Solving ARGESIM Benchmark CP2 'Parallel and Distributed Simulation' with Open MPI/GSL and Matlab PCT – Monte Carlo and PDE Case Studies

David Jammer[1,2*], Peter Junglas[2], Sven Pawletta[1]

[1]Research Group Computational Engineering and Automation, University of Applied Sciences Wismar, Philipp-Müller-Straße 14, 23966 Wismar, Germany;
[*]*david.jammer@cea-wismar.de*
[2]PHWT-Institut, PHWT Vechta/Diepholz, Am Campus 2, 49356 Diepholz, Germany;

SNE 32(4), 2022, 211-220, DOI: 10.11128/sne.32.bncp2.10625
Received: 2022-10-19; Revised: 2022-11-10
Accepted: 2022-11-15
SNE - Simulation Notes Europe, ARGESIM Publisher Vienna
ISSN Print 2305-9974, Online 2306-0271, www.sne-journal.org

Abstract. The ARGESIM benchmark CP2 provides three different tasks to study current technologies for the parallelization of simulation programs. The first task is the Monte Carlo study. In this study, a spring-mass system is simulated with different damping factors. The second task is a Latice Boltzmann simulation in which the flow of a fluid in a special geomentry is simulated. The third problem is a partial differential equation (PDE) describing a swinging rope, which is solved by the Method of Lines. The Monte Carlo and the PDE study are solved here, each one with two different methods: The first one applies the standard MPI message passing library together with the GNU Scientific Library, the second one uses Matlab from The MathWorks in combination with the Parallel Computing Toolbox. A special focus of this work is on the parallel processing functions provided by Matlab. The solutions are compared with each other in terms of performance and scalability. In most cases, the solutions with OpenMPI and GSL were faster than the solutions with Matlab PCT. The Matlab PCT offers many functionalities and applications to accelerate, but these usually have a poor runtime behavior.

Introduction

In simulation technology, methods to accelerate simulation were investigated in the early phases. The first benchmarks (CP1) of the SNE series dealing with this topic date back to 1994 [1] and were successfully solved and investigated with different technologies and on different platforms. This benchmark got an update (CP2) in 2007 [2] to adapt it to the increasing computing power. Unfortunately, no further solutions were submitted after this change. Since 2007, the computing power and the architecture of the hardware and software have changed a lot, so the parallel benchmarks should be brought back to life.

In this paper, two tasks of CP2 will be investigated. The tasks were implemented with two different technologies.

The first technology is the Message Passing Interface (MPI) [3] in version 4 together with the GNU Scientific Library (GSL) [4]. MPI was developed in the early 1990s and standardized in 1994. Since then MPI has been developed continuously and is still one of the standard technologies in parallel processing. MPI has been implemented by several institutes. In this paper Open MPI 4 [5] was used. GSL was developed in 1996 by M. Galassi and J. Theiler from Los Alamos National Laboratory and is currently updated and further developed. The solutions designed with it were implemented in the C language. Thus, the first solution is based on open source solutions.

The second technology is based on Matlab. In 1995 C. Moler published that The Mathworks would not be active any longer in the field of parallel processing with Matlab because of unsuccessful investigations [6]. But already in the early 1990s several open source projects had started to enable parallel processing with Matlab and similar systems. One of the first projects was developed by our research group and was presented on the Matlab Conference 1995 [7]. This development resulted in the Distributed & Parallel Toolbox [8]. In the following decade a number of similar open source projects appeared [9]. In 2004 The Mathworks adopted these developments eventually and published the first version of the Distributed Computing Toolbox [10], which became later the Parallel Computing Toolbox (PCT) [11]. The investigations in this paper are based on the PCT Version 7.4.

The computations were performed on the Seneca cluster of the PHWT-Institut. It consists of 3 nodes that communicate with each other over the high-speed InfiniBand network. Table 1 lists the data from Seneca.

Nodes	3
Cores	288
Processors	AMD Epyc 7552
Main memory	1536 GiByte
High-speed network	100 GBit/s InfiniBand
Management network	1 GBit/s Ethernet
Operating system	OpenSuse Leap 15.3
Middleware	OpenHPC
Cluster management	Warewulf
Job Scheduler	SLURM
Software	GCC 9.3.0 GSL 2.6 open MPI 4.1.1 ucx 1.13.0 libfabric 1.13.0 hwloc 2.1.0 Matlab R2021a

Table 1: Seneca hardware and software overview.

1 Initial Investigations

One of the most important factors in parallel processing is communication. For an application to be significantly accelerated by parallel processing, high communication performance is required. In the HPC domain, special high-speed networks are used for this purpose, which are very expensive but have great advantages over standard technologies. A common high-speed network is InfiniBand from NVIDIA (former Mellanox), which is also used in Seneca. The big advantage of InfiniBand is its low latency and high data transfer rate. To make the results of this benchmark comparable between machines with different architectures, we will initially provide some results of corresponding measurements.

Figure 1 displays the round trip time and the transmission rate between two cores of different nodes as a function of the packet size.

If the packet size approaches 0, then this corresponds approximately to the latency. This latency is approx. 3 μs for Seneca. The maximum transmission rate is approx. 96 GBit/s.

Another communication medium is the main memory. Since the multicore technology has been strongly enhanced for several years and the performance continues to increase, it is also used massively in the HPC domain. The AMD processors used are based on the NUMA (Non-Uniform Memory Access) architecture.

Two investigations can be carried out here, which measure different aspects of the communication via the memory inside one node: Firstly, the exclusive communication between two cores (cf. Figure 2), and secondly, the simultaneous communication of several cores. Figure 3 shows the corresponding transfer rate as a function of the number of cores.

In Figure 2, it is noticeable that a maximum of approx. 200 GBit/s is at a packet size of 128 KiByte. For larger packets, the transfer rate is 100 GBit/s. The latency (packet size against 0) is approx. 0.4 μs.

The transfer rate shown in Figure 3 reaches a maximum of approx. 150 GiByte/s. In this measurement, 1 GiByte was copied back and forth 500 times per core.

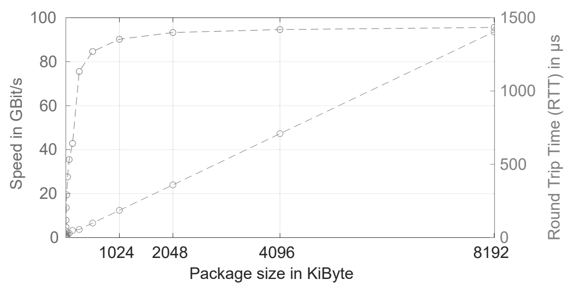

Figure 1: Communication between two nodes via InfiniBand.

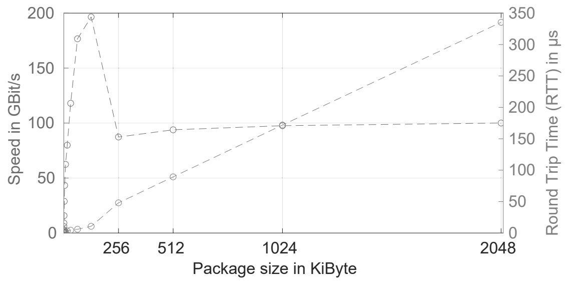

Figure 2: Communication between two cores via main memory.

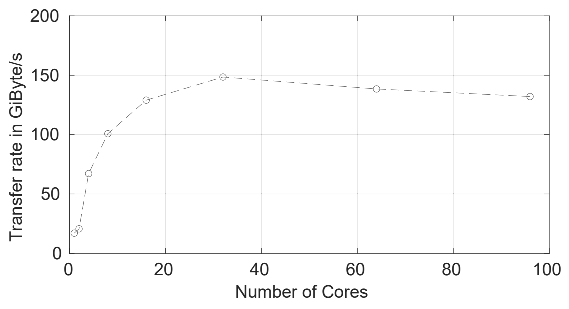

Figure 3: Memory bandwidth depending on the number of cores.

In a classic UMA (Uniform Memory Access) architecture, the transfer rate would remain constant because the instances share the memory channel to the main memory. However, since a NUMA architecture has several memory channels, the transfer rate increases because more memory channels are used.

2 Monte Carlo Study

The first benchmark is a Monte Carlo study. In this benchmark, the behaviour of a spring-mass system with different damping factors has to be calculated. The spring-mass system is described by the usual differential equation:

$$m\ddot{x} + d\dot{x} + kx = 0$$

with the parameters:

$$x(0) = 0, \dot{x}(0) = 0.1, k = 9000, m = 450$$

The damping factor d is randomly selected using a uniform distribution with the range [800,1200].

In [2] $nReps = 1000$ simulations with a step size of $h = 0.01$ in the period from 0 to 2 are required. The mean value of $x(t)$ is then to be calculated from the simulations. Since the computing power of computers has increased significantly since the publication of [2], we have increased the load of the task. For the following solutions, $nReps$ = 10,000,000 simulations were performed with a step size of $h = 0.001$.

2.1 Open MPI with GSL

This section presents the solution using Open Message Passing Interface (MPI) [5] and the GNU Scientific Library (GSL) [4]. The program is written in the C programming language. The differential equation is transformed into the usual first order form:

$$\dot{y}_1 = y_2$$

$$\dot{y}_2 = -\frac{d}{m}y_2 - \frac{k}{m}y_1$$

and then solved with an RK4 solver with fixed step size. GSL provides all necessary functions and data structures for this task. According to [12], the development of a parallel solution, starting from the entire problem, consists of the steps partitioning, communication, agglomeration, and mapping. The partitioning of the Monte Carlo study results in the following tasks: *nReps* simulations, calculation of the mean value and the storage of the results. The communication includes the transfer of the results of the *nReps* simulations to the averaging operation and from there to the storage of the results. The aim of the subsequent agglomeration step is to reduce communication and combine tasks. Here, several simulations per process and the computation of corresponding partial sum vectors are combined. This reduces the communication per process to the transmission of the partial sum. Another part is the addition of all partial sum vectors, the averaging and the storage. The addition of all partial sum vectors is done collectively via a Reduce function and the averaging and storage is combined as one task and assigned to an arbitrary process. The mapping is done automatically by the middleware and the operating system.

The solution was designed as SPMD (Single-Program Multiple-Data) as shown in Figure 4. Since all simulations need the same amount of computation, a load balancing scheme is unnecessary.

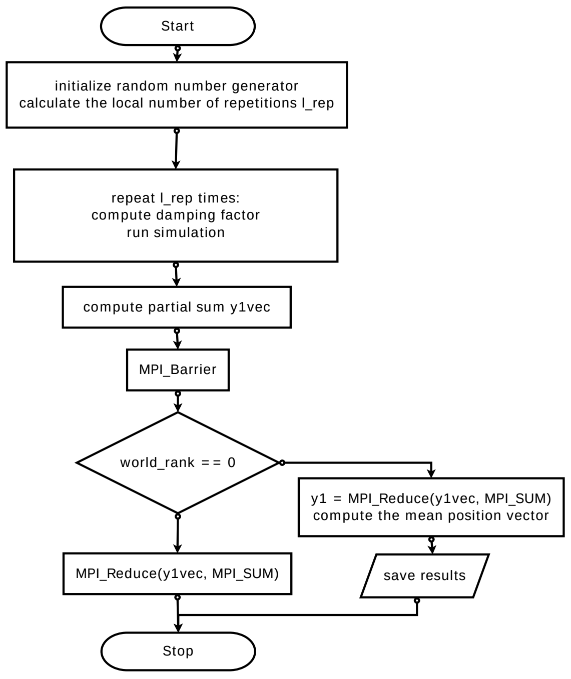
Start
initialize random number generator
calculate the local number of repetitions l_rep
repeat l_rep times:
compute damping factor
run simulation
compute partial sum y1vec
MPI_Barrier
world_rank == 0
y1 = MPI_Reduce(y1vec, MPI_SUM)
compute the mean position vector
MPI_Reduce(y1vec, MPI_SUM)
save results
Stop

Figure 4: Program flowchart of the monte carlo study with OpenMPI.

Therefore, the number l_{rep} of local repetitions can be computed beforehand by distributing the total number n uniformly among the tasks. Then each MPI process calculates the needed random numbers and performs its simulations. The seed of each local random number generator is initialized with the ID of the MPI process to get independent random numbers for each process. The position values y_1 are added locally after each simulation, thus only one vector has to be transferred afterwards. After all processes have finished their computations, the addition of the partial sum vectors is done collectively by `MPI_Reduce`. Now, the MPI process with rank 0 contains the result of the addition and only has to calculate the average and finally store the data.

2.2 Matlab PCT

The PCT can be used for parallel programming on a local multi-core machine or on a cluster like Seneca. To use PCT, one first creates a parallel pool with a given number of processes, called "workers". On a cluster, Matlab always requires an additional worker as the top instance. This is important, because this worker also needs a license. PCT defines several different models for parallel programming.

For the Monte Carlo study three technologies have been used: `parfor`, `spmd` and `parsim`.

The solution with parfor is very easy to implement, because the decomposition is performed implicitely. Parfor works like a normal for-loop with the only difference that the iterations of the for-loop are distributed to the workers. Unfortunately, the PCT documentation doesn't describe the scheduling strategy of the parfor loop, it simply states that the iterations are done in non-deterministic order [11]. However, it is not important for this Monte Carlo study, since the computational effort of the simulations is always the same. A parfor loop is only possible if the iterations are completely independent from each other. In our case, this condition is obviously fulfilled. A vector *D* with *nReps* random numbers is calculated in the sequential section before the parfor loop. In the parfor section, only the `ode45` function is called and the sum is formed as in Listing 1.

Listing 1: Matlab parfor-loop

```
parfor i=1:nReps
  [tout, yout] = ode45(@(t,y)
                      ode(t,y,K, D(i),M),
                      tvec,y0);
  ysum = ysum + yout(:,1);
end
```

The communication happens here implicitly by the calculation of the sum using the variable *ysum*, which was defined in the sequential section and can be used by all workers. After the parfor loop only the mean value remains to be computed.

Another solution was implemented and investigated with `spmd`. The `spmd` function creates a parallel section that is executed by each worker. The number of workers is given by the size of the parallel pool and can be determined in the `spmd` section via the variable `numlabs`. The index of a worker is stored in the variable `labindex`. Data can be sent explicitely between workers using the functions `labSend` and `labReceive`. For synchronization the function `labBarrier` is available. The solution with `spmd` is similar to the solution with Open MPI from section 2.1. In the `spmd` section, the local number of repetitions l_{rep} is calculated and then the simulations are performed. A special challenge is the handling of anonymous functions, because they cannot be defined in an `spmd` section. But they are needed to modify the parameters of the differential equation.

The easiest way to realize this, is to create a function outside the spmd section, which contains the ode45 call and the definition of the anonymous function. This is shown in Listing 2.

Listing 2: Matlab ode45 call with anonymous for the spmd section

```
function [t,y] = odecall(tvec,y0,K,D,M)
  [t, y] = ode45(@(t,y) ode(t,y,K,D,M),
                 tvec, y0);
end
```

Similar to section 2.1 the position values are added and the sum is calculated by the general reduction function `B = gop(fcn,A,destination)`, where `@plus` is used for `fcn` to achieve a collaborative summation.

The third solution was realized with Matlab/Simulink and parsim. For this purpose, the spring-mass system must be modeled by a signal flowchart in Simulink (cf. Figure 5). Afterwards, only

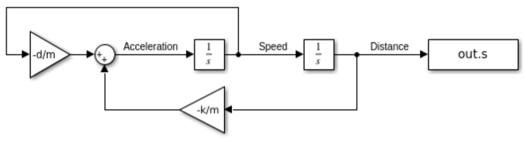

Figure 5: Spring-mass system as signal flowchart.

the input data must be defined as Simulink simulation input objects and then the simulations can be started with parsim. After the simulations are finished, the mean value can be calculated from the results. This solution is the simplest of all, but it must be noted that all results are available before averaging, which can lead to a high memory overhead.

2.3 Comparison of solutions

All approaches resulted in the same solution as shown in Figure 6.

Figure 7 shows the runtime and the speedup results of the Monte Carlo study. Apart from the parsim method, which has no speedup at all, the other implementations show a significant speedup. The runtimes of the two Matlab solutions are almost identical and have a significantly higher runtime than the C implementation. The transition from one node to two nodes is interesting: The Matlab implementations show a jump

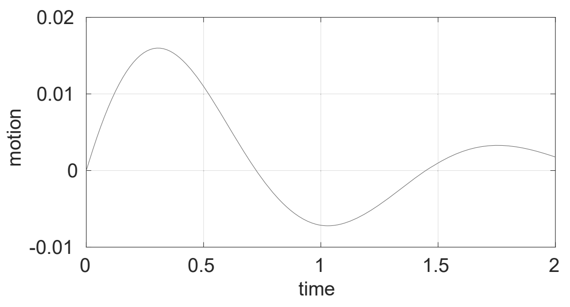

Figure 6: Plot of the mean motion.

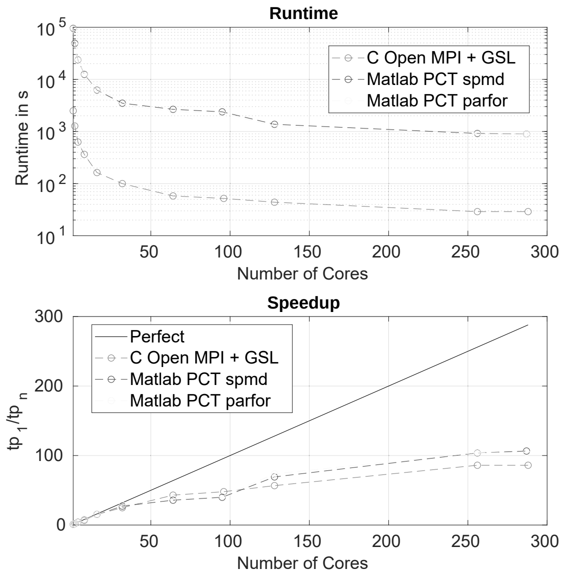

Figure 7: Runtime and speedup history of the monte carlo implementations.

in speedup in that area, whereas the C implementation does not have this jump.

The solution with Matlab and parsim did not result in any significant acceleration (cf. Figure 8). Also, the number of simulations had to be reduced significantly to cope with the high memory requirements of this solution.

Another interesting investigation in a Monte Carlo study is the scaleup. The execution time is constant in a scaleup approach, so only the spmd method was investigated in Matlab. In an spmd section, the maximum execution time can be implemented directly. Here, a time of 60 seconds was specified in which the simulations are carried out. Figure 9 then shows how many simulations are calculated depending on the number of cores used.

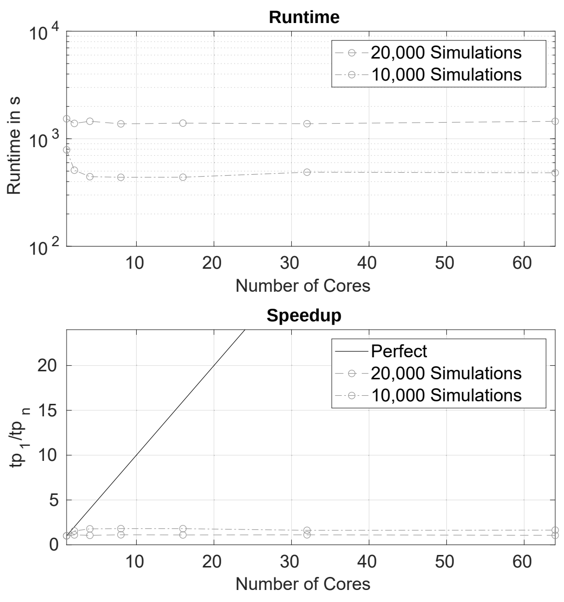

Figure 8: Runtime and speedup history of the monte carlo parsim implementation.

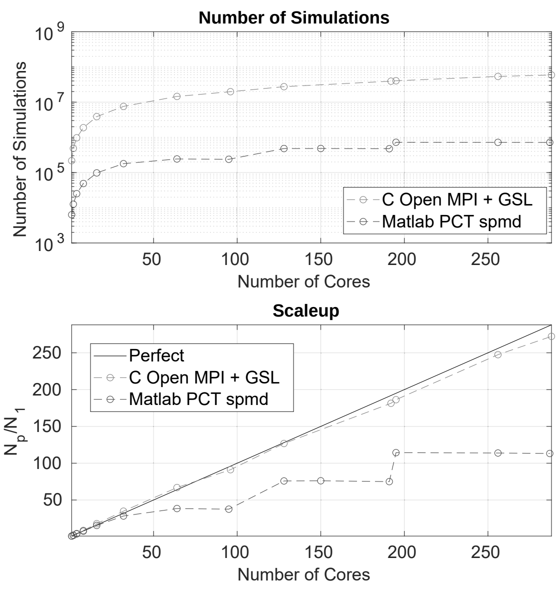

Figure 9: Scaleup history of the monte carlo implementations.

The C implementation shows an almost linear behavior, whereas the Matlab/spmd implementation behaves strangely: The transitions from one (96 cores) to two (192 cores) nodes and from two (192 cores) to three (288 cores) nodes are striking. In these transitions the number of simulations increases abruptly.

In comparison, the C implementation clearly shows the best results, as was expected. Nevertheless, the Matlab implementations can provide reasonable speedup with little programming effort.

3 Partial Differential Equation Case Study

The second benchmark investigated in this article is the solution of a partial differential equation describing a swinging rope:

$$\frac{\partial^2 u}{\partial x^2} = \frac{1}{v^2}\frac{\partial u}{\partial t^2}$$

As suggested in the benchmark, the equation will be solved with the method of lines. For this purpose, the left side of the differential equation is replaced by a central difference quotient of 2nd order:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}(t) - 2u_i(t) + u_{i-1}(t)}{(\frac{L}{N})^2}$$

Here L corresponds to the length of the rope, N to the number of equidistant intervals used for space discretization and i to the location. Applying this method, a 2nd order differential equation is obtained for each location point:

$$\ddot{u}_i = \frac{v^2}{k^2}(u_{i+1} - 2u_i + u_{i-1}), \qquad i = 1,...,N-1$$

with

$$k := L/N$$

The initial values are:

$$u_i(0) = 2\frac{h}{N}i, \qquad i = 1,...,\frac{N}{2}$$

$$u_i(0) = 2h(1 - \frac{i}{N}), \qquad i = \frac{N}{2},...,N-1$$

and

$$\dot{u}_i(0) = 0, \qquad i = 1,...,N-1$$

The boundary values are given by

$$u_0(t) = u_N(t) = 0, \qquad t \in [0, t_{end}]$$

The parameter values are given as $\nu = 0.06$, $L = 0.5$, $h = 0.05$ and $t_{end} = 10$. The benchmark specifies the space discretization as $N = 500$ and the size of time steps as $dt = 0.01$. In the following, other values will be used to adapt to the increase in computing power. As a solution, the time evolution of the amplitude at the points $x = \frac{3}{4}L$ and $x = \frac{1}{2}L$ is to be presented, as well as the space evolution at the points in time $t = 5$ and $t = 8$ and a surface plot, showing the complete solution $u(x,t)$.

3.1 Open MPI with GSL

Using the design method for parallel programs described in section 2.1 results in a very fine grained description: The main tasks are the computations of one time step at one position. This leads to a huge amount of communication, which will be reduced drastically after the proper agglomeration step. Since the resulting strategy is fairly standard, we will skip these details and immediately describe the overall parallelization strategy.

The basic idea is to distribute contiguous uniformly sized sections of the rope to the tasks, in the order of the taskIds (cf. Figure 10). Note that the local sections of the first and last taskIds include the endpoints u_0 and u_N, which contain the fixed boundary values. The algorithm consists of a time loop, where each iteration starts with an exchange of the necessary boundary values, followed by one step of an RK4 ODE solver. A barrier between steps synchronizes the tasks to guarantee the correct internal boundary values. As part of the steps all necessary results are collected by the task with taskId 0 and stored.

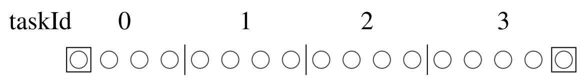

Figure 10: Decomposition of space points.

To organize the communication and to simplify the local computations so called "ghost points" ([13]) are used (cf. Figure 11): The local u arrays are extended by one additional point at each end (or only one end for the first and last task). In the communication phase the boundary values are stored here, so that all necessary values are readily available at the computation phase.

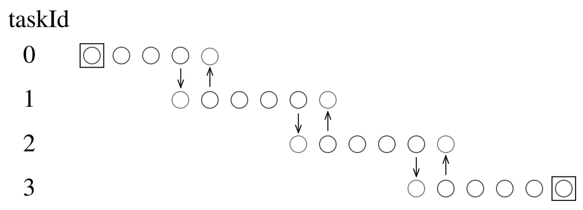

Figure 11: Decomposition including ghost points.

3.2 Matlab PCT

Three solution approaches were investigated using Matlab PCT. The first one ("loop-based") closely mimics the MPI version adopting the `spmd` environment and the `labSend`, `labReceive`, `labindex` and `labBarrier` functions. As in MPI the definition of the ODE is done in a loop over the space points. For the tedious task of mapping between the global and local indices – prone to typical one-off errors – Matlab PCT supplies the function `codistributor1d`, which provides all necessary details.

For the second approach ("matrix-based") the variables u_i are combined in a vector u and the ODE is rewritten in matrix-vector form as

$$\ddot{u} = Au$$

with

$$A = \frac{\nu^2}{k^2} \cdot \begin{pmatrix} -2 & 1 & 0 & 0 & \cdots \\ 1 & -2 & 1 & 0 & \cdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \cdots & 0 & 1 & -2 & 1 \\ \cdots & 0 & 0 & 1 & -2 \end{pmatrix}$$

The local parts of A are computed in an initial phase. The parallelization follows the pattern of the loop-based version, only the internal loop over points is replaced by a matrix-vector multiplication.

The third approach ("codistributed") applies codistributed arrays, a very convenient tool defined in Matlab PCT. A codistributed array basically is an array that is distributed to the workers, but every task can still access each element with the usual (global) index. If the corresponding element is not locally available, it is sent automatically from the hosting worker to the accessing one. Auxiliary functions are supplied to find the local indices, but they are not needed here: Several Matlab functions, such as the matrix-multiplication, automatically cope with codistributed arrays, so that a simple call by all workers is sufficient.

Of course, the crucial point here is the distribution scheme. Matlab PCT supplies the usual equal-partitioning along rows or columns and a block-cyclic twodimensional partitioning.

Surprisingly, the loop-based approach reached the shortest runtime, while the matrix-based approach was slower. The codistributed approach is very elegant from a programming point of view, since the distribution and communication parts are done implicitly, but it is extremely slow compared to the other solutions. Table 2 shows the runtimes of the implementations in comparison. For the further investigations only the loop-based approach has been used.

solution approach	runtime [s]
loop-based	4.23
matrix-based	5.77
codistributed	359.85

Table 2: Running time of the three solutions (nWorker=8, dt=0.001, N=1000).

3.3 Comparison of solutions

All implementations returned the same result. Figure 12 shows the excitation over time and space.

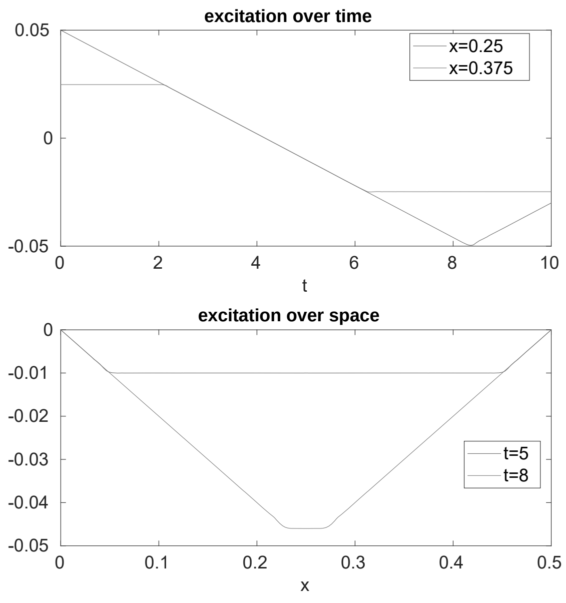

Figure 12: Solution of the PDE, excitation over time and space.

Figure 13 shows the required surface plot.

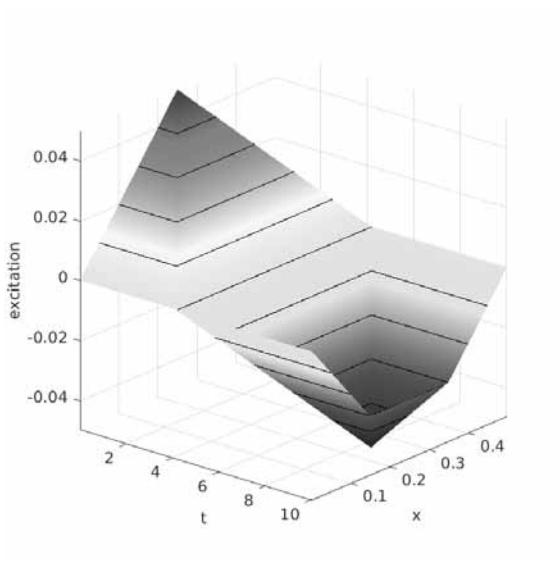

Figure 13: Surface plot for the swinging rope.

Since for the surfaceplot all data over time and space must be stored, this task does not scale well to large values of N. The result files become too large at higher space and time resolutions. In order to increase the computational effort nevertheless, the collection of all data was omitted for the following measurements. Only the excitation over time at the locations $x = \frac{3}{4}L$ and $x = \frac{1}{2}L$ and the excitation over space at the times $t = 5$ and $t = 8$ were collected and stored.

The runtime of the two implementations Open-MPI/GSL and Matlab PCT/loop-based was investigated depending on the number of processes. For comparison, runs with $N = 200,000$ and $N = 1,000,000$ have been studied. The results of the speedup analysis are shown in Figure 14 and 15. A significant speedup was reached in all scenarios. The C implementation with Open MPI/GSL reached the lowest runtimes and also the highest speedups. The runtime was reduced from about 20 minutes (N=200,000) to about 1 minute and from about 14 hours (N=1,000,000) to about 6 minutes.

This corresponds to a maximum speedup of about 20 for N=200,000 and about 140 for N=1,000,000. The Matlab PCT/loop-based solution reduced the runtime from 30 minutes to 2 minutes (N=200,000) and from about 10 hours (N=1,000,000) to 25 minutes, respectively. This is equivalent to a speedup of 15 for N=200,000 and 23 for N=1,000,000.

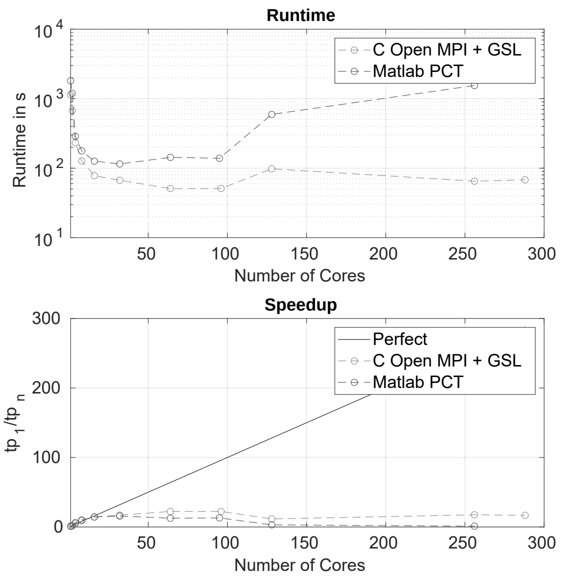

Figure 14: Runtime and speedup history of the PDE implementations with N=200,000, dt=0.00005 and t_{end}=10.

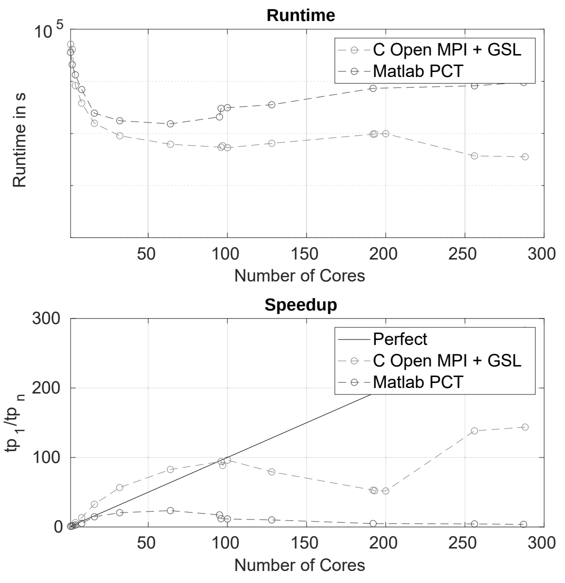

Figure 15: Runtime and speedup history of the PDE implementations with N=1,000,000, dt=0.00001 and t_{end}=10.

The bad speedup behaviour for N=200,000 indicates that this problem is still too small for the hardware, whereas for N=1,000,000 the MPI/GSL version shows reasonable speedups even for large core numbers. The superlinear speedup for one cluster probably is due to a better use of the memory lines and caches. Interestingly, Matlab PCT is faster than MPI/GSL for one core, but generally has much less speedup and can't put more than one cluster node with 96 cores to good use.

4 Conclusion

In this article, two solutions for each of two tasks of the ARGESIM CP2 benchmark have been presented and compared with each other. Both tasks are very lightweight for today's computer systems, the sequential Monte Carlo study takes less than 1 second with the original parameters. Therefore, the tasks have been scaled up considerably. A revision of the CP2 benchmark should be scalable and allow for parameters that lead to runtimes of 30 minutes or more.

As expected, the implementations in C with Open MPI/GSL generally reached shorter runtimes and reasonable speedups for up to more than 100 cores, especially for large problem sizes. The basic Matlab PCT solutions using `parfor` or `spmd` showed good speedups for up to over 100 cores in the Monte Carlo study, while in the PDE example good speedups could only be reached for up to 30 cores. Using high-level methods like `parsim` or codistributed arrays produced elegant looking programs, but they could not compete at all due to their huge execution times.

Another problem is the insufficient documentation of Matlab PCT: Its authors try to spare the reader most internal details, such as the load-balancing scheme of `parfor` or the exact behaviour of codistribution-aware functions, which are important for good parallelization strategies.

In a subsequent work the third task of the CP2 benchmark, the Lattice Boltzmann simulation, will be solved, again using implementations in OpenMPI and Matlab PCT.

References

[1] Breitenecker F, Husinsky I, Schuster G. Comparison of Parallel Simulation Techniques. *SNE Simulation News Europe*. 1994;4(10):21–22. ISSN 0929-2268.

[2] Breitenecker F, Höfinger G, Pawletta T, Pawletta S, Fink R. ARGESIM Benchmark on Parallel and Distributed Simulation. *SNE Simulation News Europe*. 2007; 17(1):53–56. ISSN 0929-2268.

[3] MPI: A Message-Passing Interface Standard. URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[4] Galassi M, et al. *GNU Scientific Library Reference Manual*. 3rd ed. URL https://www.gnu.org/software/gsl/

[5] The Open MPI Project. URL https://www.open-mpi.org/

[6] Moler C. Why there isn't a parallel MATLAB. In: *Matlab News and Notes*. Spring. 1995; p. 12.

[7] Pawletta S, Pawletta T, Drewelow W. A MATLAB toolbox for distributed and parallel processing. In: *2nd International MATLAB Conference*. Cambridge. 1995; .

[8] Pawletta S, Pawletta T, Drewelow W, Dünow P. *Distributed and Parallel Application Toolbox for Use with MATLAB: User's Guide and Reference Manual*, version 1.3 ed. 1996.

[9] Fink R, Pawletta S, Pawletta T, Lampe B. SCE based Parallel Processing and Applications in Simulation. *SNE Simulation News Europe - Special Issue on Parallel and Distributed Simulation Methods and Environments*. 2006;16(2):37–50. ISSN 0929-2268.

[10] The Mathworks Inc. *Distributed Computing Toolbox for Use with MATLAB: User's Guide*, version 1 ed. 2004.

[11] The Mathworks Inc. *MATLAB Parallel Computing Toolbox: User's Guide*, version 7.4 ed. 2021.

[12] Foster I. *Designing and Building Parallel Programs - Concepts and Tools for Parallel Software Engineering*. Amsterdam: Addison-Wesley. 1995.

[13] Gropp W, Lusk E, Skjellum A. *Using MPI, third edition - Portable Parallel Programming with the Message-Passing Interface*. Cambridge: MIT Press. 2014.