Engineering: Parallel Solution of 1-D Heat Equation

7

Chapter Summary
This chapter presents a simplified model for a computer simulation of changes in the temperature of a thin bar. The phenomena is modelled by a heat equation PDE dependant on space and time. Explicite finite differences in 1-D domain are used as a solution methodology. The paralelization of the problem, on both OpenMP and MPI, leads to a program with significant communication requirements. The obtained experimental measurements of program run-time are analysed in order to optimize performances of the parallel program.

Partial differential equations (PDE) are a useful tool for the description of natural phenomena like heat transfer, fluid flow, mechanical stresses, etc. The phenomena are described with spatial and time derivatives. For example, a temperature evolution in a thin isolated bar of length L, with no heat sources, can be described by a PDE of the following form:

$$\frac{\partial T(x,t)}{\partial t} = c\frac{\partial^2 T(x,t)}{\partial x^2}$$

where $T(x,t)$ is an unknown temperature at position x in time t, and c is a thermal diffusivity constant with typical values for metals being about $10^{-5}\,\mathrm{m}^2/\mathrm{s}$. The PDE says that the first derivative of temperature T by t is equal to the second derivative of T by x. To fully determine the solution of the above PDE, we need the initial temperature of the bar, a constant T_0 in our simplified case:

$$T(x,0) = T_0.$$

Finally, fixed temperatures, independent of time, at both ends of the bar are imposed as $T(0) = T_L$ and $T(L) = T_R$.

In the case of strong solution methodology, the problem domain is discretized in space and the derivatives are approximated by, e.g., finite differences. This results in

© Springer Nature Switzerland AG 2018

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_7

a sparse system matrix A, with three diagonals in the case of 1-D domain or with five diagonals in 2-D case. To save memory, and because the matrix structure is known, the vectors with old and new temperatures are only needed. The evolution in time can be obtained by an explicit iterative calculation, e.g., Euler method, based on the extrapolation of the current solution and its derivatives into the next time-step, respecting the boundary conditions. A developing solution in time can be obtained by a simple matrix-vector multiplication. If only a stationary solution is desired, then the time derivatives of the solution become zero, and the solution can be obtained in a single step, through a solution of the resulting linear system, $Au = b$, where A is a system matrix, u is a vector of unknown solution values in discretization nodes, and b is a vector of boundary conditions.

We will simplify the problem by analyzing 1-D domain only. Note that an extension in 2-D domain, i.e., a plate, is quite straightforward, and can be left for a mini project. For our simplified case, an isolated thin long bar, an exact analytic solution exists. Temperature is spanning in a linear way between both fixed temperatures at boundaries. However, in real cases, with realistic domain geometry and complex boundary conditions, the analytic solution may not exist. Therefore, an approximate numerical solution is the only option. To find the numerical solution, the domain has to be discretized in space with $j = 1 \ldots N$ points. To simplify the problem, the discretization points are equidistant, so $x_{j+1} - x_j = \Delta x$ is a constant. Discretized temperatures $T_j(t)$ for $j = 2 \ldots (N-1)$ solution values in inner points and $T_1 = T_L$ and $T_N = T_R$ are boundary points with fixed boundary conditions.

Finally, we also have to discretize time in equal time-steps $t_{i+1} - t_i = \Delta t$. Using finite difference approximations for time and spatial derivatives:

$$\frac{\partial T(x_j, t)}{\partial t} = \frac{T_j(t_{i+1}) - T_j(t_i)}{\Delta t} \quad \text{and} \quad \frac{\partial^2 T(x, t)}{\partial x^2} = \frac{T_{j-1}(t_i) - 2T_j(t_i) + T_{j+1}(t_i)}{(\Delta x)^2},$$

as a replacement of derivatives in our continuous PDE, provides one linear equation for each point x_j. Using explicit Euler method for time integration, we obtain, after some rearrangement of factors, a simple algorithm for calculation of new temperatures $T(t_{i+1})$ from old temperatures:

$$T_j(t_{i+1}) = T_j(t_i) + \frac{c\,\Delta t}{(\Delta x)^2}\left(T_{j-1}(t_i) - 2T_j(t_i) + T_{j+1}(t_i)\right).$$

In each discretization point, a new temperature $T_j(t_{i+1})$ is obtained by summing the old temperature with a product of a constant factor and linear combination of temperatures in three neighboring points. After we set the initial temperatures of inner points and fix the temperatures in boundary points, we can start marching in time to obtain the updated values of the bar temperature.

Note that in the explicit Euler method, thermal conductivity c, spatial discretization Δx, and time-step Δt must be in an appropriate relation that fulfils CFL stability condition, which is for our case: $c\,\Delta t/(\Delta x)^2 \leq 0.5$. The CFL condition could be informally explained with a fact that a numerical method has to step in a time slower than the simulated physical phenomenon. In our case, the impact of a change in discretization point temperature is at most in neighboring discretization points. Hence,

with smaller Δx (denser discretization points), shorter time-steps are required in order to correctly capture the simulated diffusion of temperature.

When we have to stop the iteration? Either after a fixed number of time-steps nt, or when the solution achieves a specified accuracy, or if the maximum difference between previous and current temperatures falls below a specified value. A sequential algorithm for an explicit finite differences solution of our PDE is provided below:

Algorithm 3 SEQUENTIAL ALGORITHM: 1- D_HEAT_EQUATION

Input: err - desired accuracy;
N - number of discretization points
T_0 - initial temperature
T_L and T_R - boundary temperatures

1: Discretize domain
2: Set initial solution vectors $\mathbf{T}_i$ and $\mathbf{T}_{i+1}$
3: **while** Stopping criteria NOT fulfilled **do**
4: **for** $j = 2 \ldots (N-1)$ **do**
5: Calculate new temperature $T_j(t_{i+1})$
6: **end for**
7: **end while**

Output: **T** - Approximate temperature solution in discretization points.

We start again with the validation of our program, on a notebook with a single MPI process and with the code from Listing 7.2. The test parameters were set as follows: $p = 1$, $N = 30$, $nt = [1, \ldots, 1000]$, $c = 9\in$ - 3, $T_0 = 20$, $T_L = 25$, $T_R = 18$, $time = 60$, $L = 1$. Because the exact solution is known, i.e., a line between T_L, T_R, the maximal absolute error of the numerical solution was calculated to validate the solution behavior. If the model and numerical method are correct, the error should converge to zero.The resulting temperatures evolution in time, on the simulated bar, are shown in Fig. 7.1.

The set of curves in Fig. 7.1 confirms that the numerical method produces in initial time-steps a solution near to initial temperature. Then, the simulated temperatures change as expected. While the number of time-steps nt increases, the temperatures advance toward the correct result, which we know that is a straight line between left and right boundary temperatures, i.e., in our case, between 25° and 18°.

We have learned, that in the described solution methodology, the calculation of a temperature T_j in a discretization point depends only on temperatures in two neighboring points, consequently, the algorithm has a potential to be parallelized. We will again use domain decomposition, however, the communication between processes will be needed in each time-step, because the left and right discretization point of a sub-domain is not immediately available for neighboring processes. A point-to-point communication is needed to exchange boundaries of sub-domains. In 1-D case, the discretized bar is decomposed into a number of sub-domains, which are equal to the number of processes. All sub-domains should manage a similar number

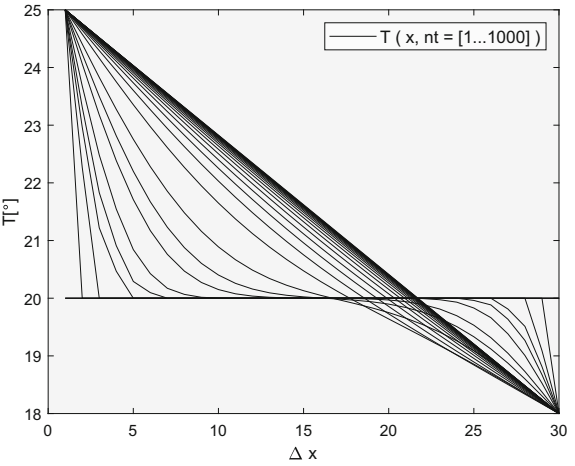

Fig. 7.1 Temperature evolution in space and time as solved by heat equation

of points, because an even load balance is desired. Some special treatment is needed for calculation near the domain boundaries.

Regarding the communication load, some collective communication is needed at the beginning and end of the calculation. Additionally, a point-to-point communication is needed in each time-step to exchange the temperature of sub-domain border points. In our simplified case, the calculation will stop after a predefined number of time-steps, hence, no communication is needed for this purpose. The parallelized algorithm is shown below:

Algorithm 4 PARALLEL ALGORITHM: 1- D_HEAT_EQUATION

Input: err - desired accuracy;
N - number of discretization points
T_0 - initial temperature
T_L and T_R - boundary temperatures

1: Get $myID$ and the number of cooperating processes p
2: Master broadcast *Input* parameters to all processes
3: Set local solution vectors $\mathbf{T}_{ip}$ and $\mathbf{T}_{ip+1}$
4: **while** Stopping criteria NOT fulfilled **do**
5: **for** $j = 1 \ldots N/p$ **do**
6: Exchange $T_j(t_i)$ of sub-domain border points
7: Calculate new temperature $T_j(t_{ip+1})$
8: **end for**
9: Master gather sub-domain temperatures as a final result $\mathbf{T}$
10: **end while**

Output: $\mathbf{T}$ - Approximate temperature solution in discretization points.

We see that in the parellelized program several tasks have to be implemented, i.e., user interface for input/output data, decomposition of domain, allocation of memory for solution variables, some global communication, calculation and local communication in each time-step, and assembling of the final result. Some of the tasks are inherently sequential and will, therefore, limit the speedup, which will be more pronounced in smaller systems.

The processes are not identical. We will again use a master process and several slave processes, two of them responsible for domain boundary. The master process will manage input/output interface, broadcast of solution parameters and gathering of final results. The analysis of parallel program performances for: OpenMP, and MPI, are described in the following sections.

7.1 OpenMP

Computing 1-D heat transfer using a multicore processor is simple if Algorithm 3 is taken as the starting point for parallelization. As already explained above, iterations of the inner loop are independent (while the iterations of the outer loop must be executed one after another). The segment of the OpenMP program implementing Algorithm 3 is shown in Listing 7.1: `Told` and `Tnew` are two arrays where the temperatures computed in the previous and in the current outer loop iteration are stored; `C` contains the constant $c\Delta t/(\Delta x)^2$.

```
#pragma omp parallel firstprivate(k)
  {
    double *To = Told;
    double *Tn = Tnew;
    while (k--) {
      #pragma omp for
      for (int i = 1; i <= n; i++) {
        Tn[i] = To[i]
          + C * (To[i - 1] - 2.0 * To[i] + To[i + 1]);
      }
      double *T = To; To = Tn; Tn = T;
    }
  }
```

Listing 7.1 Computing heat transfer in one dimension by OpenMP.

It is worth examining the wall clock time of this algorithm first. Figure 7.2 summarizes the wall clock time needed for 10^6 iterations in 10^5 points using a quad-core processor with multithreading. With more than 4 threads nothing is gained in terms of a wall clock time. As this is a floating-point-intensive application run on a processor with one floating-point unit per physical core, this can be expected: two threads running on two logical cores of the same physical core must share the same floating point unit. This leads to more synchronizing among threads and, consequently, to the increase of the total CPU time as shown in Fig. 7.3.

The interested reader might investigate how the wall clock time and the speedup change if the ratio between the number of points along the bar and the number

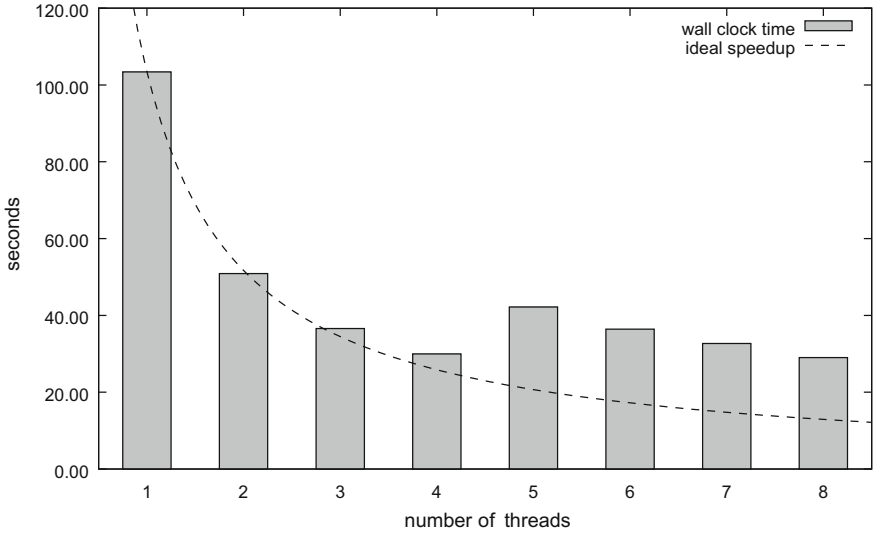

Fig. 7.2 The wall clock time needed to compute 10^6 iterations of 1-D heat transfer along a bar in 10^5 points

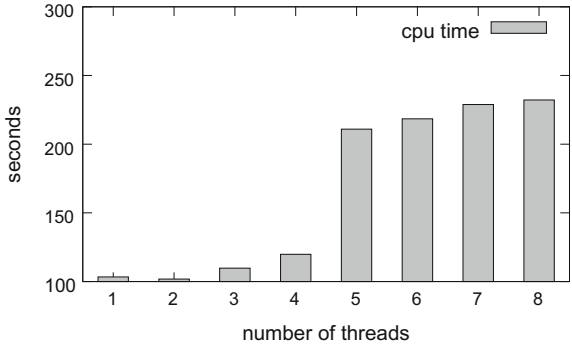

Fig. 7.3 The total CPU time needed to compute 10^6 iterations of 1-D heat transfer along a bar in 10^5 points

of iterations in time change. Namely, decreasing the number of points along the bar makes the synchronization at the barrier at the end of the parallel `for` loop relatively more expensive.

7.2 MPI

In the solution of heat equation, the problem domain is discretized first. In our simplified case, a temperature diffusion in a thin bar is computed, which is modeled by 1-D domain. For efficient use of parallel computers, the problem domain must be partitioned (decomposed) into possibly equal sub-domains. The number of sub-domains should be equal to the number of MPI processes p. We prescribe a certain

number of discretization points per process N_p, which automatically guarantees a balanced computational load. Note that the total number of discretization points N scales with the number of processes. In all active processes, an appropriate amount of memory is allocated for current and new solution vectors and initialized with initial and boundary values. Then, the CFL stability condition is verified.

In each time-step, every process that computes its sub-domain, exchanges sub-domain border temperatures with processes that compute its left-end right sub-domains. In our implementation, blocking communication is used that can adequately maintain short messages. Then, new temperatures are calculated for all discretization points in each sub-domain, by the methodology explained at the beginning of this chapter. We determine a fixed number of time-steps, and therefore, a special stopping criterion is not needed.

An exemplar MPI program implementation of a solution of 1-D heat equation for our simplified case is given in Listing 7.2.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "mpi.h"

void solve(int my_id, int num_p);

int main(int argc, char *argv[])
{
  int my_id, num_p;
  double start, end;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
  MPI_Comm_size(MPI_COMM_WORLD, &num_p);
  if (my_id == 0)
    start = MPI_Wtime();
  solve(my_id, num_p);
  if (my_id == 0)
  printf("Elapsed seconds = %f\n", MPI_Wtime() - start);
  MPI_Finalize();
  return 0;
}
void solve(int my_id, int num_p)  //compute time step T
{
  double cfl, *T, *T_new;
  int i, j, tag, j_min = 0;
  int j_max = 10000;  //number of time-steps
  int N = 5000;  //number of points per process
  double c = 1e-11;  //diffusivity
  double T_0 = 20.0, T_L = 25.0, T_R = 18.0; //temperatures
  double time, time_new, delta_time, time_min = 0.0, time_max = 60.0;
  double *X, delta_X, X_min = 0.0, Length = 0.1;
  MPI_Status status;

  if (my_id == 0)
  {
    printf("  dT/dt = c * d^2T/dx^2  for %f < x < %f\n",X_min,Length);
    printf("  and %f < t <= %f.\n", time_min, time_max);
    printf("  space discretized by %d equidistant points \n",num_p*N);
    printf("  each processor works on %d points!!\n", N);
    printf("  time discretizted with %d equal time-steps.\n", j_max);
    printf("  number of cooperating processes is %d\n", num_p);
  }
```

```
  //allocate local buffers and calculate new temperatures
  X = (double *)malloc((N+2)*sizeof(double)); //N+2 point coordinates
  for (i = 0; i <= N + 1; i++)    //ghost points are in X[0] and X[N+1]
  {
    X[i] = ((double)(my_id * N + i - 1) * Length
      + (double)(num_p* N - my_id * N - i) * X_min)
      / (double)(num_p* N - 1);
  }
  T = (double *)malloc((N + 2) * sizeof(double));  //allocate
  T_new = (double *)malloc((N + 2) * sizeof(double));
  for (i = 1; i <= N; i++)
    T[i] = T_0;
  T[0] = 0.0; T[N + 1] = 0.0;
  delta_time = (time_max - time_min) / (double)(j_max - j_min);
  delta_X = (Length - X_min) / (double)(num_p* N - 1);

  cfl = c * delta_time / pow(delta_X,2); //check CFL
  if (my_id == 0)
    printf("  CFL stability condition value = %f\n", cfl);
  if (cfl >= 0.5)
  {
    if (my_id == 0)
      printf("  Computation cancelled: CFL condition failed.\n");
    return;
  }
  for (j = 1; j <= j_max; j++)  //compute T_new
  {
    time_new = ((double)(j - j_min) * time_max
      + (double)(j_max - j) * time_min)
      / (double)(j_max - j_min);
    if (0 < my_id)    //send T[1] to my_id-1 //replace with SendRecv?
    {
      tag = 1;
      MPI_Send(&T[1], 1, MPI_DOUBLE, my_id-1, tag, MPI_COMM_WORLD);
    }
    if (my_id < num_p - 1)    //receive T[N+1] from my_id+1.
    {
      tag = 1;
      MPI_Recv(&T[N+1],1,MPI_DOUBLE,my_id+1,tag,MPI_COMM_WORLD,&status);
    }
    if (my_id < num_p - 1)  //send T[N] to my_id+1
    {
      tag = 2;
      MPI_Send(&T[N], 1, MPI_DOUBLE, my_id+1, tag, MPI_COMM_WORLD);
    }
    if (0 < my_id)    //receive T[0] from my_id-1
    {
      tag = 2;
      MPI_Recv(&T[0],1,MPI_DOUBLE,my_id-1,tag,MPI_COMM_WORLD,&status);
    }
    for (i = 1; i <= N; i++)  //update temperatures
    {
      T_new[i] = T[i] + (delta_time * c/pow(delta_X,2)) *
                        (T[i-1] - 2.0 * T[i] + T[i+1]);
    }
    if (my_id == 0)  T_new[1] = T_L;  //update boundaries T with BC
    if (my_id == num_p- 1) T_new[N] = T_R;
    for (i = 1; i <= N; i++)  T[i] = T_new[i]; //update inner T
  }
  free(T);
  free(T_new);
  free(X);
  return;
}
```

Listing 7.2 MPI implementation of a solution of 1-D heat equation.

After the successful validation, already presented, the analysis of run-time behavior, again on a two-core notebook computer and on eight cluster computers, has been performed. To fulfil the CFL condition, variables: t, nt, N_p, and c has to be appropriately selected. We set the number of all discretization points to $N = [5e5, 5e4, 5e3]$, and, therefore, the number of points per process N_p is obtained by scaling with the number of processes p. For example, if $N = 5e5$ and $p = 4$, $N_p = 1.25e5$, etc. For accurate timings and balanced communication and computation load nt was set to 1e4. To be sure about CFL condition, constant c is set to $1e$ - 11. Other parameters remain the same as in the PDE validation test. The parallel program run-time (RT) in seconds and speedup (SU), as a function of the number of processes $p = [1, \ldots, 8]$ and discretization points, on a notebook computer, are shown in Fig. 7.4.

The obtained results bring two important messages. First, the maximum speedup is only about two, which is smaller than in the case of π calculation. The explanation for this could be in a smaller computation/communication time ratio. It seems that the time spent on communication is almost the same as on the calculation. Second, the speedup drops significantly on more than 4 MPI processes, below one, which is even more pronounced with smaller number of discretization points. The explanation of such a behavior is in the relative amount of communication, which is performed after each time-step.

Next experiments were performed on eight computing cluster nodes with the same approach as in Chap. 6 and with the same parameters as in the notebook test. In this case, we use `mpirun` parameter `-bind-to core:1`, which appears to be more promising in previous tests. We can expect a high impact of communication load. Even that the messages are short, with just a few doubles, the delay is significant

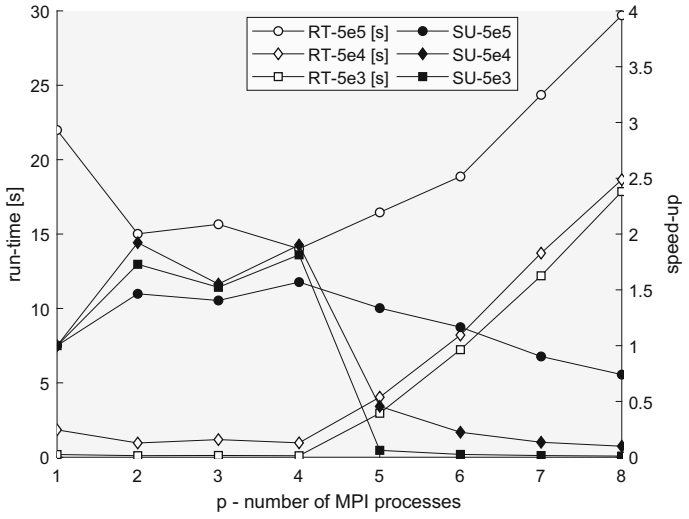

Fig. 7.4 Parallel run-time (RT) and speedup (SU) of heat equation solution for $N = [5e5, 5e4, 5e3]$ and 1 to 8 MPI processes on a notebook computer

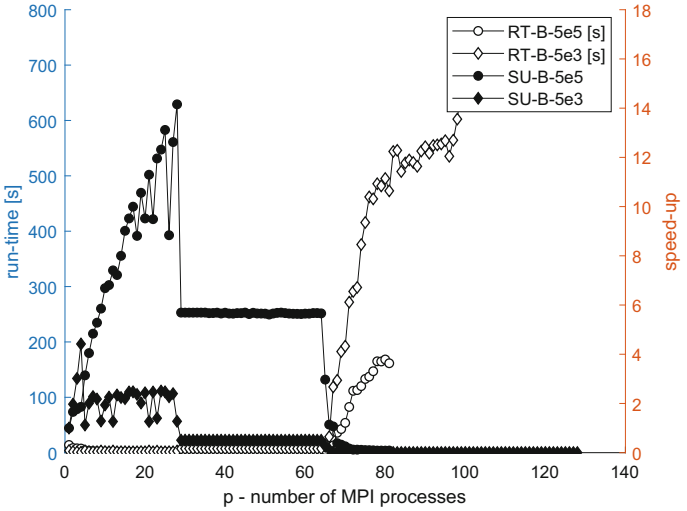

Fig. 7.5 Parallel run-time (RT-B) and speedup (SU-B) of heat equation solution for $N = [5e5, 5e3]$ and 1 to 128 MPI processes bound to cores

mainly because of the communication start-up time. The run-time (RT-B) in seconds and corresponding speedup (SU-B), as a function of the number of MPI processes $p = [1, \ldots, 128]$ and the number of discretization points $N = [5e5, 5e3]$ is shown in Fig. 7.5.

The results are a surprise! The speedup is very unstable and approaches to maximum value 14 with about 30 processes. Then it jumps to 6 and remains stable until 64 processes with another jump to almost 0, with more than 64 processes. We guess that the problem is in communication.

The first step occurs when the number of MPI processes increases from 28 to 29. This step happens because one communication channel gets the additional burden. Such a behavior slows-down the whole program because the remaining processes are waiting.

The results are a surprise! The speedup is very unstable and approaches to maximum value 14 with about 30 processes. Then it jumps to 6 and remains stable until 64 processes with another jump to almost 0, with more than 64 processes. We guess that the problem is in communication. The first step occurs when the number of MPI processes increases from 28 to 29. This step happens because one communication channel gets the additional burden. Such a behavior slows-down the whole program because the remaining processes are waiting. Then processes from number 30 to 64 are only adding to communication burden of other neighboring nodes, which happens in parallel and, therefore, does not additionally degrade the performance. Since the communication overhead at this number of MPI processes easily overwhelms calculation, the speedup seems constant from there on.

Then processes from number 30 to 64 are only adding to communication burden of other neighboring nodes, which happens in parallel and, therefore, does not addi-

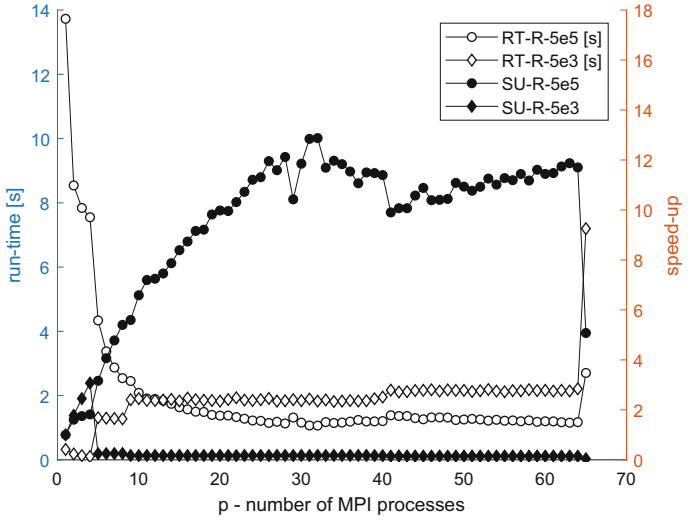

Fig. 7.6 Parallel run-time (RT-R) and speedup (SU-R) of heat equation solution for $N = [5e5, 5e3]$ and 1 to 65 MPI processes on a ring interconnection topology and implemented by `MPI_Sendrecv`

tionally degrade the performance. Since the communication overhead at this number of MPI processes easily overwhelms calculation, the speedup seems constant from there on.

Second step happens, when number of processes passes 64. Process 65 is assigned to node 1. This makes it the ninth MPI process allocated on this node, which only supports 8 threads. Ninth and first process on node 1, therefore, have to share to the same core, which seems to be the recipe for abysmal performance. The speedup drop is so overwhelming at this point because MPI uses busy waiting as a part of its synchronous send and receive operations. Busy waiting means that processes are not immediately switched when waiting for MPI communication, since the operating systems do not see them as idle but rather as busy. Therefore, the waiting times for MPI communication dramatically increase and with them the execution times.

Therefore, we repeat the experiment again. Now the processors are connected in a true physical ring topology with two communication ports per processor. Additionally, we replace the `MPI_Send` and `MPI_Recv` pairs by `MPI_Sendrecv` function. We reduce the number of processes in this experiment to 64, because larger numbers have been proved as useless. The run-time (RT-R) in seconds and corresponding speedup (SU-R), as a function of the number of MPI processes $p = [1, \ldots, 64]$ and the number of discretization points $N = [5e5, 5e3]$ is shown in Fig. 7.6.

We can notice several improvements now. With a larger number of discretization points, the speedup is quite stable but the maximum is not higher than in the previous experiment from Fig. 7.5. With smaller number of discretization point a speedup is detected only in up to four processes, because a local memory communication is used, which confirms that the communication load is prevailing in this case. Further

investigation needs a lot of exciting engineering work, however, it is beyond the scope of this book and is left to enthusiastic readers.