Chapter 1

Modeling Parallel Applications in the MERPSYS Environment

Paweł Czarnul

Abstract

The chapter presents how to model parallel computational applications for which simulation of execution in a large-scale parallel or distributed environment is performed within the MERPSYS environment. Specifically, it is shown what approaches can be adopted to model key paradigms often used for parallel applications: master-slave, geometric parallelism (single program multiple data), pipelined and divide-and-conquer applications. A detailed approach is proposed for geometric parallel applications along with steps that need to be taken in the MERPSYS environment: from modeling to simulation of execution.

1.1 Modeling and Running Simulations in MERPSYS – Key Concepts

The MERPSYS environment allows to model parallel applications implementing in particular one of the following paradigms:

1. master-slave,
2. SPMD,
3. pipeline,
4. divide-and-conquer

as well as corresponding systems on which simulations of the former are performed.

The advantage of the approach is the possibility to model large scale applications and systems by provision of solution to automatically scale a simulation in two dimensions:

1. Data sizes – increasing the data size processed by processes or threads does not increase the running time of a simulation as opposed to the running time of a real application. This is because simulation of computations is performed by computing the running time of a computational block using

a formula and the input data size and CPU/GPU performance. Then the simulator skips until the next event.

2. The number of processes/threads and system nodes on which the former run. Specifically, it is possible to set the numbers which could not be run on available systems because the available resources would not allow to do so. In this respect, it allows both:

   a) prediction of application behavior especially speed-ups taking into account limitations such as the communication/synchronization overhead

   b) simulation of various hardware configurations and selection which one would be the best for the simulated application.

MERPSYS was designed in order to be able to run simulations of parallel applications on large scale systems composed of high performance computing clusters and volunteer based subsystems. The goal of a simulation was to compute:

1. execution time of an application,
2. energy consumed by the system during the application run,
3. probability of successful execution of an application.

These are especially important for running applications on large scale systems composed of a very large number of nodes and cores. MERPSYS (following [1]) offers, in particular, the following benefits:

1. running a simulation faster than real runs, especially for applications with large data sizes – this allows testing various parallelization concepts within code,
2. possibility to configure an environment that is not available on site e.g. much larger, with any compute devices and interconnects – this allows testing e.g. potential hardware upgrades that would maximize gains considering applications typically run in the system,
3. optimization of running sets of applications i.e. suggesting how many nodes/-cores each of the applications should be run on, which resources each of the application should be run on.

In order to make use of these benefits, a typical workflow needs to be executed in the system. A basic workflow is composed of the following steps:

1. Definition of a system model. A multi-layered model can be defined at the following levels:

   WAN level where clusters are interconnected with a WAN,

   cluster level where nodes are interconnected with a LAN,

node level where components such as CPUs and GPUs are connected with PCIe.

It is important to note that apart from the system structure, a user needs to define the following:

a) labels that will be used to mark distinct codes of processes or threads in the application model and provision of logical containers for these processes/threads at the system level. In other words, it is necessary to specify where a particular number (count) of processes/threads with a given label could be launched. Later, it will be up to a scheduler to decide exactly about mapping.

b) a user needs to select a model with formulas that determine particular execution time of communication operations (this can depend on underlying hardware environment).

2. Definition of an application model.

Basic functions that can be used within a program that accepts all Java constructs are as follows (see the system manual for more functions, in particular collective communication calls):

function `sim.computation(...` for modeling execution time of computations. An examplary call can be as follows:

```
sim.computation(data_size,
       ComputationType.CPU,
       SoftwareStack.Undefined,
       1,
       complexityFunction,
       OperationType.Calculations,
       OptimizationType.None);
```

that models execution of computations on data of size `data_size` with the number of reference (within a program) instructions returned by function `complexityFunction`. Function `complexityFunction` would take into account data size and can use some coefficients. An exemplary function could be:

```
String squareComplexity = "function "
+ ConstVar.complexityFunctionName + "("
+ ConstVar.parameters + "){" + "return "
+ ConstVar.getDataSize + "*"
+ ConstVar.getDataSize + ";" + "}";
```

for simple square complexity vs data size. The other arguments can be used to define details that can be taken into account by a simulator for potentially more precise results or target compute devices. These are: type of computations such as (`ComputationType.CPU` or `ComputationType.GPU`), specification of type of operations (currently

supported mode is `OperationType.Calculations`), optimization options (reserved for future use, currently `OptimizationType.None` should be specified).

functions `sim.p2pCommunicationSend(data_size, <dest_label>)` and `sim.p2pCommunicationReceive(<source_label>)` that model point-to-point communication in which `data_size` is the size of data (in bytes) to be sent from a sender to a thread with `<destination_label>`. On the receiving side, a message is received from a sender identified with label `<source_label>`.

functions for collective communication and synchronization such as scatter, gather, broadcast, barrier.

3. Setting up simulation parameters. This includes at least definition of the number of processes/threads with code identified by a specific label that should be launched within the application. Additionally, values of some external variables can be defined that will be passed to the application.

4. Running a simulation.

5. Calibration of the models against real runs for selected configurations. In many cases, it is difficult to assess in advance coefficients that will be used in functions passed to computational functions because it might require detailed analysis of the algorithm. Furthermore, some network parameters might be different under load etc. Consequently, for very precise results, the model might require finding coefficients based on results (execution times) of real runs for particular configurations (data sizes, numbers of processes/threads).

6. Running more simulations that will allow prediction of execution time, energy consumption and reliability for other configurations.

In this chapter we focus on step 2, 3 and 4, especially for geometric SPMD type of applications. The whole workflow was discussed for a master-slave type application computing similarity measures between multidimensional vectors in [2].

For modeling it is important to know the basics of how a simulator works with the application model. As outlined in [3], there are several other simulation environments targeted especially for grid or cloud-based systems. These include a discrete-event based Java-based SimJava [4], a discrete-event, C++ framework OMNeT++ [5], GridSim [6] is a toolkit for modelling and simulation of grid systems, Grid Scheduling simulator (GSSIM) [7] also for grid systems, Mars framework [8] or SST/macro [9] for HPC systems that can use logs from MPI applications, CloudSim [10] for simulation of cloud environments, SimGrid [11] for grid, cloud and HPC systems,

In the MERPSYS system, in order to be able to simulate running of a very large number of processes, the simulator starts a multithreaded Java applications such that for each application process/thread marked with a distinct label one

Java thread is run. Then proper scaling is used in order to simulate execution of a given number of processes or threads with this label.

Consequently, some recommendations can be proposed regarding what labels and counts of processes/threads with particular labels can be proposed for the aforementioned application paradigms. These are summarized in Table 1.1.

Table 1.1: Recommendations on how to use labels in the MERPSYS environment to model particular parallel application paradigms

Application paradigm	Labels in a MERPSYS application model the following
Master-slave	label `master` models a master with count 1 and label `slave` models a group of slave processes with desired count
Geometric parallel SPMD	in 2 dimensional space 5 distinct labels can be used to model interprocess dependencies and communication: `center`, `top`, `bottom`, `left` and `right` as suggested in Section 1.2.2, in 3 dimensions two additional labels can be added `front` and `back` for analogous communication; counts of particular labels should in total amount to a desired number of processes
Pipelined	in this case a distinct label should be used for every distinct stage of a pipeline, various numbers of processes/threads with a given label will model further parallelization of the given stage
Divide-and-conquer	each level of the divide-and-conquer tree can be modeled with a distinct label. This will allow to model communication between particular levels of the tree. For instance a process with label `levelX` will first receive data from one with label `level{X-1}`, then send data to label `level{X+1}`, receive result from label `level{X+1}` and send it to label `level{X-1}`. Sizes of data might vary depending on the level as might numbers of processes/threads at particular levels.

1.2 Case Study of a Geometric Single Program Multiple Data Application

1.2.1 Typical Geometric SPMD Model

Typical SPMD applications model, simulate and find solutions to real life problems in which a 1D, 2D or 3D domain is partitioned into a large number of cells and updates of such cells are performed in successive time steps. Specifically, in every time step of a simulation all of cells need to have values updated. Such updates are very specific for the particular problem being solved in the application such as weather modeling, medical simulations, electromagnetic analysis etc. In the parallel application the typical code in terms of logical steps are as follows:

The code is viewed from the point of view of an individual process or a thread as shown in Listing 1.1.

Listing 1.1: Typical geometric SPMD main loop

```
for(t=0;t<TMAX;t++) {

// updates of particular cells assigned to the process/thread
  computations();

// exchange of boundary cells which are needed as so-called
// ghost cells in the processes/threads handling
// neighboring domains
  communication();
}
```

Such typical code can usually be improved in terms of performance. Specifically, the usual technique of overlapping communication and computations can be used. This results in updates of boundary cells first in each process/thread, then sending the values of the boundary cells to processes/threads handling the neighboring subdomains, subsequent updates of the interior cells and completing the aforementioned sends. The same strategy needs to be applied to the so-called ghost cells that are to be sent by processes/threads handling the neighboring subdomains. Consequently, the final pseudocode of such optimized solution will be as shown in Figure 1.2.

Listing 1.2: Typical geometric SPMD main loop with overlapping communication and computations

```
for(t=0;t<TMAX;t++) {

// compute the new values of boundary cells
// so that these can be sent to processes/threads
// handling neighboring domains
computations(boundary);

// now send boundary cells to processes/threads
// handling neighboring domains
```

```
send_start(boundary);

// and start receiving boundaries from processes/threads
// handling neighboring domains
recv_start(ghost_cells);

// updates of interior cells assigned to the process/thread
  computations(interior);

// complete the started send and recv operations

wait(sends);
wait(recvs);
}
```

This strategy allows to overlap computations for the interior cells as well as communication with processes/threads handling neighboring subdomains.

1.2.2 A Possible Approach to Modeling in MERPSYS

It should be noted that the simulator creates as many distinct simulator threads as the number of labels specified to run. In a real environment, each process in 2D communicates with 4 neighbors. Because of this, an approach shown in Figure 1.1 is suggested by the author for modeling the problem in MERPSYS. In this case, the following can be noted:

1. communication with 4 direct neighbors is naturally modeled,
2. the whole 2D space can be covered with the cross element as shown in Figure 1.1 that also indicates which simulation thread sends/receives data to each other simulation thread.

1.2.3 Modeling in the MERPSYS Editor

The proposed application can be modeled in the MERPSYS environment [1] as shown in the following figures. The MERPSYS environment features three panels that allow to model and manage the following:

1. system model with:
   a) computational components with multiplicity,
   b) network components,
   c) hardware model used for particular components – this model can be substituted even for the already available components which allows changing definition of performance parameters for CPUs, network components etc. and effectively easily changing definitions of several components in a complex model,

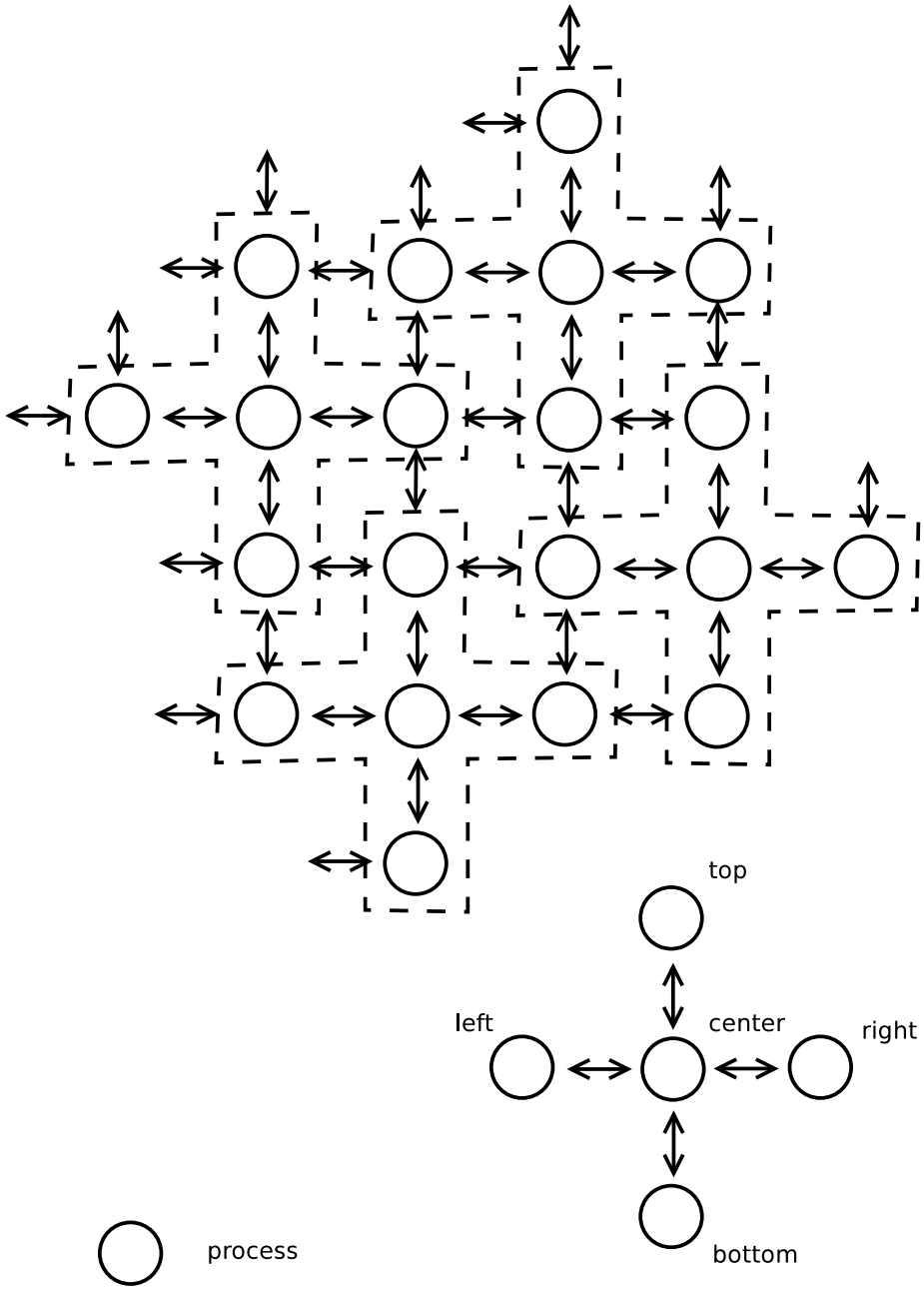

Figure 1.1: An approach to modeling SPMD applications in the MERPSYS environment (2D)

d) definition of so-called labels with multiplicity. The multiplicity denotes how many processes/threads marked with the given label can be launched on the given resource

2. application model

3. simulation management panel.

1.2.4 Proposed Models

Figure 1.2 presents an exemplary model of a cluster that is composed of a number of nodes of a given type with a particular CPU. The nodes are connected with a marked interconnect.

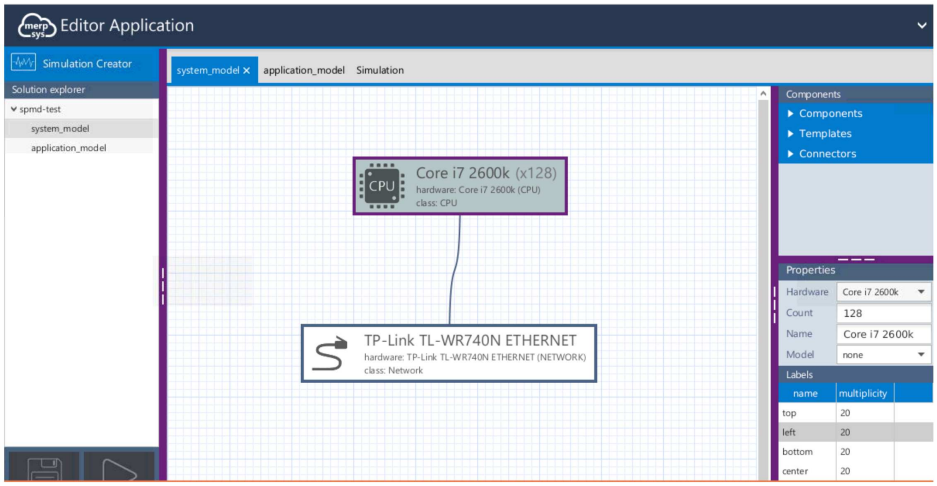

Figure 1.2: System editor

Figure 1.3 presents the view of a parallel SPMD application that was used for subsequent tests.

Figure 1.4 presents the view of the management panel used for subsequent tests.

Figure 1.3 presents the code of the testbed SPMD application that corresponds to the model presented in Figure 1.3. All respective communication actions have been shown.

Listing 1.3: Code of the testbed SPMD application in the MERPSYS environment

```
int domain_size_x=2048*32; // in X dimension
int domain_size_y=2048/32; // in Y dimension
int proccount=sim.getNumberOfProcessesForLabel("top")
+sim.getNumberOfProcessesForLabel("right")
+sim.getNumberOfProcessesForLabel("bottom")
```

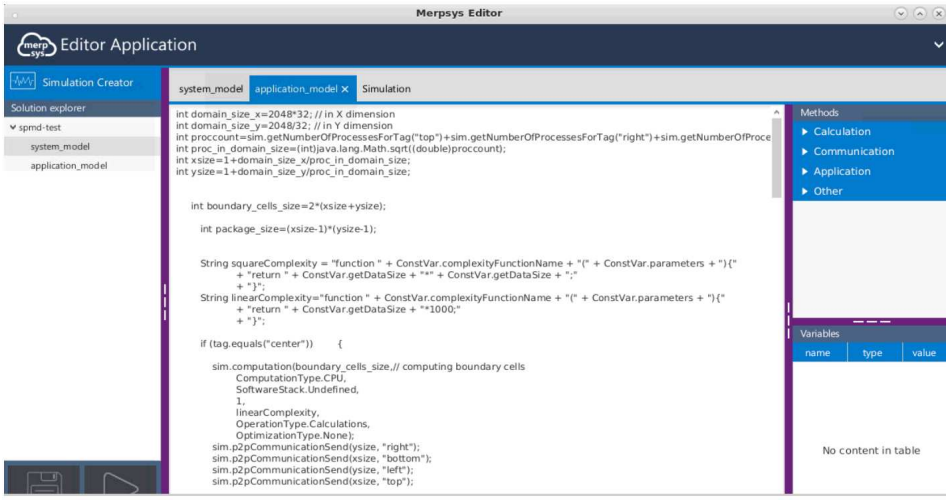

Figure 1.3: Application editor

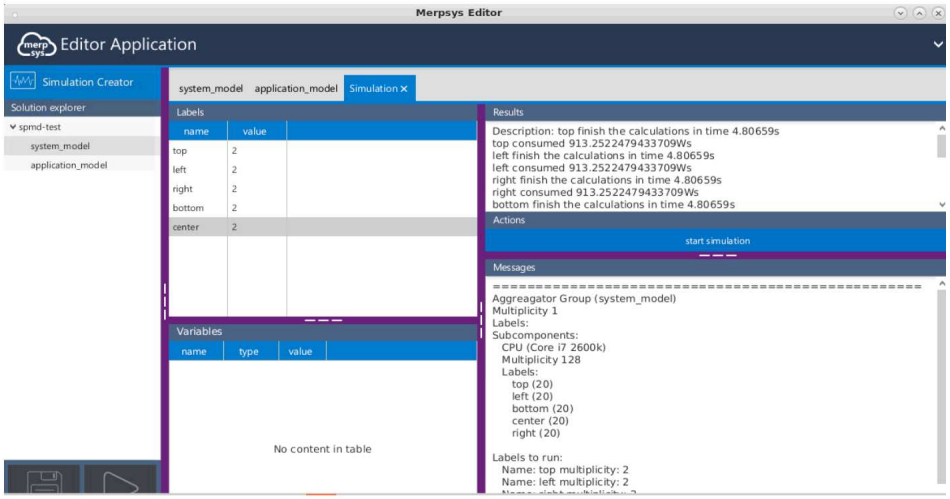

Figure 1.4: Simulation panel

```
+sim.getNumberOfProcessesForLabel("left")
+sim.getNumberOfProcessesForLabel("center");
int proc_in_domain_size=(int)java.lang.Math.sqrt((double)
    proccount);
int xsize=1+domain_size_x/proc_in_domain_size;
int ysize=1+domain_size_y/proc_in_domain_size;

int boundary_cells_size=2*(xsize+ysize);
```

```
int package_size=(xsize-1)*(ysize-1);

String squareComplexity = "function "
            + ConstVar.complexityFunctionName
            + "(" + ConstVar.parameters + "){"
            + "return " + ConstVar.getDataSize
            + "*" + ConstVar.getDataSize + ";"
            + "}";
String linearComplexity="function "
            + ConstVar.complexityFunctionName
            + "(" + ConstVar.parameters + "){"
            + "return " + ConstVar.getDataSize + "*1000;"
            + "}";

if (tag.equals("center"))        {

    sim.computation(boundary_cells_size,// computing boundary
        cells
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
            linearComplexity,
            OperationType.Calculations,
            OptimizationType.None);
    sim.p2pCommunicationSend(ysize, "right");
    sim.p2pCommunicationSend(xsize, "bottom");
    sim.p2pCommunicationSend(ysize, "left");
    sim.p2pCommunicationSend(xsize, "top");

     sim.computation(package_size,
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
            linearComplexity,
            OperationType.Calculations,
            OptimizationType.None);

    sim.p2pCommunicationReceive("top");
    sim.p2pCommunicationReceive("right");
    sim.p2pCommunicationReceive("left");
    sim.p2pCommunicationReceive("bottom");

}

if (tag.equals("top"))        {
    // computing boundary cells
    sim.computation(boundary_cells_size,
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
```

```
                linearComplexity,
                OperationType.Calculations,
                OptimizationType.None);
        sim.p2pCommunicationSend(ysize, "right");
        sim.p2pCommunicationSend(ysize, "bottom");
        sim.p2pCommunicationSend(xsize, "left");
        sim.p2pCommunicationSend(xsize, "center");

         sim.computation(package_size,
                ComputationType.CPU,
                SoftwareStack.Undefined,
                1,
                linearComplexity,
                OperationType.Calculations,
                OptimizationType.None);

        sim.p2pCommunicationReceive("right");
        sim.p2pCommunicationReceive("bottom");
        sim.p2pCommunicationReceive("left");
        sim.p2pCommunicationReceive("center");

}

if (tag.equals("bottom"))          {
        // computing boundary cells
        sim.computation(boundary_cells_size,
                ComputationType.CPU,
                SoftwareStack.Undefined,
                1,
                linearComplexity,
                OperationType.Calculations,
                OptimizationType.None);
        sim.p2pCommunicationSend(xsize, "right");
        sim.p2pCommunicationSend(xsize, "center");
        sim.p2pCommunicationSend(ysize, "left");
        sim.p2pCommunicationSend(ysize, "top");

         sim.computation(package_size,
                ComputationType.CPU,
                SoftwareStack.Undefined,
                1,
                linearComplexity,
                OperationType.Calculations,
                OptimizationType.None);

        sim.p2pCommunicationReceive("right");
        sim.p2pCommunicationReceive("center");
        sim.p2pCommunicationReceive("left");
        sim.p2pCommunicationReceive("top");
```

```
}

if (tag.equals("left"))          {
    // computing boundary cells
    sim.computation(boundary_cells_size,
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
            linearComplexity,
            OperationType.Calculations,
            OptimizationType.None);
    sim.p2pCommunicationSend(xsize, "right");
    sim.p2pCommunicationSend(ysize, "bottom");
    sim.p2pCommunicationSend(ysize, "center");
    sim.p2pCommunicationSend(xsize, "top");

     sim.computation(package_size,
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
            linearComplexity,
            OperationType.Calculations,
            OptimizationType.None);

    sim.p2pCommunicationReceive("right");
    sim.p2pCommunicationReceive("bottom");
    sim.p2pCommunicationReceive("center");
    sim.p2pCommunicationReceive("top");

}

if (tag.equals("right"))          {
    // computing boundary cells
    sim.computation(boundary_cells_size,
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
            linearComplexity,
            OperationType.Calculations,
            OptimizationType.None);
    sim.p2pCommunicationSend(ysize, "center");
    sim.p2pCommunicationSend(xsize, "bottom");
    sim.p2pCommunicationSend(xsize, "left");
    sim.p2pCommunicationSend(ysize, "top");

     sim.computation(package_size,
            ComputationType.CPU,
            SoftwareStack.Undefined,
            1,
            linearComplexity,
```

```
                OperationType.Calculations,
                OptimizationType.None);

        sim.p2pCommunicationReceive("center");
        sim.p2pCommunicationReceive("bottom");
        sim.p2pCommunicationReceive("left");
        sim.p2pCommunicationReceive("top");
}
```

1.2.5 Simulations and Results

Exemplary results for the given application are shown in Figure 1.5 with execution times corresponding to particular numbers of processes used. It can be seen, according to Amdahl's law, that for the given data size and parameters of the network interconnect, execution times are decreasing until a certain number of nodes is reached after which the speed-up drops. This is in line with the expected nature of this function which is partly demonstrated e.g. for medical simulations shown in [12, 13].

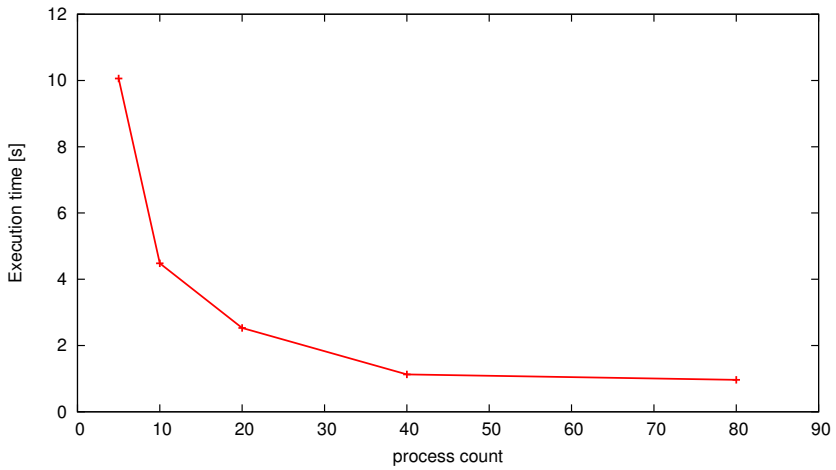

Figure 1.5: Execution time vs number of processes

Apart from these tests, we can verify how execution times change for various partitioning ways of the proposed application. Specifically, as demonstrated in [12], stripe partitioning gives good load balancing as partitioning by using cuts in two dimensions. However, these partitioning ways differ considerably in communication times, especially for larger numbers of processors used. Speed-ups that can be obtained for stripe partitioning are much worse than for the latter. The author has verified this in the simulator by changing the values of the domain size and correspondingly xsize and ysize in the application code and maintaining the total size of the domain. Figure 1.6 presents execution times that were obtained

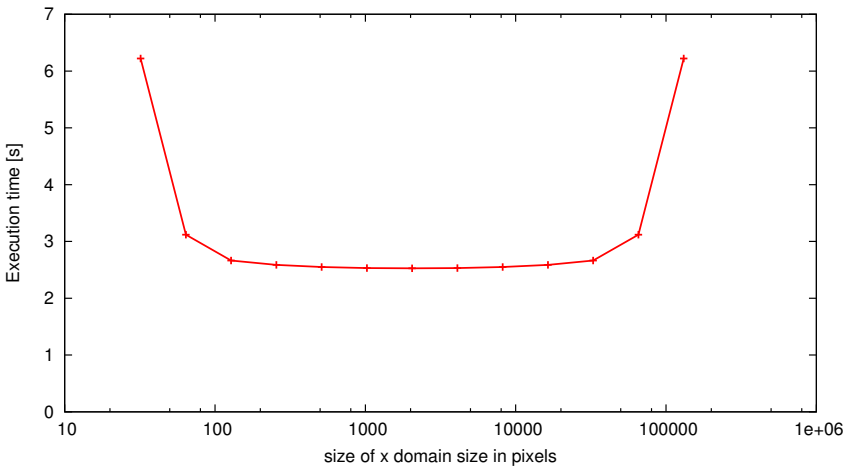

Figure 1.6: Execution time vs x size of domain with a fixed total area of the domain

for various values of xsize while maintaining ysize such that the total size of the domain remains the same. It can be easily seen that the best execution times are obtained for rectangular subdomains which is in line with the practical results that were obtained in [12]. It should be noted that the total size of the domain in these experiments stays the same but the shape changes so that shapes of particular subdomains change correspondingly. It suffices in terms of the goal of this experiment.

These results depict typical dependencies seen in running parallel applications. Further chapters of the book focus on precise modeling of real applications in the MERPSYS environment and validation against results from runs of real parallel applications.

1.3 Summary

The chapter presented a way of representing modern parallel and applications run on large scale parallel systems in the MERPSYS environment. A typical workflow was presented including system modeling, application modeling, running a simulation, calibration against real application runs for selected configurations and using the simulation environment for testing other configurations. Furthermore, an approach was proposed on how master-slave, geometric SPMD, pipelined and divide-and-conquer applications can be modeled within the environment. A case study of geometric SPMD application was presented with the use of the MERPSYS environment. Subsequent chapters build on this approach for even more detailed modeling of geometric SPMD in 3 dimensions, divide-and-conquer as well as master-slave examples with verification of MERPSYS results against results from actual runs performed in a real parallel environment.

Bibliography

[1] Czarnul, P., ed.: Modeling Large-Scale Computing Systems. Concepts and Models. Gdansk University of Technology Press (2013) ISBN 978-83-938367-0-3.

[2] Czarnul, P., Rosciszewski, P., Matuszek, M.R., Szymanski, J.: Simulation of parallel similarity measure computations for large data sets. In: 2nd IEEE International Conference on Cybernetics, CYBCONF 2015, Gdynia, Poland, June 24-26, 2015, IEEE (2015) 472–477

[3] Rościszewski, P., Sidorczak, P.: Simulation of Parallel Applications on Large-scale Distributed Systems. In: Modeling Large-Scale Computing Systems. Concepts and Models. Gdansk University of Technology Press (2013) ISBN 978-83-938367-0-3.

[4] Kreutzer, W., Hopkins, J., van Mierlo, M.: Simjava - a framework for modeling queueing networks in java. In: Winter Simulation Conference. (1997) 483–488

[5] Varga, A.: Omnet++. In Wehrle, K., Günes, M., Gross, J., eds.: Modeling and Tools for Network Simulation. Springer (2010) 35–59

[6] Buyya, R., Murshed, M.M.: Gridsim: a toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. Concurrency and Computation: Practice and Experience **14** (2002) 1175–1220

[7] Bak, S., Krystek, M., Kurowski, K., Oleksiak, A., Piatek, W., Weglarz, J.: Gssim - a tool for distributed computing experiments. Scientific Programming **19** (2011) 231–251

[8] Denzel, W.E., Li, J., Walker, P., Jin, Y.: A framework for end-to-end simulation of high-performance computing systems. Simulation **86** (2010) 331–350

[9] Adalsteinsson, H., Cranford, S., Evensky, D.A., Kenny, J.P., Mayo, J., Pinar, A., Janssen, C.L.: A simulator for large-scale parallel computer architectures. Int. J. Distrib. Syst. Technol. **1** (2010) 57–73

[10] Calheiros, R.N., Ranjan, R., Beloglazov, A., Rose, C.A.F.D., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw., Pract. Exper. **41** (2011) 23–50

[11] Casanova, H., Legrand, A., Quinson, M.: Simgrid: A generic framework for large-scale distributed experiments. In Al-Dabass, D., Orsoni, A., Brentnall, A., Abraham, A., Zobel, R.N., eds.: UKSim, IEEE (2008) 126–131

[12] Czarnul, P., Grzeda, K.: Parallel simulations of electrophysiological phenomena in myocardium on large 32 and 64-bit linux clusters. In Kranzlmüller, D., Kacsuk, P., Dongarra, J., eds.: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 3241 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 234–241

[13] Czarnul, P., Grzeda, K.: Portable parallel simulator using mpi for 2d and 3d domains: design and performance testing. GESTS Int. Trans. on Comp. Sci. and Eng. **15** (2005) 56–64 ISSN 1738-6438.