Parallel Numerics '05, 47-56 M. Vajteršic, R. Trobec, P. Zinterhof, A. Uhl (Eds.)
Chapter 3: Differential Equations ISBN 961-6303-67-8

Parallel Numerical Solution of 2-D Heat Equation

Verena Horak*, Peter Gruber

Department of Scientific Computing,
University of Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria

In this paper, we will discuss the numerical solution of the two dimensional Heat Equation. An approximation to the solution function is calculated at discrete spatial mesh points, proceeding in discrete time steps. The starting values are given by an initial value condition. We will first explain how to transform the differential equation into a finite difference equation, respectively a set of finite difference equations, that can be used to compute the approximate solution. We will then modify this algorithm in order to parallelize this task on multiple processors. Special focus is given on the performance respectively performance improvement of a parallelized algorithm on different hardware platforms. Additionally we will run the implemented algorithm on two different clusters and calculate speedup based on the execution time of 1 to 32 CPUs.

1 Introduction

In this paper, we will describe how to solve a 2-dimensional differential equation using parallel algorithms. We will illustrate the procedure with a concrete example, namely the so called "Heat Equation"

$$u_t = c \cdot (u_{xx} + u_{yy}),\ 0 \leq x, y \leq 1,\ t \geq 0,$$

with initial and boundary conditions

$$\begin{aligned} u(0, x, y) &= f(x, y), \\ u(t, 0, y) &= \alpha_0(y), \end{aligned}$$

*Corresponding author. E-mail: vhorak@cosy.sbg.ac.at

$$\begin{aligned} u(t,1,y) &= \alpha_1(y), \\ u(t,x,0) &= \beta_0(x), \\ u(t,x,1) &= \beta_1(x)\ . \end{aligned}$$

The solution function $u(t,x,y)$ of this differential equation describes the temperature of, for example, a thin metal plate of area 1, at every position (x,y), $0 \leq x,y \leq 1$, at any time $t \geq 0$. At the edge points of the plate, we have constant temperatures $\alpha_0(y)$, $\alpha_1(y)$, $\beta_0(x)$ and $\beta_1(x)$. At time $t = 0$, the temperature of every point (x,y) is given by $f(x,y)$. In the most simple case, we may assume that $f(x,y)$ is a constant function with its value somehow in the range defined by the boundary conditions.
For example, the temperature at position $(x = 0.3, y = 0.5)$ at time $t = 10.4$ is given by $u(10.4, 0.3, 0.5)$.
In this paper, we try to approximate the values of the solution function $u(t,x,y)$. This means, we will consider discrete time points $t_0 = 0, t_1, \ldots, t_k, \ldots$ and discrete positions (x_i, y_j), $0 \leq i,j \leq (n+1)$ and compute the values of $u(t_k, x_i, y_j)$, $0 \leq i,\ j \leq n+1,\ k \geq 0$.
The differential equation will be replaced by a finite difference equation in the solution process. With this method we can calculate the approximated values $u(t_k, x_i, y_j)$ by proceeding from time point t_{k-1} to time point t_k, if approximate values at time t_{k-1} are already known for all positions (x_i, y_j).

2 Solution Method

We want to approximate the solution function $u(t,x,y)$ at discrete points (t_k, x_i, y_j). Thus, we will first of all define spatial mesh points

$$(x_i, y_j) = (i \cdot \Delta s,\ j \cdot \Delta s),\ i,j = 0,1,2 \ldots, n+1\ ,$$

where $\Delta s = \frac{1}{n+1}$, and temporal mesh points

$$t_k = k \cdot \Delta t,\ k = 0,1,2,\ldots\ ,$$

for suitably chosen Δt. (Note that in principle there is no upper boundary for time points t_k!)
With respect to computational effort and exactness of the approximation, n and Δt may be chosen arbitrarily. In particular, one has to secure that the numerical method will be stable. This means, that small perturbations e.g. resulting from rounding errors do not cause the resulting numerical solutions to diverge from each other without bound.

For the Heat Equation, we know from theory that we have to obey the restriction

$$\Delta t \leq \frac{(\Delta s)^2}{2c}$$

in order for the finite difference method to be stable.
The solution function $u(t, x, y)$ represents the temperature at point (x, y) at time t. The area in question is the unit square, so that we can discretize this area using a $(n+2) \times (n+2)$-matrix U^k, where the entries u_{ij}^k correspond to the temperature value at the point (x_i, y_j) (for $0 \leq i,\ j \leq n+1$) at time point t_k (for $t \geq 0$).
In order to calculate the approximate solution, we will replace any derivative by finite differences. In particular, for any point (t_k, x_i, y_j) we will set

$$u_t \ \approx \ \frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t}$$

and

$$u_{xx} \ \approx \ \frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{(\Delta s)^2} \ , \ u_{yy} \ \approx \ \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}k}{(\Delta s)^2} \ .$$

So the original differential equation

$$u_t = c \cdot (u_{xx} + u_{yy})$$

becomes

$$\frac{u_{i,j}^{k+1} - u_{i,j}^k}{\Delta t} \ = \ c \cdot \left(\frac{u_{i+1,j}^k - 2u_{i,j}^k + u_{i-1,j}^k}{(\Delta s)^2} + \frac{u_{i,j+1}^k - 2u_{i,j}^k + u_{i,j-1}^k}{(\Delta s)^2} \right) \ .$$

Expressing $u_{i,j}^{k+1}$ from this equation yields

$$u_{i,j}^{k+1} = u_{i,j}^k + c \cdot \frac{\Delta t}{(\Delta s)^2} \left(u_{i+1,j}^k + u_{i-1,j}^k - 4u_{i,j}^k + u_{i,j+1}^k + u_{i,j-1}^k \right) \ ,$$

where $\Delta s = \frac{1}{n+1}$ and Δt is set appropriately. We will consider this equation for points (t_k, x_i, y_j) with $k \geq 1$ and $1 \leq i \leq n$. For $k = 0$, the solution is given by the initial value condition $u(0, x, y) = f(x, y)$, and for $i = 0$ or $i = n + 1$ the solution is given by the boundary conditions $u(t, 0, y) = \alpha_0(y)$, $u(t, 1, y) = \alpha_1(y)$, $u(t, x, 0) = \beta_0(x)$ and $u(t, x, 1) = \beta_1(x)$.
In particular, the values for u_{0j}^k, $0 \leq j \leq n+1$, $u_{(n+1),j}^k$, $0 \leq j \leq n+1$, u_{i0}^k, $0 \leq i \leq n+1$, $u_{i,(n+1)}^k$, $0 \leq i \leq n+1$ remain constant for all k (that is for all time points t_k) and have to be set initially according to the boundary

conditions.
The starting values u_{ij}^0 for all inner grid points are given by an initial value condition.

The implementation on a single processor is quite straight forward:

```
double[n+2][n+2] u_old, u_new;
double c, delta_t, delta_s;
Initialize u_old, u_new with initial values and boundary
conditions;

while (still time points to compute) {
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      u_new[i, j] = u_old[i, j] + c * delta_t/delta_s^2 *
           (u_old[i+1, j] + u_old[i-1, j] - 4*u_old[i, j] +
            u_old[i, j+1] + u_old[i, j-1]);
    } //end of for
  } // end of for
  u_old = u_new;
} // end of while
```

We can parallelize this algorithm by one dimensional data domain distribution along the y-axis among multiple processors. Each participating processor will thus be responsible for a submatrix of the complete (discretized) unit square. For the sake of simplicity, let's assume that the number of processors m is a divisor of the number of spatial mesh points in each direction, $n+2$. Every processor will be responsible for a set of $\frac{n+2}{m} \cdot (n+2)$ points; this means in particular, processor number p will compute values of points $(x_{\frac{p\cdot(n+2)}{m}}, y_j), (x_{\frac{p\cdot(n+2)}{m}+1}, y_j), \ldots, (x_{\frac{p\cdot(n+2)}{m}+\frac{n+2}{m}-1}, y_j)$.
(Note that in the actual implementation there is some mechanism to handle the case that m is not a divisor of $n+2$.)
Evidently, some communication between the processors will be necessary, as some values from neighboring processors are needed for computation in the next iteration. In order to implement this structure, every process has two local submatrices *uold* and *unew* of size $(\frac{n+2}{m}+2) \times (n+2)$. Thus, the first as well as the last entry represents the values of a neighboring process but values have only to be computed for all inner rows of this matrix.
We have to handle two exceptional cases here: for the first as well as for the last processor there is either no left or no right neighbor. Thus, either the first or the last row of the local matrix remains unused, which poses no problem. Furthermore, either the first or the last row is given by boundary conditions

and so there is no need to be updated with every iteration. We will handle this exceptions by introducing two integer variables *ifirst* and *ilast* that define the first and last row to be updated in each iteration. Usually, *ifirst* will be set to 1, and *ilast* will be set to $\frac{n+2}{m}$, that is the second and last but one row of the local matrices. For the first and last process, either *ifirst* or *ilast* are reset accordingly.
The concept of the implemented algorithm is shown below. Note that the actual implementation needs some extensions in order to cope with the situations that the number of rows is not a multiple of the number of processors.

```
double[(n+2)/m + 2, n+2] u_old, u_new;
double c, delta_t, delta_s;
int ifirst, ilast; //first/last index corresponding to x-coord.
int myid; // number of processor
double f(index i,index j);
  // function that gives initial values for every point x_i;
  // boundary conditions have to be considered for i,j = 0 or n+1
Set delta_s to 1/(n+2) and c, delta_t appropriately;
// Initialize u_old with initial values and boundary conditions;
// set and eventually reset ifirst and ilast
ifirst = 1;
ilast = (n+2)/m;
for (int i = ifirst-1; i <= ilast+1;i++) {
  for (int j = 0; j <= n+1; j++){
    uold[i, j] = f(myid*(n+2)/m + i -1, j);
  }
}
if (myid == 0) {ifirst++;}
if (myid == m-1) {ilast--;}
while(still time points to compute) {
  for (int i = ifirst; i <= ilast;i++) {
    for (int j = 1; j <= n+1; j++) {
      u_new[i, j] = u_old[i, j] + c * delta_t/delta_s^2 *
                  ( u_old[i+1, j] + u_old[i-1, j] - 4*u_old[i, j]
                  + u_old[i, j-1] + u_old[i, j+1]);
  }} // end of for
  if (myid < m-1) {
    Send values for row u_{myid*(n+2)/m+(n+2)/m-1}=u_new[(n+2)/m]
    to processor number myid + 1;
    Get values for row u_{myid*(n+2)/m+(n+2)/m} from processor no.
    myid + 1 and write value to u_new[(n+2)/m + 1];
```

```
  }
  if (myid > 0) {
    Send values for row u_{myid*(n+2)/m} = u_new[1]
    to processor number myid - 1;
    Get values for row u_{myid*(n+2)/m - 1} from processor number
    myid - 1 and write value to u_new[0];
  }
  u_old = u_new;
} // end of while
```

3 Obtained Results

We will present the calculated solutions for the two dimensional Heat Equation $u_t = c \cdot (u_{xx} + u_{yy})$ with graphical representations. We will always set the constant c to be 0.1. Furthermore, we will consider different numbers $n+2$ of spatial mesh points and different numbers p of processors.
Computed solutions for the values of the spatial mesh points (x_i, y_j) for a temporal mesh point t_k are independent on the number of processors used for its calculation. Thus, we will present the numerical solutions with no relation to the number of processors used, but compare the calculation time needed with different numbers of processors used. In particular, we will analyze speedup and efficiency on different hardware platforms. In this context, "different processors" actually means that the processes run indeed on different machines.
Note that for the Heat Equation not only the final (static) solution is usually of interest, but also the way the temperature field changes from the initial values as time proceeds. In other words, initial values for this problem are not only chosen in order to minimize the effort to calculate the final solution, but represent an essential part of the practical problem.
As different choices of n lead to different values for Δs, it is also sensible to choose different values for Δt. In our case, we will always set $\Delta t = \frac{(\Delta s)^2}{4c}$ in order to guarantee stability.

In the example presented here, initial values of all inner grid points will be set to zero. Boundary conditions are constant values at the edges.
Calculation will be stopped after 10000 iterations. We will list the computation time required for various numbers of grid points and participating processors.

Testing the performance of the algorithm on the network of the Jožef Stefan Institute, Ljubljana, we got the results listed in table 1. The computer cluster of Jožef Stefan Institute is composed of 16 AMD Opteron 244 dual processors

	n = 3721	n = 34225	n = 319225	n = 3200512
p = 1	0.449	4.637	61.390	569.551
p = 2	0.867	3.133	38.105	410.633
p = 4	1.855	3.746	24.910	202.648
p = 8	2.949	4.203	20.090	108.445
p = 16	4.082	4.972	12.500	73.508
p = 32	5.664	5.930	9.695	52.438

Table 1: Computation time (Jožef Stefan Institute, Ljubljana)

	n = 3721	n = 34225	n = 319225	n = 3200512
$s_{1,2}$	0.518	1.480	1.611	1.387
$s_{1,4}$	0.242	1.238	2.464	2.811
$s_{1,8}$	0.152	1.103	3.056	5.252
$s_{1,16}$	0.110	0.932	4.911	7.748
$s_{1,32}$	0.079	0.782	6.332	10.862

Table 2: Calculated Speedup (Jožef Stefan Institute, Ljubljana)

with Linux Fedora 2 operating system. The processing nodes are connected in a two dimensional toroidal 4-mesh by Gigabit Ethernet. Our reference implementation uses the programing language C and MPICH implementation of MPI library.

The resulting speedup (Ljubljana) for different values for n is listed in table 2; $s_{i,j}$ denotes the speedup when using j processors instead of i. The speedup is calculated as $s_{i,j} = \frac{t_i}{t_j}$, where t_i denotes the time consumed when using i processors.

Doing the same tests on the "Gaisberg"-cluster of the University of Salzburg, Department of Scientific Computing, led to the results given in tables 3 and 4. This cluster is composed of 36×2 Athlon MP2800+ processors (2 GB RAM) with a RedHat Linux operating system. The nodes are connected as a 6×6 SCI torus ($\approx$ 250 MByte/sek).

A sample temperature field that was evaluated as the stable solution to the Heat Equation can be seen in figure 1. For this illustrating example we chose a field of 30×30 points with initial values randomly between -30 and 30. For boundary conditions we set the left border to 10 and the right one to 40. The points in front were chosen to be 30 and the points in the last line 50.

	n = 3721	n = 34225	n = 319225	n = 3200512
p = 1	1.771	22.631	226.414	2221.875
p = 2	1.090	9.809	114.949	1097.856
p = 4	0.798	4.640	59.113	557.677
p = 8	0.648	2.784	29.917	282.519
p = 16	0.641	1.857	13.302	145.271

Table 3: Computation time (University of Salzburg)

	n = 3721	n = 34225	n = 319225	n = 3200512
$s_{1,2}$	1.624	2.307	1.970	2.024
$s_{1,4}$	2.219	4.877	3.830	3.984
$s_{1,8}$	2.733	8.129	7.568	7.865
$s_{1,16}$	2.762	12.185	17.021	15.295

Table 4: Calculated Speedup (University of Salzburg)

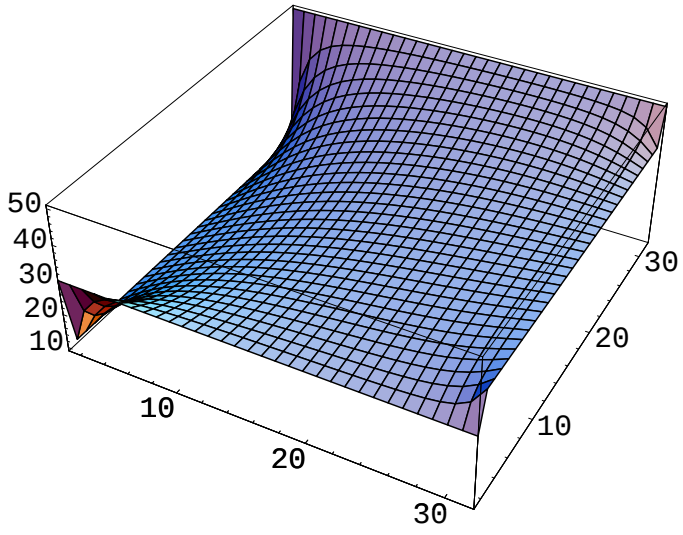

Figure 1: Solution of Heat Equation after 2478 iteration steps

4 Conclusions

As the example presented in the previous section illustrates, the results of this paper confirmed the thesis that good numerical approximations to the solution of the two dimensional Heat Equation can be obtained using finite difference method. One would expect this from theory; the outcome of our work demonstrates, however, that good results can be obtained even with comparably little effort on choosing suitable parameters, for example.

The project showed that finite difference algorithms are well suited for parallel programming. The solution for the single processor case can be transferred with only minor modifications to the multi processor case. We discussed this transformation in detail in the previous section "Solution Method".

A common purpose in using parallel algorithms is to reduce computation time needed to calculate the solution. In our case, one can observe that using multiple processors does sometimes indeed increase the computation time needed. A probable explanation might be that for this special problem, the (additional) time needed for communication between the processors is almost huge compared to the time needed to perform the actual computations. Actually, the processors need to synchronize after every iteration as they depend on the values of neighboring processes for the next iteration. The computational effort for each iteration is quite low, but a lot of iterations might be necessary. The effect of using multiple processors in parallel is much more significant when increasing the number of points to be evaluated, probably due to the bigger computational effort each processor has to carry in every iteration. This confirms the assumption that parallelizing an algorithm saves time only if there has to be done a considerable amount of calculations in each iteration, as otherwise too much time is needed for communication between participating processors in relation to the actual computational work being done.
For further studies, one could use this algorithm for more complex operations in each iteration, such that the computation time needed for each iteration gets bigger in relation to the time needed for communication. We would expect that the parallel algorithm needs less time than the single processor-algorithm in this case.

Acknowledgment

We would like to give a special thanks to Roman Trobec for his help and encouragement during the whole work. We also want to thank the Institute Jožef Stefan, Ljubljana, Slovenia, and the Department of Scientific Computing, Salzburg, Austria, for granting access to their Computer Clusters as well as Ernst Forsthofer and Stefan Jenisch for technical support.

References

[1] M. T. Heath: *Scientific Computing: An Introductory Survey*, Second Edition, McGraw-Hill, New York 2001.

[2] M. Praprotnik, M. Sterk, R. Trobec: *A new explicit numerical scheme for nonlinear diffusion problems, Parallel numerics '02 : theory and applications*, Jozef Stefan Institute and University of Salzburg, 2002, 163-176.

[3] P. Trunk, B. Gersak, R. Trobec: *Topical cardiac cooling - computer simulation of myocardial temperature changes*, Comput. biol. med., vol. 33, 2003, 203-214.

[4] I. Foster: *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison-Wesley Longman Publishing Co., Inc, 1995

Supporting MPI collective communication on network processors

[5] Q. Zhang, C. Keppitiyagama, A. Wagner: *Supporting MPI collective communication on network processors*, IEEE International Conference, 2002, 75 - 82

[6] *MPI: A message passing interface*, Supercomputing '93. Proceedings, 15-19 Nov. 1993, 878 - 883

[7] V. Makarov, R. Chapko: *The Cayley transform and boundary integral equations to an initial boundary value problem for the heat equation*, 1999. Proceedings of IVth International Seminar/Workshop, 20-23 Sept. 1999, 59 - 64