-1}^{(m)} \right), \tag{8}$$

where we drop the time level superscripts k and use $v_{i,j}^{(m)}$ to stand for the approximation of value of $v_{i,j}^{k+1}$ at mth iteration.

The proof of the convergence of (8) is out of the scope of this tutorial. Interested readers may refer to the works on numerical linear algebra, for example [1]. The Jacobi iteration (8) resembles the characteristics of the procedure of explicit scheme. At each iteration, the update of $v_{i,j}^{(m+1)}$ depends only on the values of v at neighboring points at the previous iteration. Therefore the same parallel processing procedure can be easily applied to the iteration as is. Algorithm 4 describes the numerical procedures as illustrated in Figure 8 in a pseudo code

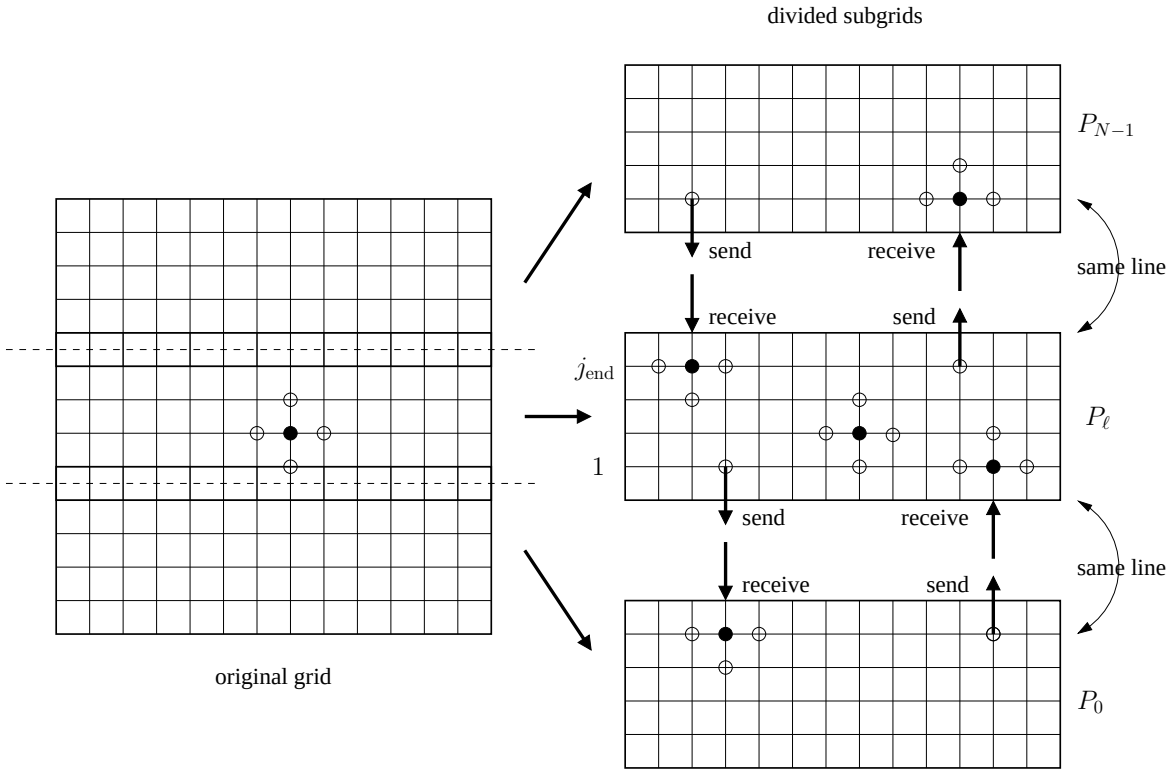

Figure 8: Jacobi iterations processed in parallel.

ALGORITHM 4 *Parallel Jacobi iteration.*

```
set initial condition on uold(i,j), for all i,j;
alpha := 0.5*dt*a/dx2; beta := 0.5*dt*a/dy2;
if (0 == myid) send chunks of uold to others;
for n=0,t=0.0; n<nt; n=n+1,t=t+dt; do
   set initial values for u(i,j);
   for m=0 to mmax; do
      exchange top boundary data;
      for i=1 to nx; j=1 to jend, do
         uxx := (uold(i+1,j) - 2*uold(i,j) + uold(i-1,j))/dx2;
         uyy := (uold(i,j+1) - 2*uold(i,j) + uold(i,j-1))/dy2;
         b := uold(i,j) + 0.5*dt*a*(uxx + uyy)
         unew(i,j) := b - (alpha*u(i-1,j) + alpha*u(i+1,j) +
```

```
            beta*u(i,j+1) + beta*u(i,j-1))/(1 + 2*(alpha+beta));
      end
      exchange bottom boundary data;
      if u converges, exit;
      swap u and unew;
   end
   swap u and uold;
end
if (0 != myid) then
   send u to process 0;
else
   recv u from process 1 to np-1 and place in u;
end if
```

Jacobi iteration in general converges very slowly. A more practical method is successive overrelaxation (SOR).

ALGORITHM 5 *SOR iteration – one "in-place" (Gauss-Seidel) iteration followed by a "relaxation"*

$$v_{i,j}^{*} = \frac{1}{1+2(\alpha+\beta)}\left(b_{i,j} - \alpha v_{i+1,j}^{(m)} - \beta v_{i,j+1}^{(m)} + -\alpha v_{i-1,j}^{(m+1)} - \beta v_{i,j-1}^{(m+1)}\right), \quad (9)$$

$$v_{i,j}^{(m+1)} = \omega v_{i,j}^{*} + (1-\omega) v_{i,j}^{(m)} \quad (10)$$

where α, β and $b_{i,j}$ are defined respectively in (5), (6) and (7), and ω is some empirically chosen parameter. In iteration (9), the value of $v_{i,j}$ is obtained using the most recent updated values, wherever available, at the neighboring points.

Alternatively (10) can be rewritten as the following more convenient form

$$r_{i,j}^{(m)} = b_{i,j} - [1+2(\alpha+\beta)]v_{i,j}^{(m)} + \alpha v_{i+1,j}^{(m)} + \beta v_{i,j+1}^{(m)} + +\alpha v_{i-1,j}^{(m)} + \beta v_{i,j-1}^{(m)}, \quad (11)$$

$$v_{i,j}^{(m+1)} = v_{i,j}^{(m)} + \frac{\omega}{1+2(\alpha+\beta)} r_{i,j}^{(m)}. \quad (12)$$

where the cumulative residual $r_{i,j}^{(m)}$ can be used for stop criterion test.

Assume the updates of $v_{i,j}$ are done in a lexicographic order (as shown in Figure 9) The fact that the $v_{i,j}$ are obtained using the most recent updated values, wherever available, at the neighboring points prohibits the updates of

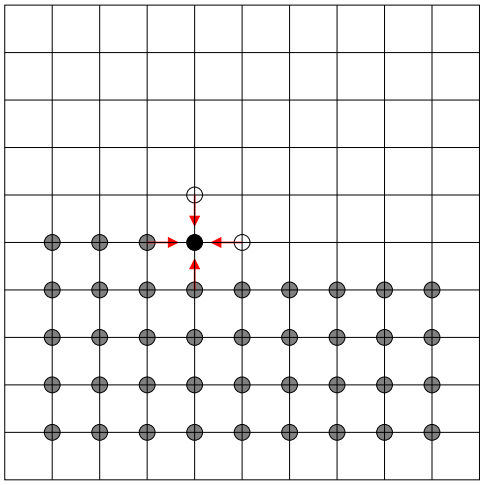

Figure 9: Iterative updates in lexicographic order. Both old and new values are used. The updates cannot be done in parallel.

$v_{i,j}^{(m+1)}$ from being calculated in parallel. To regain the parallelism, we modify the order of updates in the following way. First we label the grid points by indices as odd (black ones) and even (white ones) as shown in Figure 10. Then we perform a Jacobi iteration through odd (black) points, followed by a second pass through even (white) points. When sweeping through the odd (black) points, at each of of them, the update of $v_{i,j}$ depends only on the values at its neighboring even (white) points, which are not updated. This means that the updates at odd points can be done independently or in parallel. Similarly the updates at even points can be done in parallel as well.

The parallel SOR algorithm is summarized in the pseudo code below.

ALGORITHM 6 *Parallel SOR iteration.*

```
set initial condition on uold(i,j), for all i,j;
alpha := 0.5*dt*a/dx2; beta := 0.5*dt*a/dy2;
if (0 == myid) send chunks of uold to others;
for n=0,t=0.0; n<nt; n=n+1,t=t+dt; do
   set initial values for u(i,j);
   while ||r|| < tol and m < mmax; do
      m := m + 1;
      update odd points;
```

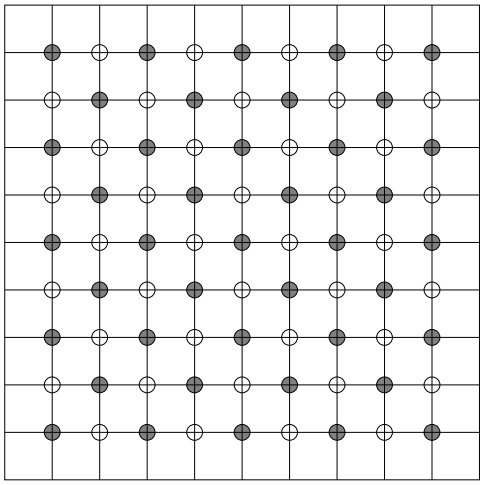

Figure 10: SOR iteration sweeping through odd points first, followed by a second pass through even points. Each pass can be performed in parallel.

```
      cumulate ||r||;
      exchange bottom boundary data;
      update even points;
      cumulate ||r||;
      exchange bottom boundary data;
      reduce to get global ||r||
   end
   swap u and uold;
end
if (0 != myid) then
   send u to process 0;
else
   recv u from process 1 to np-1 and place in u;
end if
```

Technical Points 4 *The finite difference method is based on local approximation and results in linear, sparse systems to solve at each time step. In practice, neither Jaocobi, nor SOR is used for solving linear systems due to the slow convergence.*

Instead, Krylov subspace based methods and multigrid method are widely used. Interested readers may refer to the context of iterative solution of linear sparse systems.

References

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[2] K. W. Morton and D. F. Mayers. *Numerical Solution of Partial Differential Equations*. Cambridge University Press, 1995.

[3] John Strikwerda. *Finite Difference Schemes and Partial Differential Equations*. SIAM, 2nd edition, 2004.

[4] Ewing Lusk William Gropp and Anthony Skjellum. *Using MPI*. The MIT Press, 2nd edition, 1999.