Comparison of various runs of a simple parallel Jacobi code in MPI

Anirudh Modi

April 10, 1998

1 Algorithm

The Jacobi method here is applied to solve the Laplace's equation $\nabla^2\phi = 0$. Using finite difference, the Laplace's equiation can be written as

$$\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\Delta x^2} + \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\Delta y^2} = 0$$

For $\Delta x = \Delta y$, we get

$$-4\phi_{i,j} + \phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} = 0$$

So, the Jacobi method simply becomes

$$\phi_{i,j}^{n+1} = \frac{1}{4}\left[\phi_{i+1,j}^{n} + \phi_{i-1,j}^{n} + \phi_{i,j+1}^{n} + \phi_{i,j-1}^{n}\right]$$

where n is the iteration level.

For the problem we are solving, we have a square domain with constant ϕ (in this case temperature) along two different continuous boundaries ($\phi = 0$ for one and $\phi = 100$ for the other which represents the initial temperature) and it varies linearly (from 0 to 100) in between the two regions.

2 Program Listing

The compile command used for the SP2 is:
mpxlf90 jacmpi.f -O3 -qstrict -qarch=pwr2 -qtune=pwr2 -qhot -o jacmpi

```
!****************************************************************
! FILE: jacmpi.f90
!
! DESCRIPTION:
!     This code solves the Laplace equation
!     using the Jacobi method, domain decomposition,
!     and MPI.  The boundary conditions used here are:
!----------------------------------------------------------------
!       f(0,y) = 0               0   < y < 5
!       f(x,5) = 0               0   < x < 5
!       f(x,0) = 0                  0   < x < 1
!       f(x,0) = linear         1   < x < 1.2
!       f(x,0) = 0                  1.2 < x < 5
!       f(5,y) = 100                  0   < y < 3
!       f(5,y) = linear               3   < y < 3.2
!       f(5,y) = 0                  3.2 < y < 5
!----------------------------------------------------------------
!     The initial guess is a random field for f.
!
! You may realize that certain portions of this code could
! be done in a simpler manner, but it is written the way
! it is for pedagogical reasons.  There are a number of
! things that this code does that you may find useful
! later on.
!
! ORIGINAL AUTHOR: Lyle Long, lnl@psu.edu, Penn State University
! LAST REVISED BY: Anirudh Modi, anirudh-modi@psu.edu (4/7/98-Tue)
!****************************************************************

Program jacobi

implicit none
include 'mpif.h'
integer mpierr,status(MPI_STATUS_SIZE)

integer nodex, nodey, nnodes, nx,ny,master,myeast, mywest, iter, &
        iters, myj,nabor,myi, mynorth, mysouth, nxw,nyw,nn, &
             cont, ntotal, request,i1,i2,j1,j2,ih1,ih2,jh1,jh2
```

```
! NODEX*NODEY gives the number of domains
parameter (nodex = NPROC_X, nodey = NPROC_Y)

! total no. of processors:
parameter (Nnodes = nodex*nodey)

! these are the total number of cells in each direction:
parameter (NX = NGRID_X, ny=NGRID_Y)

! total no. of cells:
parameter (ntotal = nx*ny )

! no. of cells on each processor
! (there are 2 ghost points around the edge of each domain)
parameter (nxw=nx/nodex+2, nyw=ny/nodey+2 )

parameter (MASTER = 0)

integer   numtasks,taskid,mtype,i,j,ncont,indgr

real*8    f(nxw,nyw),fold(nxw,nyw)
real*8    x(nxw,nyw),y(nxw,nyw)
real*8    fn(nxw), fe(nyw), fs(nxw), fw(nyw)
real*8    time,mflop,tottime,totmflop,mfloptmp,von,ranf
real*8    tstart, tend, second, xrange, yrange, xhole1, xhole2, &
                yhole1, yhole2, T1, T2, tcomm1, tcomm2, tcomm

character*30 pltitl, myfile

! find out how many processors there are (numtasks) and what is
! 'my' taskid is:
call MPI_INIT(mpierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numtasks,mpierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,taskid,mpierr)

if ( taskid == master ) then
        ! this code reads in the number of iterations from a file:
```

```
        open(11,file='input.jacmpi',form='formatted',status='unknown')
        read(11,*) iters
        close(11)
endif

! broadcast iters to all processors
call MPI_BCAST(iters,1,MPI_INTEGER,0,MPI_COMM_WORLD,mpierr)

tcomm = 0.0                     ! Initialise comm time                     !
T1 = 0.0                        ! Temperature BC 1
T2 = 100.0                      ! Temperature BC 2

! x-direction
xrange = 5.0                    ! x-range (origin at 0.0)
xhole1 = 1.0                    ! location of beginning of hole
xhole2 = 1.2                    ! location of ending of hole
ih1 = int(nx*xhole1/xrange)     ! global index of beginning of hole
ih2 = int(nx*xhole2/xrange)     ! global index of ending of hole

! y-direction
yrange = 5.0                    ! x-range (origin at 0.0)
yhole1 = 3.0                    ! location of beginning of hole
yhole2 = 3.2                    ! location of ending of hole
jh1 = int(ny*yhole1/yrange)     ! global index of beginning of hole
jh2 = int(ny*yhole2/yrange)     ! global index of ending of hole

if (taskid == 0) then
        write(6,*) ' This is node', taskid, ' (node 0)'
else
        write(6,*) ' This is node', taskid, ' (other nodes)'
endif

if ( nodex*nodey .ne. numtasks ) then
        if ( taskid == master ) then
                print *, ' must have nodex*nodey = numtasks !'
                print *, ' nodex=',nodex
                print *, ' nodey=',nodey
                print *, ' numtasks=',numtasks
```

```
        endif
        call mpi_abort( mpi_comm_world, 1 )
        stop
endif
! ******************** master task ****************************
if (taskid == MASTER) then

        print *, 'Number of tasks     = ', numtasks
        print *, 'Total cells         = ', ntotal
        print *, 'Cells per processor = ', nxw*nyw

        print *, 'No. of iterations   = ', iters

        print *, 'Data sent/node/step =', &
               (nyw*2.0 + nxw*2.0 )*8.0/1.0e6, &
             ' mbytes'

        print *, 'Total data on each node =', &
               (nxw*nyw*8.0*5.0)/1.0e6 , ' mbytes'

endif

! find i,j processor id nos.:
! (in addition to having taskid, I also label the processors
!  with i,j indices, this makes it easier to determine who
!  the neighboring processors are)
myj = taskid / nodex + 1
myi = taskid - (myj-1)*nodex  + 1

print *,'task, i, j = ',taskid,myi,myj

! find taskid numbers of neighbor processors :
mynorth = nabor(myi,   myj+1, nodex, nodey)
mysouth = nabor(myi,   myj-1, nodex, nodey)
myeast  = nabor(myi+1, myj,   nodex, nodey)
mywest  = nabor(myi-1, myj,   nodex, nodey)

print *, 'task,n,s,e,w=',taskid,mynorth,mysouth,myeast,mywest
```

```
! -- initialize f and edge arrays on each processor --------
! Initialize f randomly
fn = 0.0
fs = 0.0
f = 10.0 * ranf()
fe = 0.0
fw = 0.0

! lets also put it a spike in the initial guess
if ( taskid == master ) f(nxw/2,nyw/2) = 100.0

!----- initialize x,y on each processor, note these are global x,y's
x(1,:) = 0.0
x(nxw,:) = 0.0
y(:,1) = 0.0
y(:,nyw) = 0.0

do i=2,nxw-1
  x(i,:) = dble((dble(myi-1)+dble(i-2)/dble(nxw-2))*dble(xrange/nodex))
enddo
do j=2,nyw-1
  y(:,j) = dble((dble(myj-1)+dble(j-2)/dble(nyw-2))*dble(yrange/nodey))
enddo

! -----begin iterations-----------------------------------------
if (taskid == master) print *,'Begin iterating...'
tstart = mpi_wtime()

do iter = 1, iters
        if ( taskid == master ) print *, 'iter = ',iter

        ! apply BOUNDARY CONDITIONS on edges of domain:
        ! note that only certain processors need to do this.

        if ( myi == 1 )  f(2,2:nyw-1) = T1      ! x = 0.0

        if ( myi == nodex ) then                ! x = 5.0
```

```
        j1 = max( min(jh1 - nyw * (myj-1), nyw-1), 1)
        j2 = max( min(jh2 - nyw * (myj-1), nyw-1), 1)
        if (j1 > 1) f(nxw-1,2:j1) = T2
        do j = j1+1,j2
                f(nxw-1,j) = dble((j-j2-1)*(T1-T2)/(j2-j1))
        enddo
        if (j2 < nyw-1) f(nxw-1,j2+1:nyw-1) = T1
endif

if ( myj == 1 ) then                    ! y = 0.0
        i1 = max( min(ih1 - nxw * (myi-1), nxw-1), 1)
        i2 = max( min(ih2 - nxw * (myi-1), nxw-1), 1)
        if (i1 > 1) f(2:i1,2) = T1
        do i = i1+1,i2
                f(i,2) = dble((i-i1-1)*(T2-T1)/(i2-i1))
        enddo
        if (i2 < nxw-1) f(i2+1:nxw-1,2) = T2
endif

if ( myj == nodey ) f(1:nxw,nyw-1) = T1 ! y = 5.0

! copy edge data to smaller arrays for transfer:

!---- North & South----------
fs(2:nxw-1) = f(2:nxw-1,2)
fn(2:nxw-1) = f(2:nxw-1,nyw-1)

!--------East & west
fw(2:nyw-1) = f(2,2:nyw-1)
fe(2:nyw-1) = f(nxw-1,2:nyw-1)

tcomm1 = mpi_wtime()            ! Start of communication time

mtype = 111
! send to north
call mpi_isend(fn,nxw,MPI_DOUBLE_PRECISION,mynorth,mtype, &
        MPI_COMM_WORLD, request,mpierr)
```

```
! receive from south
call mpi_recv(fn,nxw,MPI_DOUBLE_PRECISION,mysouth,mtype, &
        MPI_COMM_WORLD,status,mpierr)

mtype = 222
! send to south
call mpi_isend(fs,nxw,MPI_DOUBLE_PRECISION,mysouth,mtype, &
        MPI_COMM_WORLD,request,mpierr)

! receive from north
call mpi_recv(fs,nxw,MPI_DOUBLE_PRECISION,mynorth,mtype, &
        MPI_COMM_WORLD,status,mpierr)

mtype = 333
! send to east
call mpi_isend(fe,nyw,MPI_DOUBLE_PRECISION,myeast,mtype, &
        MPI_COMM_WORLD,request,mpierr)

! receive from west
call mpi_recv(fe,nyw,MPI_DOUBLE_PRECISION,mywest,mtype, &
        MPI_COMM_WORLD,status,mpierr)

mtype = 444
! send to west
call mpi_isend(fw,nyw,MPI_DOUBLE_PRECISION,mywest,mtype, &
        MPI_COMM_WORLD,request,mpierr)

! receive from east
call mpi_recv(fw,nyw,MPI_DOUBLE_PRECISION,myeast,mtype, &
        MPI_COMM_WORLD,status,mpierr)

tcomm2 = mpi_wtime()            ! End of communication time
tcomm = tcomm + tcomm2 - tcomm1

! copy transferred arrays into ghost cells in main f arrays:

!---- North & South----------
f(2:nxw-1,1) = fn(2:nxw-1)
```

```
        f(2:nxw-1,nyw) = fs(2:nxw-1)

        !--------East & west
        f(1,2:nyw-1) = fe(2:nyw-1)
        f(nxw,2:nyw-1) = fw(2:nyw-1)

        fold = f
        f(2:nxw-1,2:nyw-1) = 0.25 * (fold(3:nxw,2:nyw-1) &
                             + fold(1:nxw-2,2:nyw-1)  &
                             + fold(2:nxw-1,3:nyw) &
                             + fold(2:nxw-1,1:nyw-2))
enddo
!   ---end iterations---------------------------------------------

tend = mpi_wtime()

time = tend - tstart
mflop = (4.0*iters*(nxw-2)*(nyw-2))/time/1.0e6
time = time - tcomm
print *, 'comp time=', time,' mflops/node=',mflop,' on task=',taskid
print *, 'comm time=', tcomm,' mflops/node=',mflop,' on task=',taskid

mtype = 20
call mpi_send(mflop,1,MPI_DOUBLE_PRECISION,0,mtype,MPI_COMM_WORLD,mpierr)

if ( taskid == MASTER ) then
        print *,'Receiving mflops nos. back to Master'

        mtype=20
        totmflop=0.0
        do nn=0,numtasks-1
        call mpi_recv(mfloptmp,1,MPI_DOUBLE_PRECISION,nn,mtype, &
                 MPI_COMM_WORLD,status,mpierr)
        totmflop = totmflop + mfloptmp
        enddo

        print *,' TOTAL MFLOPS = ',totmflop
endif
```

```
! have each node write a file :
!     --  create a processor-specific output file name:
! the following writes character info to the character myfile
! to create unique file names:

write(myfile,900) 'fort.', taskid+1
900      format(a,i3.3)

if ( ntotal < 5000000 ) then
        print *, ' my file = ',myfile
        open(22,file=myfile,form='formatted',status='unknown')
        write(22,*) 'title = "Jacobi"'
        write(22,*) 'variables = "x","y","T"'
        write(22,*) 'zone i = ',nxw-2,' j = ',nyw-2
        do i=2,nxw-1
        do j=2,nyw-1
                write(22,*) x(i,j), y(i,j), f(i,j)
        enddo
        write(22,*)
        enddo
endif

if (taskid == master) print *, ' Program Done.'

! quit MPI
call MPI_FINALIZE(mpierr)

stop
end
!*******************************************************************
function nabor(i,j,nx,ny)

integer i,j,nx,ny,nabor

if ( i <  1 ) i = nx
if ( i >  nx) i = 1
```

```
if ( j <  1 ) j = ny
if ( j >  ny) j = 1

nabor = (i-1) + (j-1)*nx

return
end

!*****************************************************************
real function ranf (  )

!    ***********************************************************
!    ** RETURNS A UNIFORM RANDOM VARIATE IN THE RANGE 0 TO 1.    **
!    **                                                         **
!    **                 ***************                         **
!    **                 **  WARNING  **                         **
!    **                 ***************                         **
!    **                                                         **
!    ** GOOD RANDOM NUMBER GENERATORS ARE MACHINE SPECIFIC.      **
!    ** PLEASE USE THE ONE RECOMMENDED FOR YOUR MACHINE.         **
!    ***********************************************************

integer     l, c, m
parameter ( l = 1029, c = 221591, m = 1048576 )

integer     seed
double precision dummy
save        seed
data        seed / 0 /

seed = mod ( seed * l + c, m )
ranf = real ( seed ) / m

return
end
!*****************************************************************
```

3 Results

The timing runs are carried on two different SP-2 machines, the NPACI-SP2 and the Penn State SP-2. The comparisons for the PSU-SP2 runs are also provided.

The Penn State SP2 (mcnally.cac.psu.edu) has 32 POWER2 Super Chip (PSC model 390) processors with 128 MBytes of memory on each processor running at 66 Mhz and are capable of a peak performance of 250 MFLOPS each. It is capable of a peak bi-directional data transfer rate of 35 MB/second between each node pair.

The NPACI SP2 (sp.npaci.edu) has 128 thin node POWER2 Super Chip (P2SC) processors with 256 MBytes of memory on each processor running at 160 Mhz and are capable of a peak performance of 640 MFLOPS each. It is capable of a peak bi-directional data transfer rate of 110 MB/second between each node pair.

Table 1: Data for MPI run after 100 iterations on PSU-SP2

P	**N**	**Comm**	**Calc**	**Total**	**Mflops/P**	**% R**	**% Eff**	**MB/s**
1	400	0.14	3.63	3.77	16.99	-	100	-
4	800	1.76	3.85	5.61	11.40	83.7	67.1	2.92
8	1100	2.75	3.48	6.24	9.69	89.2	57.0	3.85
16	1600	4.59	3.31	7.90	8.11	92.7	47.7	4.49

Table 2: Data for MPI run after 100 iterations on NPACI-SP2

P	**N**	**Comm**	**Calc**	**Total**	**Mflops/P**	**% R**	**% Eff**	**MB/s**
1	400	0.050	0.839	0.889	71.98	-	100	-
4	800	0.115	0.828	0.943	67.84	98.0	94.2	44.86
8	1100	0.460	0.800	1.260	48.00	92.9	66.7	23.05
16	1600	0.289	0.834	1.123	57.00	98.2	79.2	71.35
32	1600	0.729	0.872	1.601	41.24	97.6	57.3	60.87

4 Discussion

The code listing and the table of comparisons along with corresponding graphs are included with this report. The contour plots for the 500 and

Table 3: Data for HPF run after 100 iterations on PSU-SP2

P	**N**	**Comm**	**Calc**	**Total**	**Mflops/P**	**% R**	**% Eff**
1	400	6.35	5.36	11.73	5.46	-	100
8	1100	8.06	5.15	13.26	4.57	97.2	83.6
16	1600	10.62	5.39	16.07	3.99	97.5	73.0
32	2300	8.67	5.81	14.52	4.55	99.4	83.4

Table 4: Comparison table for HPF run vs MPI run after 100 iterations

Case	**P**	**HPF Timing**	**MPI Timing**	**% Gain**
1	1	11.73	3.77	311
2	4	-	5.61	-
3	8	13.26	6.24	213
4	16	16.07	7.90	203
5	32	14.52	-	-

1000 iteration cases (for 800×800 grid) are also shown. The 1 iteration case is omitted as nothing wortwhile appears in the plot and it reflects just the boundary conditions which are not clearly seen.

From the PSU-SP2 run, the code appears to be approximately 90% parallel (which seems to be increasing slowly with the increasing in the number of nodes). Since the problem size is not fixed, it is not expected that Amdahl's law will be followed. It is also seen that the parallel fraction of the code seems to be increasing with the increase in the number of processors, which is an indication of the Gustafson's law being followed. However, Gustafson's law is also not completely followed as the parallel efficiency seems to be going down with the number of processors rather than remaining constant. One of the possible reasons is that the communication to computation ratio is not fixed in all the runs. The 8 processor case uses a 4×2 grid of processors which causes communication workload on each node to be more than for the $2 \times 2 = 4$ node run or the $4 \times 4 = 16$ node run. It is also observed that although the computational time remains almost constant in each case (≈ 3.6 seconds for the PSU-SP2 and ≈ 0.83 seconds for the NPACI-SP2), the communication time seems to be rising thus affecting the efficiency. Thus the trend from this minimal data is seen to be somewhat between Amdahl's law and Gustafson's law. More runs on higher number of processors are recommended for a better picture of the trend.

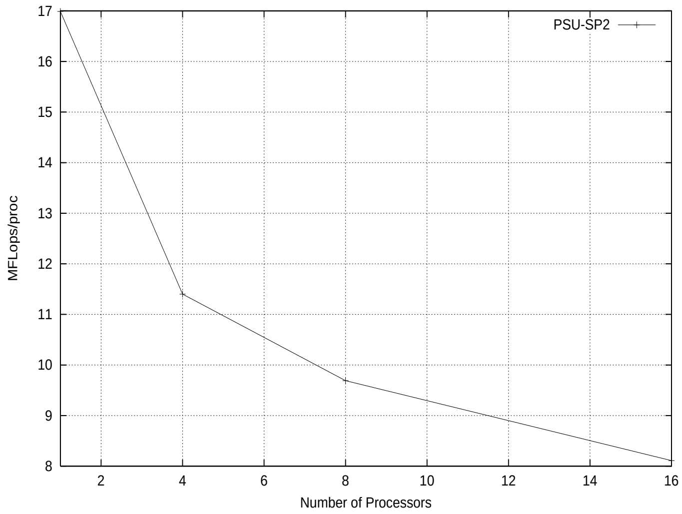

Figure 1: MFLOPs/proc vs Number of Processors

The comparison between HPF and MPI clearly indicates MPI to be much superior to HPF code. The MPI run not only has better computational time (40% faster) for the same run, but also has offers better communication times (which seem to differ only by a constant for each run).

The MFLOPs per processor for the scaled problem case is seen to be decreasing but averages around 10 (4.0% of peak) for the PSU-SP2 and 50 (7.8% of peak) for the NPACI-SP2. This is somewhat in contradiction to the HPF run on the PSU-SP2 in which the MFLOPs appears to be more or less constant around 5 for the scaled problem case.

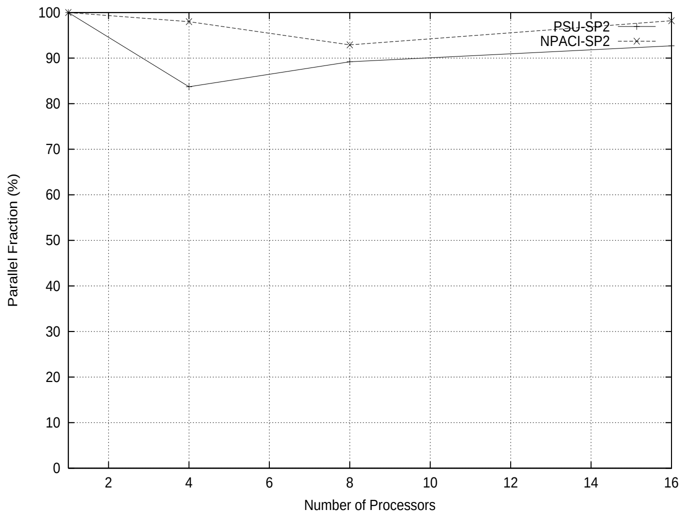

Figure 2: Parallel fraction vs Number of Processors

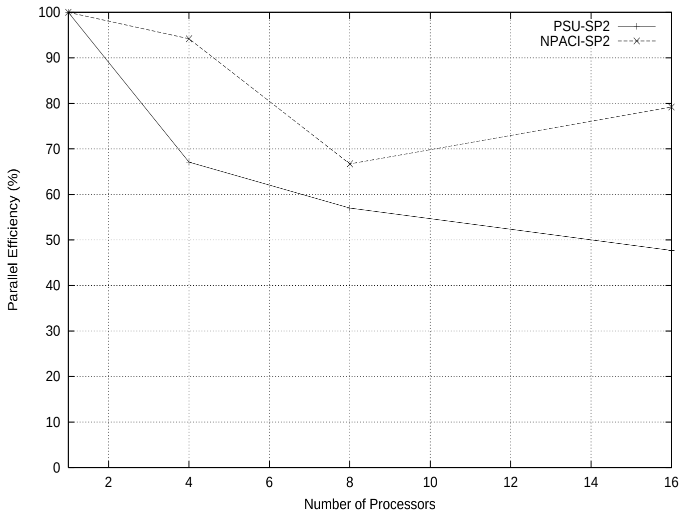

Figure 3: Parallel efficiency vs Number of Processors

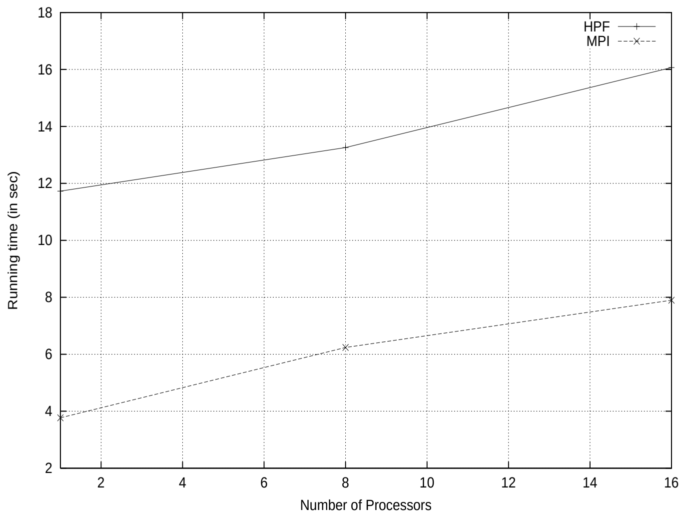

Figure 4: Execution time comparison for MPI vs HPF case

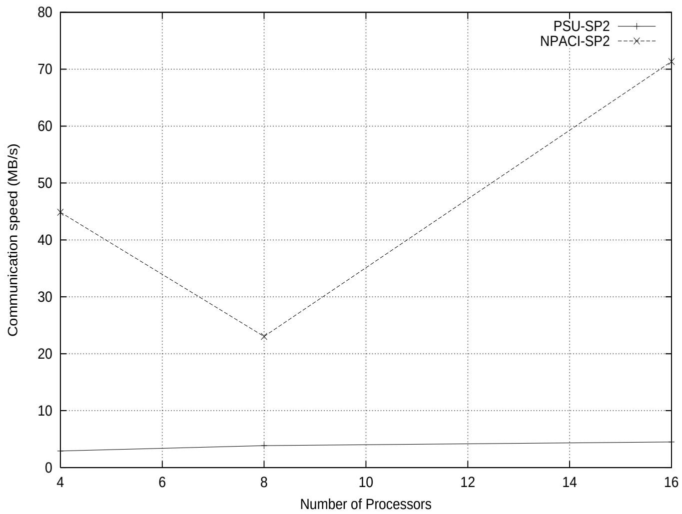

Figure 5: Communication speed vs Number of Processors

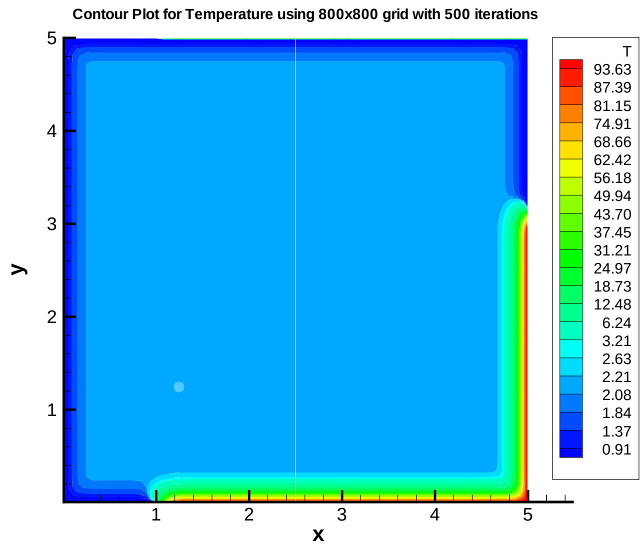

Figure 6: Temperature contours after 500 iterations

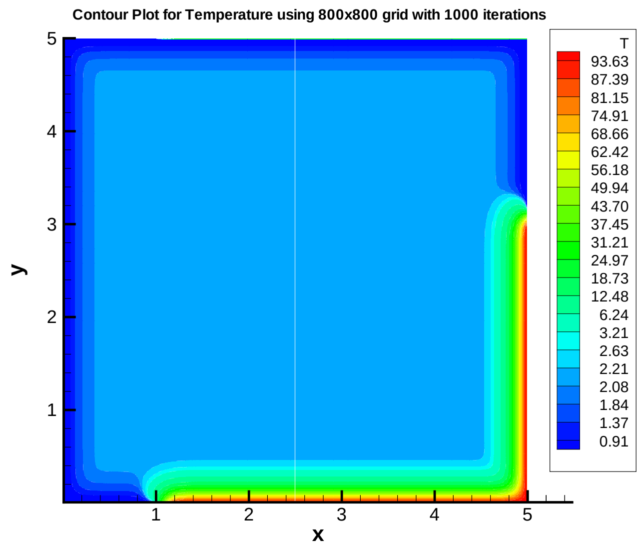

Figure 7: Temperature contours after 1000 iterations