Chapter 2

P-CAM: A Framework For Parallel Complex Systems Simulations

"If [man] thinks of the totality as constituted of independent fragments, then that is how his mind will tend to operate, but if he can include everything coherently and harmoniously in an overall whole that is undivided, unbroken, and without a border then his mind will tend to move in a similar way, and from this will flow an orderly action within the whole."

-D. Bohm, Wholeness and the Implicate Order

2.1 Introduction

In this chapter we introduce a software framework to support parallel simulation of dynamical complex systems. As already discussed in Chapter 1, most complex systems show emergent behavior making it in many cases impossible to analytically determine the destination of the system. The complexity of such a system renders many questions regarding the system intractable. No closed form analytical solutions can be constructed, which necessitates a simulation approach.

The only remaining possibility is explicitly following the system trajectory through its phase space by a direct simulation. However, due to the complex nature of the system, i.e., the great amount of disparate elements and their mutual interactions, the required computing resources are often enormous. The required scale of correct modeling often depends on problem specific parameters. For example in fluid dynamics, the spatial modeling scale depends on Peclet and Reynolds numbers. In turbulent flow regimes, spatial effects on the smallest length scale play a significant role in the observed macroscopic behavior. Besides such system characteristics, also complex boundaries may result in the failure of analytical calculations. In many other areas, like biology, the importance of microscopic modeling is mostly not caused by a breakdown of
macroscopic formulae, but merely by the lack of such analytical models. For realistic biological modeling there simply is no such thing as a formal description of static or dynamic behavior. Modeling comes about by observing reality and by an intuitive description of the important entities and their interactions. The only possible way of knowing the fate of such models is through explicit simulation.

In this chapter, we design a complex systems simulation framework that allows for a natural mapping between “real” complex systems and complex parallel computer systems. We have implemented the software framework, and use it throughout the remainder of this thesis.

The trajectory of modeling a natural phenomenon to executing this model on a (parallel) computer platform is usually undertaken each time a new system under study is simulated. In Fig. 1.1 (Chapter 1) the trajectory of an application to the parallel machine is schematically given. Ideally, one should employ a fixed generalized path which is instantiated by a specific simulation model. In order to realize such a method of “mapping” a natural system onto a machine, it is common to refer to both the natural system as well as the machine as a complex system [52, 145]. The framework of Dynamical Complex Systems (DCS), should provide a generic interface for “mapping” these systems.

To implement the DCS view we introduce a methodological framework and a software kernel. On the one hand, the framework is used to formalize and experiment with “mapping” issues in DCS. Specifically, for studying parallelization problems like static and dynamic load balancing, discussed in Chapter 3 and Chapter 5 of this thesis. On the other hand, a software kernel is used to implement simulation models transparently on parallel computer systems. Much of the cost and effort in the implementation of a parallel complex system simulations stems from the continuous rediscovery and reinvention of core concepts and components. To circumvent this problem, we introduce a framework suited for building parallel simulation programs. A framework is defined as a reusable, “semi-complete” application that can be specialized to produce custom applications [46]. Similar environments have been developed by other research groups, such as CAMEL [20] and PECANS [21] and Parallel Cellular Automata [68]. However, the emphasis of these frameworks is less on the parallel computing point of view but more on the model specification site. Our approach is based on an efficient use of the parallel computing resources by the complex system model.

In this chapter we introduce a computer implementation of a parallel simulation environment based on a conceptual complex systems framework of interconnected virtual particles as introduced in Chapter 1. The concepts and implementation of our Parallel Cellular Automata Modeling environment (P-CAM) are discussed in Section 2.2. P-CAM enhances modularity by encapsulating volatile implementation details behind interfaces. Its modularity helps improve the parallel application quality by localizing the impact of design and implementation changes, reducing the required effort of understanding and maintaining. Moreover, the re-usability leverages the domain knowledge and prior effort of tedious implementation details in order to avoid re-creating and reval-
idating common solutions to recurring parallelization issues. In Section 2.3, a very important issue in parallel computing, namely, dynamic load balancing (DLB), is discussed in the context of our framework. It is shown that the framework is suited to incorporate a well known DLB strategy and for efficient testing of new heuristics.

2.2 A Parallel Complex Systems Simulation Environment

The concept of DCS, a set of interconnected virtual particles which evolve through time using some execution model, is implemented in P-CAM. In order to satisfy the requirements of DCS, the system must meet several features. Foremost, the definition of a virtual particle can be completely arbitrary, as long as some basic computation can be carried out within this particle. The interconnection structure between particles can be anything from a regular grid to random connected graphs. During the evolution of the system, particles and connections can be created or annihilated. To support parallelism, virtual particles can be allocated to virtual processors. In addition, the allocation is allowed to be dynamic, that is a virtual particle may be re-allocated to another virtual processor. By separating parallel computing functionality from the actual simulation one is forced to use specific framework peculiarities, but there does not have to be any concern about parallelization issues like domain allocation, synchronization, maintaining parallel code, portability and efficient parallel computation. In Fig. 2.1, P-CAM is positioned in the trajectory of constructing parallel code from a given simulation model. In the remainder of this section, the meaning of this figure will be clarified.

2.2.1 Decoupling decompositions, virtual particles and execution models

One of the major design philosophies behind P-CAM is the decoupling of the domain decomposition and the interconnected virtual particles from the actual computation acting upon those particles. The main reason for following this approach is to support the usage of advanced decomposition and graph generation methods for optimal exploitation of application domain knowledge. Both decompositions and particle interconnection depend heavily on the specific (simulation) problem and do not need to be integrated with the parallel communication/computation layer. It enables the application builder to employ sophisticated computer aided modeling techniques to define for example solid obstacles in a virtual (Computational Fluid Dynamics) world. Subsequently, from this world specification, a description of the connected particles can be generated, defining their interconnections. This definition can in turn be fed to a domain decompositioning program, for example ORB, RSB [141] or even a heuristic mapping algorithm like Simulated Annealing (SA) or a Genetic Algorithm
The actual computation carries out calculations on the virtual particles and defines communication points between calculation phases. The execution model is not defined by P-CAM, it may either be a synchronous or asynchronous time stepping scheme or even a parallel discrete event scheme. For efficient parallel computation, P-CAM is able to use a Dynamic LoadBalancer (DLB) for re-allocation of particles, which will be discussed in Section 2.3.

2.2.2 Task interaction graphs and update functions

P-CAM is based on the topological model of task interaction graphs or for short, task graphs. A task graph defines cells and edges between cells. The cells correspond to virtual particles on which useful computations are executed by evolving the local cell states. The presence of an interaction between cells is defined by a shared edge. An interaction between cells is defined either as an exchange of cell states or the alteration of one cell state initiated by the other. The logical order of computations (calculation) on and interactions between cells is not specified by the task interaction graph, i.e., it is only a topological specification. Cellular Automata (CA), for example, are a special instance of a task interaction graph augmented with an execution model to define the order between calculation and interaction. CA reduce to $d$-dimensional grids with a specific neighborhood size defining the presence of edges between cells. In this case, the
2.2 A Parallel Complex Systems Simulation Environment

An execution model is a synchronous time stepping method, alternating between calculation and communication phases. All cells simultaneously update their states, based on their own state and those of their connected cells. The separation between the spatial order imposed by a task graph and the temporal order of the execution model is visualized in Fig. 2.2. The topology of the task graph is taken care of by the P-CAM framework, while the execution model can be chosen rather arbitrarily. Examples of execution models are Continuous Time, Discrete Time and Discrete Event models.

![Spatial Decomposition: P-CAM](image)

**Figure 2.2:** A parallel simulation framework view, based upon the orthogonal relation between spatial and temporal decomposition.

Formally a task interaction graph can be defined as a triple \( C = (N, E, S) \):

- \( N \) is a set of cells, each cell is identified by a cell id \( n_i \).
- \( E \) is a set of edges. Let \( e_{ij} \) be an edge between cells \( n_i \) and \( n_j \), and \( e_{ij} \in E \) iff \( n_i \) and \( n_j \) interact.
- \( S \) is a set of states of arbitrary cardinality, each cell \( n_i \) has a state \( s^t_i \) at “time” \( t \).

To define the computation, we need a transition function \( f_{i:C} : S^{(m+1)} \rightarrow S \), where \( m \) is the number of connected tasks, defined by:

\[
f_{i:C}(s^t_i \cup s^t_j : e_{ij} \in E) = s^{(t+1)}_i
\]  

Furthermore, in the case of a set of transition functions, a permutation or ordering \( \pi \) may be defined according to which the local maps \( f_{i:C} \) may be defined.
The majority of the applications discussed in this thesis use the so called parallel (or synchronous) application of an update functional, as applied to Cellular Automata. For Eq. 2.1 this implies: $a = t - 1$, that is the new cell state is determined from adjacent previous cell states.

### 2.2.3 Decomposing task graphs

Complementary to the task graph, the *initial* allocation of the cells to (virtual) processors must be defined. This requires a decomposition file, specifying for each cell to which processor it is allocated. Following this scheme, each processor ends up with having a possibly unconnected (or even empty) sub-graph of the complete task graph.

The main task of P-CAM is to keep the administration of the current allocation of cells, creating the cell structure and their initial allocation. Additional functionality such as synchronization of all boundary cells is also implemented. It is easy to implement extra functionality, such as communicating states of a specified set of boundary cells, as long as the integrity of the cell administration is certified.

### 2.2.4 Data structure

The basic atomic unit of computation in P-CAM is the cell or virtual particle, where atomic denotes the indivisibility of the cell from a parallel computation point of view. In Fig. 2.3 an example is given of a decomposed task graph of cells that define atomic task graphs. The system is free to re-allocate the atomic cells during a simulation (see Section 2.2.5).

![Figure 2.3: A task graph decomposed over two processors, each cell contains yet another task graph.](image)

A task graph is specified using a task graph specification file, which defines the following:
- the number of cells in the initial task graph.
- the initial connections of the cell to other cells.
- for each cell a \(d\)-dimensional coordinate is optionally assigned.

Complementary to the task graph, a decomposition file must be defined and passed to P-CAM. The decomposition file can be generated from the task graph specification using a graph decomposition tool.

A cell (see Fig. 2.4) in P-CAM has a couple of basic attributes next to the user defined cell state. These attributes include the cell ID, current cell allocation (Processor ID), its neighbors and a flag register containing among others, a cell active flag. This flag is used to disable or enable virtual particles during a computation. A disabled virtual particle is not processed and does therefore not contribute to the integral work load of a processor. In Section 2.3, the number of active cells will be used as a measure for processor work load and to steer a dynamic load balancing algorithm. The actual cell state is specified by the user application.

![Figure 2.4: Schematic representation of a virtual particle and the various linked lists.](image)

Each processor reads cells from the task graph specification, as specified by the decomposition specification. The cells are first stored in a linked list. Next, their connectivities are resolved by setting pointers to neighboring cells in each cell specification. A neighboring cell residing on a foreign processor is found by creating a so called immigrant list of dummy cells. A dummy cell contains all fixed cell attributes and space to store the current value of the cell state.
The cell states in the immigrant list are updated at user specified times. Complementary to this immigrant list, an emigrant list is constructed containing pointers of local cells which contain foreign connections. During a synchronization procedure, all processors send the local cell states in the emigrant lists to their matching neighbouring processors followed by receiving foreign cell states of those cells specified in the immigrant list. Sorting both immigrant and emigrant lists on cell ID, allows fast communications without having to resolve the correct position of a cell in the linked list.

Disadvantages of the approach followed by P-CAM are mainly memory overhead due to memory referencing compared to a direct addressing approach in array implementations. Of course, this disadvantage is only valid for regular task graphs (e.g., grids). Another disadvantage is that arbitrary (non-local) communications among processors will occur, because the framework does not consider processor locality. Major advantages are: no restriction on virtual particle connectivity, no restriction on initial cell-to-processor assignments, arbitrary definition of cell states, dynamic creation/deletion of cells, dynamic creation/deletion of connectivity, dynamic assignment of cells to processors (see Section 2.2.5) and the possibility to skip inactive cells.

**Compressed task graph specification**

Specifying a task graph as discussed in the section above may require a significant amount of disk space. For each cell, its coordinates are stored together with a list of other cells to which it is connected. In the case of a regular task graph, for example a grid, most of the information in the task graph specification will be redundant. For regular task graphs there is a reasonable amount of order present in its specification. In order to reduce the size of a task graph specification we have constructed a so called compressed task graph specification, which is based on a kind of run-length encoding. An example is given in Fig. 2.5.

**Figure 2.5:** Transformation of a 200x200 2D task graph specification to a compressed task graph specification

In this figure parts of a 200x200 2D regular grid specification are depicted. The left side of the figure shows the original specification while the right hand side
2.2.5 Cell migration

In order to support transparent re-assignment of cells to processors, P-CAM supports the migration of an arbitrary set of cells to an arbitrary processor. Cell migration involves packing the cell (and its state) and sending it to the designated processor. Moving a cell to another processor involves notifying all processors that have connections to it. Hence, moving a cell may result in a complex restructuring of immigrant and emigrant lists. Efficient implementation of the cell migration functionality requires solving issues like multiple cell migration interference: expensive restructuring operations should not be applied to cells that are also going to be migrated.

Note that the cell migration facility does not decide when to re-assign cells, it only supports the use of a dynamic load balancing method. Therefore, applications are not tied to specific dynamic load balancing algorithms. Moreover, the framework supports fast implementation and evaluation of new algorithms.

2.2.6 Cell annihilation/creation

During the simulation, it may be true that certain cells becomes obsolete in calculating the new state of the system. In this case, it can be sensible to physically remove the cell from the system, reducing memory use and unnecessary communication with the invalidated cell. On the other hand, the course of the simulation could also require the creation of additional cells, for example due to local grid refinement (e.g., used in grid refinement methods). P-CAM supports both annihilation and creation of cells, which is implemented similarly as cell migration. Cell migration is a combination of cell annihilation and cell creation, augmented with packing and sending the cell and its state to another processor.

2.3 Dynamic Load Balancing

In the previous section, it was shown that the P-CAM application framework allows us to automate the process of deriving (possibly irregular) inter process communication patterns, that are necessary for proper data exchange between connected processes. Another paramount problem in many simulation applications is the fact that the amount of workload associated with the parallel processes can be subject to change during program execution. This change is likely to induce imbalance in the workload distribution and, hence, requires
some action, in order to rebalance the load over the processes.

Although one might be able to realize a good initial partition (or decomposition) of the problem in question, at some point in time, due to the aforementioned changes in the workload distribution, the parallel execution may become unbalanced. The most obvious solution would be to reconsider the whole decomposition problem at this point in time and perform a complete new partitioning incorporating the new workload distribution. However, this solution is usually not practical. The main reason for this is that most “good” partitioning methods, like, for instance, recursive spectral bisection (RSB) [141], take a serious amount of time to be carried out. So, if such an algorithm has to be carried out on a regular basis during runtime, with great certainty it will seriously slow down the program execution. The temporal gain of having a load balanced simulation is completely diminished by the fact that a computationally expensive partitioning algorithm has to be carried out every once and a while. In other words, it is not unlikely that the costs of load balancing in this manner are much higher than the gain of having a load balanced simulation.

Another practical problem with a method like RSB is that a slight change in the workload distribution can easily result in a decomposition that greatly differs from the previous decomposition. In practice, this means that large parts of the computational domain have to be migrated to other processes, such that each process obtains the proper parts of the whole simulation problem. So, a small change in the workload distribution might very well lead to a big change in the decomposition. Hence, a lot of extra administration is necessary to move data to other processes, which can be very costly. Therefore, we need a dynamic load balancing method under “whose” supervision the allocation of atomic processes will only vary “gradually.” Small variations in the workload distribution should only induce small variations in the process allocations. That is, subsequent partitions should “look alike.” Secondly, we need a method which is not expensive to perform, and as such, is not a seriously hampering factor when it comes to real time execution.

In this section, first we will formally introduce the load balancing problem. Next, we discuss two state-of-the-art methods to solve the load balancing problem. The input that both methods get is the workload on each parallel process, and the process connectivity. With this information a “workflow” pattern can be computed. This workflow denotes how much work has to be migrated between every connected pair of processes in order to realize load balancing.

We present four different heuristic methods that can be used to associate the abstract flow of work with an actual selection of atomic processes (or cells) that have to migrated from one processor to another.

After these two steps, the migration of atomic processes between pairs of connected processors can take place. The tedious task of moving lists of atomic processes between the parallel processors is completely carried out by the kernel.
2.3 Dynamic Load Balancing

For the purpose of load balancing we are interested in the workload associated with the atomic processes (or cells), and their connectivity.

For the following the terminology is very important. We assume that we have $|T|$ parallel processes, each of which can be composed of zero or more atomic processes (or cells or ViPs). At this point we make no explicit statement whether the $|T|$ processes are allocated on parallel processors. They may either be executed concurrently on a single multi-tasking processor, or in parallel on a distributed memory computer system (or some combination of both).

Formal Description

We view a parallel simulation program as a graph $H = (T, E)$, with $T$ the set of processes, and $E$ the set of interprocess links. If we consider $P$ to be the set of the atomic processes to be distributed among the parallel processes we can define the following:

- A mapping is a function $\pi : P \rightarrow T$, assigning each atomic process to a parallel process.

- The weight of process $i$ relative to a mapping $\pi$ is defined as

$$\text{weight}_{\pi}(i) = \sum_{\mu \in P} \text{comp}(\mu),$$

(2.2)

where $\text{comp}(\mu)$ denotes the abstract computational load of the atomic process $\mu$, which, for instance, can be expressed in flop. The actual execution time associated with the computational load $\text{comp}(\mu)$ can be calculated when the CPU speed ($\text{CPU speed}$), which is for instance expressed in units $\text{flop/s}$, is known. For the moment we assume that we allocate each parallel process on a unique processor, and that the CPUs are equally fast. Hence, we may replace the weight $\text{weight}_{\pi}(i)$ by the real time complexity, $\text{load}_{\pi}(i)$, which is expressed in seconds.

$$\text{load}_{\pi}(i) = \text{weight}_{\pi}(i)/\text{CPU speed},$$

(2.3)

- The global cost function $\Gamma(\pi)$ is defined as

$$\Gamma(\pi) = \sum_{i \in T} (\bar{t} - \text{load}_{\pi}(i))^2,$$

(2.4)

where

$$\bar{t} = \frac{\sum_{i \in T} \text{load}_{\pi}(i)}{|T|},$$

(2.5)

that is, the average load per process.
The load balancing problem can then be defined as follows:

\[
\text{find a mapping } \pi \text{ which minimizes the global cost function } \Gamma(\pi).
\]

Cybenko [34] has shown that the global minimum for Eq. (2.4) can be reached by parallel optimization of the local load, that is, by locally optimizing:

\[
\Gamma_i(\pi) = \sum_{i,j \in E} (\text{load}_\pi(i) - \text{load}_\pi(j))^2, \tag{2.6}
\]

In the following algorithms pseudo-code is used to explain the various load balancing strategies. For this purpose we introduce the following definitions.

(1) Definition: The flow of work from process \( i \) to process \( j \) is denoted by \( \text{Workflow}[i][j] \).

(2) Definition: The work on process \( i \) is denoted by \( \text{Work}[i] \) and is equal to the load given by Eq. (2.3).

Two methods to determine the workflow are considered. In both methods it is assumed that it only is sensible to allow workflow between parallel processes that are connected, that is, pairs of processes that fall in the set \( E \). In other words, if two processes are disconnected no direct workflow between them is allowed.

**Workflow by Poisson Iteration**

In Fig. 2.6 the pseudo code for a “Poisson-like” iteration process that is used to compute the workflow is given. The term “Poisson-like” follows from the fact that this iteration closely resembles an algorithm that can be used to solve the Poisson equation numerically. This equation describes the way that heat is transferred between regions of different temperature in physical systems (e.g., a plate of metal). In our case we don’t have a difference in temperature, but a difference in workload. The workflow that is calculated is completely analogous to heat flow in physics.

Let us turn to Fig. 2.6.

(1) The first loop simply checks whether the load is already balanced.

(2) The next loop iterates over all processes.

(3) The third loop iterates over the neighboring (or connected) processes of process \( i \).

(4) The value \( \text{index} \) is the real process ID of a process with relative ID \( j \), which can be found from the ConnectionList\([i][j]\). That is, ConnectionList\([i][j]\) holds the ID for the \( j \)-th neighbor of process \( i \).

(5) The \( \text{WorkDifference} \) expresses the amount of work that is “shipped” in one Poisson iteration from process \( i \) to process \( \text{index} \). \( \max deg \) expresses the maximum degree of connectivity in the process connectivity graph. This value has to be constant for the following reason: due to the fact that the inter process communication pattern can be irregular, the degree of connectivity of the parallel processes is not a constant, while the workflow following from the iteration algorithm has to be symmetric, that is \( \text{Workflow}[i][j] = -\text{Workflow}[j][i] \).
2.3 Dynamic Load Balancing

(1) while (NotLoadBalance())

(2) for i = 1 to number of processes

(3) for j = 1 to number of processes connected to process i

(4) index = ConnectionList[i][j]

(5) WorkDifference = (OldWork[i] - OldWork[index]) / α * max_deg

(6) Workflow[i][index] += WorkDifference

(7) Work[i] -= WorkDifference

(8) for i = 1 to number of processes

(9) OldWork[i] = Work[i]

Figure 2.6: Pseudo code for workflow by Poisson iteration.

This requirement can only be satisfied by introducing the max_deg variable. The factor α is used to account for numerical stability of the Poisson iteration and has to be larger than 1 [34].

(6) The Workflow[i][j] between connected processes i and j must be cumulated over the contributions of successive iterations.

(7) The value Work[i] has to be decreased with WorkDifference.

(8) and (9) a loop to set the proper values in OldWork for the next Poisson iteration.

The algorithm is assumed to have reached convergence if Work[i] ≈ 1, i.e., if each process contains the average workload. Formally, this pseudo code is written down in Eq.(2.7), where we consider the workload on process i after iteration n + 1 as a function of the workloads after iteration n on process i and the processes connected to it.

\[
load_{n+1}(i) = load_n(i) - \sum_j L_{ij} load_n(j) / (α * max_deg),
\]

where

\[
L_{ij} = \begin{cases} 
-1 & \text{if } i \neq j \\
\text{deg}(i) & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]

L_{ij} is known as the Laplacian matrix, since it resembles the discrete analog of the Laplacian operator (\(\nabla^2\)), which is present in the Poisson equation.

Workflow by Constrained Optimization

An elegant alternative method to derive a workflow that results into proper load balancing has been suggested by Hu et al. [74]. Their method works as follows.

Firstly, a vector \(\vec{b}\) (or array) is defined, where
that is, component \( b_i \) is equal to the workload on process \( i \) minus the average workload. Before we formalize our problem, first take a look at the following example in Fig. 2.7. We observe a directed graph, connecting four vertices (processes) by means of five directed arcs. The direction of each arc is from the vertex with the highest number to the one with the lowest number. The total flow of workload towards a vertex is the summation of the work-flows over the arcs ending and starting at this vertex.

If the workflow over an arc has a positive value, this means that the work flows in the direction of the arc, whereas if the value is negative this means that the work is flowing in the opposite direction.

We introduce the following matrix \( A \) to formalize the “directedness” of the workflow graph.

\[
A_{ij} = \begin{cases} 
1 & \text{if vertex } i \text{ is the start of arc } j \\
-1 & \text{if vertex } i \text{ is the end of arc } j \\
0 & \text{otherwise}
\end{cases}
\]

Hence for the example in Fig. 2.7, we find the following for matrix \( A \):

\[
A = \begin{pmatrix} 
-1 & 0 & 0 & -1 & -1 \\
1 & -1 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0
\end{pmatrix}
\]

Next, we define the workflow vector \( x = (x_1, x_2, \ldots, x_n)^T \) as the unknown in our load balancing problem, where it is assumed that the graph contains \( n \) arcs. \( x_i \) is equal to the amount of work that arc \( i \) has to carry in order to solve the load balancing problem.

With these definitions, it is not difficult to see that in order to reach load balancing, we have to demand that

\[
b_i = \sum_j A_{ij} x_j,
\]

or in matrix notation:

\[
Ax = b.
\]

Since the number of arcs can be larger than the number of vertices (see e.g., Fig. 2.7), in principle the solution vector \( x \) can have infinitely many solutions (as the matrix \( A \) is not square).

In order to cope with this problem we consider Eq.(2.11) as a set of linear constraints to a quadratic optimization problem, namely find a workflow vector \( x \) with minimal Euclidean norm \( x^T x \), under the constraint that it satisfies Eq.(2.11). It is found that this constraint requires that [74]:

\[
b_i = \text{load}_i(i) - \bar{I},
\]
\[ x = A^T \lambda, \quad (2.12) \]

where \( \lambda \) is known as the vector of Lagrange multipliers for the constraint. As a consequence, the problem of finding optimal load distribution becomes that of solving Eq. (2.13). It is not difficult to see that \( L = A^T A \) is equal to the well known Lagrangian matrix, of the previous subsection.

\[ L \lambda = b, \quad (2.13) \]

The workflow between processes \( i \) and \( j \) is equal to \( \lambda_i - \lambda_j \). The vector \( \lambda \) can be solved with any matrix inversion algorithm of choice. In our implementation it is solved with the well known conjugate gradient method.

\[ \text{Figure 2.7: 4 processes connected by 5 uni-directional arcs.} \]

### 2.3.2 Cell Selection: Prerequisites

After the workflow has been determined, the next question is, given the workflow, how to select cells from the overloaded processes that are most suited for migration to under-loaded processes. In the following, four different heuristic methods are proposed as a solution to this problem. At this point we can use the advantage of our cell based application framework. Namely, it allows us to define a clean and simple interface to the cell selection methods, which makes it easy to “plug and play” with the different strategies. No alterations to the kernel code are necessary.

As the basis of the cell selection heuristics, we assume that it is more sensible to migrate cells between processes that lie on (or close to) the inter process
boundaries. This idea, amongst others, finds support in the work by Walshaw et al. [156], where comparable assumptions are made for selecting “migratable elements” for dynamic load balancing in finite element meshes.

In addition to the connectivity of the cells, it is possible that with each cell a coordinate in space can be associated, for instance, if cells in fact correspond to a phenomenon that is simulated in Euclidean space. At this point, a cell selection strategy might incorporate this a priori knowledge and take advantage of it.

To explain the functionality of each of the four selection methods below, we take one example problem, using it over and over again. Furthermore, for each method we provide a pseudo code, explaining it by describing its operation in case of the example.

The cell oriented view maintained by the kernel allows for easy maintenance of a number of key characteristics of all domains. Amongst others, for each process, lists are kept up to date specifying which local cells lie on inter process boundaries. As we will see below, this is very convenient for the process in which we select cells to account for workflow.

Given two processes $i$ and $j$, depicted in Fig. 2.8, with process $j$ consisting of the black cells, and process $i$ of the white cells. We assume that each cell has an equal workload. Furthermore, $\max{\deg}$ again is the maximum number of cells that one cell can be connected to (i.e., $\max{\deg} = 4$ in this case.). Clearly, the example system is out of balance. $j$ has 13 black cells, vs. 17 white cells on $i$.

Without too much effort we can imagine that the workflow will be directed from process $i$ to process $j$, and be of size 2, which corresponds to 2 cells (in general the size of the workflow will have to be provided by any of the above workflow algorithms of the previous section. For this simple example we can solve it by hand).

Note that in Fig. 2.8 only the cells on the boundary in process $i$ are numbered.
2.3 Dynamic Load Balancing

(1) SelectedWork = 0

(2) while (Workflow[i][j] > SelectedWork)

(3) NextCell = BoundaryList[index = Random(ListLength)]

(4) Lock(index)

(5) SelectedWork = SelectedWork + Work(NextCell)

(6) if (BoundaryList == Empty AND SelectedWork < Workflow[i][j])

(7) CreateNewBoundaryList()

Figure 2.9: Pseudo code for random cell selection.

2.3.3 Graph Based Selection Methods

Random

The only a priori knowledge that is put into the selection strategy, of which the pseudo code is found in Fig.2.9 is that we have a list of cells on process \( i \) that are connected to cells on process \( j \), called BoundaryList. Next, a description of the pseudo code follows, step by step.

(1) The amount of selected work is initialized to zero.

(2) Check whether the workflow from process \( i \) to \( j \) is positive.

(3) From the BoundaryList a random cell is drawn.

(4) This cell is locked, since it has been selected, and hence may not be chosen in subsequent iterations.

(5) The work associated with this cell is added to the total amount of selected work.

(6) The only guarantee we have, is that cells on the domain edge are selected. If it might turn out that we run out of boundary cells, that is, BoundaryList == Empty, before the workflow is satisfied, we must start to “peal off” the next layer of cells, therefore (CreateNewBoundaryList()) is invoked.

Projecting this algorithm on our example in Fig. 2.8, we know that we have to select two cells from process \( i \) on the boundary. Clearly, any pair of boundary cells can be the result of this selection scheme.

Breadthfirst

In the next heuristic we apply a cell selection method that we toss as “breadthfirst.” The algorithm depicted in Fig.2.10 is structured as follows.

(1) A cell is selected randomly from the boundary list.
(1) FirstCell = BoundaryList[index = Random(Listlength)]
(2) Queue = EnQueue(BoundaryList, FirstCell)
(3) while (Workflow[i][j] > SelectedWork)
   (4) NextCell = DeQueue(Queue)
   (5) Lock(NextCell)
   (6) SelectedWork = SelectedWork + Work[NextCell]
   (7) if (BoundaryList == Empty AND SelectedWork < Workflow[i][j])
      (8) CreateNewBoundaryList

Figure 2.10: Pseudo code for breadth-first cell selection.

(2) Next, a queue is formed in which in ascending order the neighbors of the first cell are stored, with the constraint that these neighbors are on the domain boundary. That is, the neighbors of the first cell follow first, then the neighbors' neighbors and so on.

(3) The loop is entered which continues until enough work has been selected to satisfy the workflow.

(4) – (8) The algorithm dequeues cells from the queue, one at a time, until the total selected work has passed the workflow threshold. Analogously, to the first method the BoundaryList is updated when we run out of cells, before the workflow is satisfied.

Projecting this algorithm on our example in Fig. 2.8, we know that we have to select two connected cells from process /' on the boundary. Clearly, any pair of connected boundary cells can be the result of this selection scheme.

Categorical

The third selection heuristic that utilizes purely the connectivity information of the cells is tossed as “categorical” selection. The algorithm in Fig. 2.11 operates as follows.

(1) For each boundary cell it is investigated “how strong” it is connected to its local process, and how strong it is “pulled” by one or more alien domains. For instance, in Fig. 2.8, cell 1 has two connections with process i, whereas it only has one connection to process B. On the other hand, cell 2 has one connection to process i and one to process j, and in that sense no specific stronger “desire” to be on any of the two processes. Finally, cell 3 is connected quite strongly to process i, since it has three connections to this process, opposed to one connection to a cell on process j. In this way we can “categorize” the boundary cells, from weakly
(1) CategorizedList[M] = Categorize(BoundaryList)

(2) FirstCell = WeakestConnection(CategorizedList, Random)

(3) Queue = EnQueue(BoundaryList, FirstCell)

(4) while (Workflow[i][j] > SelectedWork)
    (5) NextCell = DeQueue(Queue)
    (6) Lock(NextCell)
    (7) SelectedWork = SelectedWork + Work[NextCell]
    (8) if (BoundaryList == Empty AND SelectedWork < Workflow[i][j])
        (9) CreateNewBoundaryList

Figure 2.11: Pseudo code for categorical cell selection.

connected (no connections with any cell on the host party; note that it is, in prin­
ciple, possible that a cell is completely surrounded by “alien” cells), to strongly
connected (max_deg - 1 connections with the host process,; not max_deg, since
any cell with max_deg neighboring cells that are all on the same host process is
an internal cell and, hence, cannot be present in the BoundaryList.).

(2) - (9) After this categorization, in essence the same algorithm as in the
breadth-first method is carried out. The difference is that the FirstCell is se­
lected from the list of weakly connected cells, instead of being “any” cell on the
boundary of process i. In the example this means that cells 2 and 4 are first
candidates as “starting cells” to start a breadth-first selection from.

2.3.4 Center Of Mass Based Selection Methods

In the case that we can associate a set of coordinates with each computational
cell in our system, we can take advantage of this a priori knowledge, in addition
to the purely “graph-based” view applied in the previous methods. For this pur­
pose, we have devised a method that operates around the center of mass (COM)
of parallel processes.

COM

The pseudo algorithm depicted in Fig.2.12 operates as follows:

(1) Each parallel process:

(2) computes the location of its private center of mass in the simulation space.
This CenterOfMass, or com is calculated as follows:
On each process

CenterOfMass = ReturnCenterOfMass(Cells)

SEND(CenterOfMass, Neighbors)

RECV(CentersOfMass, Neighbors)

For each Local Cell

ClosestProcess = DetermineClosestNeighbor(Cell, CentersOfMass)

List[ClosestProcess] = AddCellList(ClosestProcess)

For each NeighborProcess

SortonDistance(List[NeighborProcess])

Figure 2.12: Pseudo code for the center of mass cell selection.

\[
\bar{c}m = \frac{\sum_{i=1}^{N} m_i \bar{x}_i}{N},
\]

where the summation runs over \( N \) local cells with coordinates \( \bar{x}_i \) and masses \( m_i \).

In our example we take the cell-mass as a constant.

Next, each process notifies its Neighboring processes about the value of its own CenterOfMass, and stores the centers of mass that it receives from its neighbors.

Then, for each local cell it is determined to which neighboring center of mass, besides that of the local process, it is closest. And for each neighbor process a list is created which contains the IDs of the “closest” cells.

Finally, each of these lists are sorted in ascending order. That is, the cells closest to a neighbor process are first in the list.

If a workflow from one process to another is required, we simply take as many subsequent cells from the List that was sorted on distance, as the corresponding workflow requires.

### 2.3.5 Edge Smoothing

Since inter process edges are likely to become rugged after applying subsequent dynamic load balancing steps, incorporation of an additional algorithm that can smoothen out these edges can be helpful, in order to minimize the communication. For this purpose, we apply a filter that runs over the inter process edges,
after the work-flows have been carried out physically. The action applied by this filter is the following.

First, for each boundary cell its “preference” is determined, which means that we search to which process it is most strongly connected. If we look at Fig. 2.8, we see, for example, that cell 1 has a preference to be on process \( i \), whereas cell 2 doesn’t have a preference to be either on process \( i \) or process \( j \), and the preference of cell 3 is again process \( i \).

From the opposite side, namely process \( j \), clearly the black cell that is surrounded by white cells 2, 3, and 4 has a strong preference to be on process \( i \). Hence, the result of the edge smoothing procedure on process \( j \) will be that this cell is transported to process \( i \).

Note that, the edge smoothing step will usually harm the balance in the workload that has been realized by the workflow and cell selection algorithms. In Chapter 5, where we review the pros and cons of the various heuristics, the consequences of edge smoothing are investigated experimentally, to see to what extent it is useful or (perhaps) damaging.

### 2.3.6 Cell Migration

As soon as the inter process workflow has been determined, and for each workflow cells have been selected, these cells can be migrated from one process to another. The kernel handles all tedious administrative details that are related to the physical (from one process to another) migration of cells (see Section 2.2).

### 2.3.7 When to Balance?

An entirely different story, which is very important in the case of load balancing is the decision-making part. That is, what are appropriate moments to carry out a full dynamic load balancing procedure. Obviously, this is strongly connected with the specific dynamical work load behavior of the simulation in question. If, for instance, at a certain moment in time one of the processes suddenly experiences a serious increase of workload, it is best to spread out this extra work evenly over the processes. We will illustrate this with an example (see Fig. 2.13). A local workload increase of say 40%, will result in a parallel simulation that takes 40 % more time to execute. Spreading the extra load evenly out over, say, 4 processes will (optimally) result in a execution gain of \( \approx 20\% \).

On the other hand, if workload “disappears” at one process, say, suddenly one process has 40% less workload than the others, the consequences will be much less dramatic than in the previous case. If we have 4 processes still executing 100%, and only one process for 60%, the execution gain will be maximally 10%. Besides the fact that we may be confronted with load imbalance “as such,” there are a few more problems that must be covered. For one, we need some monitoring device that keeps track of the load imbalance in our system. That is, we need to “probe” if load imbalance is present. Clearly, there is a trade-off in the probe-frequency, the expenses (in terms of extra execution time) that have to be paid, the dynamics of the application (does workload change gradually or suddenly?).
So with respect to the issue, we can state that it represents a great number of unresolved questions. Unfortunately, in this chapter we are not in the position to fully investigate this subject. However, it is a very important part of the future work with respect to general dynamic load balancing strategies for our integrated parallel simulation environment.

2.4 Conclusions

In this chapter we have introduced a software framework, intended to simplify the parallelization of complex simulation models both practically as well as methodologically. The practical aspect lays in the fact that cumbersome administrative parallelization issues are hidden from the simulation model, enabling a complex simulation to exploit computational concurrency at a minimum cost. On the other hand, the framework facilitates both the analysis and design of existing and new methods. Important problems in parallel computing, like load balancing can be studied by experimentation and new heuristics may be easily tested. Different schemes or execution models are possible, for example with an asynchronous communication structure. Note that P-CAM only dictates the
formulation of the simulation model as a task graph, i.e., constraints the topology. Issues like causality and update functional ordering should be dealt with by an appropriate execution model. Fundamentally, P-CAM does not exclude other execution models, like asynchronous updating occurring in discrete event simulation models. In many cases an event driven approach is more appropriate for following the phase space trajectory of the simulation. In the situation where events are not continuously generated, but instead have a non uniform arrival rate, an update strategy based on events is often more efficient. In cases, involving asynchronous updating, discrete event simulations are the only way to go. For example, asynchronous cellular automata can be efficiently simulated using a parallel discrete event approach (see Chapter 6) [119]. Effectively, this results in a load balancing of timed events, which can be called temporal load balancing. It may be fruitful to combine these spatial and temporal load balancing strategies, to realize an even more efficient mapping between natural complex systems and complex parallel computers. The realization of such a combined framework may be a step further ahead in creating a generic complex systems simulation environment.
Figure 2.18: Result of redistributing 40% load decrease (A) and a 50% load increase (B) on a specific processor.

So with respect to the main, we can state that it represents a great number of unresolved questions. Unfortunately, in this chapter we are not in the position to fully investigate this subject. However, it is a very important part of the future work with respect to general dynamic load balancing strategies for our integrated parallel simulation environment.

2.4 Conclusions

In this chapter we have introduced a software framework, intended to simplify the parallelization of complex simulation models both practically as well as methodologically. The practical aspect lays in the fact that controversial administrative parallelization issues are hidden from the simulation model, enabling complex simulation to exploit computational efficiency at a minimum cost. On the other hand, the framework facilitates both the analysis and design of existing and new methods. Important problems in parallel computing, like load balancing can be studied by experimentation and new algorithms may be easily tested. Different schemes or execution models are possible, for example with an asynchronous communication protocol. However, P-CAM only dictates the