Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

A NON-FRAGMENTING PARTITIONING ALGORITHM FOR HIERARCHICAL MODELS

Roland Ewald
Jan Himmelspach
Adelinde M. Uhrmacher

University of Rostock
18059 Rostock, Germany

ABSTRACT

The simulation system JAMES II is aimed at supporting a range of modeling formalisms and simulation engines. The partitioning of models is essential for distributed simulation. A suitable partition depends on model, hardware, and simulation algorithm characteristics. Therefore, a partitioning layer has been created in JAMES II which allows to plug in partitioning algorithms on demand. Three different partitioning algorithms have been implemented. In addition to the well known Kernighan-Lin algorithm and a geometric approach, a partitioning algorithm for hierarchically structured models has been developed whose performance is evaluated.

1 INTRODUCTION

Experimenting with large or computational intensive models requires special mechanisms due to the restrictions a single machine implies. One solution is the distributed simulation of the model. A model is split up into partitions which are computed on different machines, so that the overall computation time for the model is decreased. Several aspects must be considered thereby, otherwise the computation time may even increase. One of the most important aspects is the network communication cost factor (Nicol 1998). Partitioning research has been done for over three decades, thus quite a lot of different approaches have been developed so far (Schloegel, Karypis, and Kumar 2000). Similarly to the efficiency and effectiveness of parallel distributed simulation, the way of partitioning depends largely on the characteristics of the model and the hardware. E.g., if the model graph consists of highly symmetrical subgraphs, specialized hierarchical partitioning approaches can be applied (Lemeire et al. 2004).

However, most partitioning algorithms have been developed for general graphs (Fjällström 1998), and do not make use of the particular constraints that some modeling formalisms put on the model structure. Instead, they are aimed at working well for a broad class of problems.

Not only the structure of the model, but also the type of simulation has an impact on the partitioning (Boukerche and Tropper 1994). However, typically partitioning algorithms in simulation are only evaluated in the context of a particular modeling formalism and simulation engine, e.g., (Bailey, Briner, and Chamberlain 1994). To evaluate different partitioning algorithms for different formalisms and simulators, a flexible partitioning system is required that supports to plug in different partitioning algorithms on demand. In addition, such a partitioning system will allow to tailor partitioning algorithms to the characteristics and demands of specific models, simulators, and hardware environments.

Flexibility of simulators has been achieved by exploiting software patterns, like the template method (Himmelspach and Uhrmacher 2004a), which also lend themselves for designing a partitioning layer, e.g., (Li, Huang, and Tropper 2003), and a set of partitioning algorithms. This is an idea we will adopt in developing a partitioning layer for the simulation system JAMES II.

The paper is structured as follows: first we will shortly describe JAMES II, then the architecture of the partitioning layer will follow. The main part of the paper is dedicated to a new partitioning algorithm, which exploits the hierarchical structure of DEVS models. The developed partitioning algorithm will be evaluated based on the imbalance and cut size indices, and in comparison with other algorithms.

2 BACKGROUND - JAMES II

JAMES II has been developed as a modeling and simulation framework which supports a variety of modeling formalisms and even more simulation algorithms. Currently, simple cellular automata models and several DEVS (Zeigler, Praehofer, and Kim 2000) based modeling formalisms can be used for modeling. The different simulators implemented so far (Himmelspach and Uhrmacher 2004a, Himmelspach and Uhrmacher 2004b) mainly support the DEVS based formalisms. The focus of these simulators has been on the parallel distributed simulation of models, e.g., of models of multi-agent systems.

As stated in (Himmelspach and Uhrmacher 2004a), the integration of external processes (e.g., needed for testing software Uhrmacher, Röhl, and Himmelspach 2003) imposes some constraints on the simulator tree, e.g., if the external process is only available or accessible on one specific host.

The different simulators in JAMES II have different advantages and drawbacks which should be considered if a parallel and distributed execution of a model shall be done. Due to the variety and the idea to even increase the number of the available modeling formalisms and simulation algorithms, a highly flexible and adaptable partitioning layer is required.

3 THE STRUCTURE OF THE PARTITIONING LAYER

The support of different modeling formalisms and simulation algorithms needs to be reflected in the partitioning layer because of their differing characteristics. For being able to use different partitioning algorithms, a mechanism must be provided which allows the selection of an algorithm during simulation start up - depending on model and hardware properties and user wishes. The partitioning layer in JAMES II has been designed by applying the abstract factory pattern (Gamma et al. 1994).

The instantiation process of a distributed execution can be split up into two phases: (1) Hand over the model, infrastructure information, and user settings to the partitioning factory. (2) Hand over the partitions to an appropriate simulator factory. The partitioning process itself has been split up into three parts: (1) Infrastructure analyzer: what does the network structure look like? (2) Model structure analyzer: what does the model look like? (3) Partitioning algorithm: create partitions according to the results of the analyzing process.

The infrastructure analyzer computes network latencies and the availability of resources: information which is required for being able to create good initial partitions. It returns a graph whose nodes represent the available processors. The nodes are labeled with the computing power of the corresponding processor, whereas network topology and communication capacities are represented by labeled edges between them. In a first implementation, this algorithm is realized as a stub which returns a complete graph holding the same resource information for all items. The model structure analyzer maps a given model on a graph for further processing. The weight of edges reflect the communication costs and additional properties are attributed to nodes to facilitate further processing, e.g., to take specific constraints during partitioning into account. Having a hierarchical model, these properties may either be propagated from father nodes to children nodes or vice versa. An example for a bottom-up propagation is the usage of one of EPI (external process interface) DEVS models, which enforces the use of appropriate simulator components for all parent models (Himmelspach and Uhrmacher 2004a). Finally, the partitioning algorithm splits up the model graph into partitions to be computed on the available resources. Besides the approach presented in section 4.2, the following partitioning algorithms have been implemented:

Kernighan-Lin (KL, Kernighan and Lin 1970): This classical partitioning algorithm creates bisections (i.e., 2-way partitions), but can be used recursively to generate partitions for more processors. The original version – which was implemented here – does not consider node costs, but the number of nodes instead (there are several extensions, as described in Fjällström 1998).

A geometric approach: Usually, geometric approaches can only be applied to graphs whose nodes have spatial coordinates. We overcome this problem by generating a node list using the Cuthill-McKee approach (Cuthill and McKee 1969), i.e., we generate coordinates for a one-dimensional space. At first, the node with the smallest communication cost is added to the node list. Its neighbors are added to a second list, which is sorted by the communication cost (lower communication costs first). After the first node of the second list has been moved to the node list, all of its neighbors which are not in one of the two lists are sorted by the communication costs as well, in a third list. Then, the third list is appended to the second list. Again, the first node of the second list is added to the node list, whereas a sorted list of its formerly ’unknown’ neighbors is appended to the second list. This process continues until the second list is empty, which means that all nodes are sorted in the node list. Afterward, the node list is partitioned into partitions of equal size.

The separation into three parts reduces the complexity of the partitioning process and facilitates to experiment with algorithms that vary only with respect to one or two parts of the partitioning process. Thus, given an infrastructure analyzer and a model structure analyzer, the impact of different partitioning algorithms can be evaluated more easily. Based on this information, it is possible to select a suitable partitioning algorithm that takes model, simulation and hardware characteristics into account.

The selection is done by certain factories, which again are created by factories (see Abstract Factory Pattern, Gamma et al. 1994). This allows us to develop different sets of algorithms in combination with a suitable selection mechanism for them (e.g., regarding model size, model formalism, simulation method, graph properties, etc.). Due to this flexibility, heuristics that take advantage of those properties may be integrated in all parts of the partitioning process.

4 A NON-FRAGMENTING PARTITIONING ALGORITHM

JAMES II supports the modeling formalism DEVS, which supports a hierarchical modular construction of models. A hierarchical construction of models is of increasing importance the more complex the models become. Thereby, the complexity of models refers to the problem to describe its overall behavior even if the number and the heterogeneity of behavior and interaction patterns of sub-models is well known (Edmonds 1999). A hierarchical model structure can be interpreted as a first partitioning which a modeler has created according to semantical differentiations, and for keeping the complexity in designing a model at bay.

4.1 Related Work on Hierarchical Models

Besides the more general partitioning algorithms, several special partitioning algorithms for hierarchical models (e.g., DEVS models Zeigler, Praehofer, and Kim 2000) have been proposed as well (Kim et al. 1995, Park and Zeigler 2003). But these algorithms have several preconditions that do not necessarily apply for JAMES II: Firstly, models are computed by using the abstract simulator (a processor is associated with each model, which implies that each (sub)model can be freely placed on a different computer and thus the model can get fragmented). Secondly, models have no dependencies to non - model resources (e.g., an external program only installed on one of the available computers).

JAMES II supports, among other modeling formalisms and simulators, the execution of simple PDEVS models based on the traditional abstract simulator, i.e., each model is associated with a simulator in a processor tree. Yet, the use of this plain abstract simulator is not recommended for more complex models, because this simulator requires a thread per model for simulating it. I.e., even if a complete subtree has to be computed on one computer, each model requires a separate thread. Therefore, other simulators, e.g., a sequential variant, have been implemented. JAMES II also supports the testing of software. Software is interacting with the simulation system via *external process interface* (EPI) models. To simulate these models, we need the capability to consider constraints during the partitioning process: e.g., a software might only be executed on one of the available computers – thus, its associated EPI model should not be placed freely.

Furthermore, existing partitioning algorithms for DEVS (Kim et al. 1995, Park and Zeigler 2003) concentrate on partitioning the computation load. However, the communication costs in distributed simulation are often more crucial (Nicol 1998). Especially, if wide area networks (WANs) come into play and communication intensive simulators are used, like the typical DEVS simulators and its variants. The processing of DEVS models (Zeigler, Praehofer, and Kim 2000) is rather communication expensive, pulsing up and down the processor tree. Thus, the number of messages which need to be propagated through the network for computing an imminent model's (a model with an internal state transition – triggered by a star message) state is at least four, for an influenced model (a model with an externally triggered state transition – triggered by an x message) it is at least two for PDEVS.

Let's assume that we have got three coupled models, each containing 1000 sub models. In each coupled model there is one imminent model, and 50 models are influenced by each of them. Hence, inside those three coupled models we need three star messages, three y messages, 153 x messages, and 153 done messages (312 messages altogether). Taking the communication of the three coupled models with their parent into account, we need 12 more messages. If the models are randomly placed on hosts, we could end up with 324 messages to be sent across the network. In contrast, if all children would be placed on the processor which hosts their coupled model, we would only end up with 12 messages to be sent over the network.

4.2 The Basic Algorithm

In the following, we consider an arbitrary planar and upward representation of the model tree, i.e., a tree representation without overlapping edges and the root at the top. An example tree is given in figure 1 (I1). The nodes of each tree level can be ordered according to the tree representation. If node A is situated left of node B ($A < B$), both nodes either have the same parent, or the parent of node A is situated left to the parent of B on the level above.

The basic idea of the algorithm is to consider the distances between the nodes of a level. The distance between two nodes is the length of the path between them, i.e., the number of edges. Since both nodes are on the same level, the distance d between them is an even number. It indicates the level of the tree in which their first common parent is situated, $\frac{d}{2}$ levels above. Thus, a high distance between two nodes marks a borderline between two branches of the tree. This correlation can be used to partition a level for p processors by splitting it at the p largest distances between its nodes (see figure 1 (I1)). To apply this approach, a suitable level for partitioning has to be found. Moreover, the partitioning of nodes which are not situated on the partitioning level needs to be specified.

The basic algorithm has three phases (as shown in algorithm 1), which address the following issues: The first step is finding the level on which the tree of the model shall be partitioned (`level`). This is done top-down, as described in algorithm 2. Afterward, the model tree is partitioned by using `level`. The models in the tree below the level are assigned to the available computing resources by trying to minimize the number of edges between the

Algorithm 1 Phases of the Basic Algorithm

```
1  Function Partitions partition (model, resources)
2    levelIndex := findPartitioningLevel (model,
          resources)
3    partitions := partitionizeLevel (model, resources,
          getLevel(model, levelIndex), null, false)
4    partitions := assignUpperLevels (model, resources,
          levelIndex, partitions)
5    return partitions
```

partitions (see alg. 4). Thereafter, the levels above `level` have to be assigned to the available partitions. This is done in the third step (see alg. 6).

Algorithm 2 findPartitioningLevel

```
1  Function Level findPartitioningLevel(model,
        resources)
2    levels := getLevels(model)
3    for levelIndex := 0 to size(levels)-1 do
4      if criteriaFulfilled(levels, levelIndex,
            size(resources)) then
5        return levelIndex
```

In the first phase, the algorithm computes the level with which the partitioning process shall start with. This level is the first level the `criteriaFullfilled` method returns true for (see alg. 3).

Algorithm 3 criteriaFullfilled

```
1  Function Boolean criteriaFullfilled (levels,
        levelIndex, numOfPartitions)
2    if size(levels[levelIndex]) > ParamNodeNum *
          numOfPartitions then
3      return true
4    if (levelIndex = size(levels)-1) or
          (size(levels[levelIndex+1]) <
          size(levels[levelIndex])) then
5      return true
6    return false
```

The `criteriaFullfilled` function returns true if the last level is reached, the number of nodes in this level is greater than the number of nodes in the next level, or the size of the given level (the number of nodes) exceeds the number of partitions to be generated, multiplied by the parameter `ParamNodeNum`. This parameter denotes the average node number per partition for the partitioning level, i.e., it controls the desired graining.

The core of the algorithm is the `partionizeLevel` function (see algorithm 4). It takes the previously calculated parameters and partitions the given model according to them. At first, the maximum number of partitions is computed as the minimum of the number of processors and the number of nodes on the starting level of the partitioning process (line 2). This ensures that not more partitions are created as processors are available and it ensures that – even if this means that some processors are not used – some complete subtrees will exist. Afterward, the split indices are computed by using a threshold and the aforementioned distances between the nodes on the given level (l. 3).

Algorithm 4 partionizeLevel

```
1  Function Partitions partitionizeLevel (model,
        resources, level, partitions,
        assigningUpperLevels)
2    maxPartBlocks := min (size(resources), size(level))
3    splitIndices  := getSplitIndicesByDistances (level,
          maxPartBlocks - 1)
4    if (assigningUpperLevels) then
5      partitions  := partIndirectlyAffectedNodes(level,
            partitions, splitIndices,
           assigningUpperLevels)
6    partitions    := greedyAssignment (model,
          resources, level, partitions, splitIndices)
7    return partitions
```

The actual partitioning is done by two functions: Firstly, `partIndirectlyAffectedNodes` (l. 5) assigns the level segments by calculating the best processor for each one and assigning its nodes if possible (see algorithm 5). Each processor can be used, as long as it has not already been used during the current execution of the function. To calculate a suitable processor for a segment , the `voteForProcessor` function considers the partitions of the surrounding nodes, whose votes are weighted by their distance to the nodes of the segment. Therefore, calling this method can be omitted when partitionizing the first time (on the partitioning level). This is done by checking if the last phase of the algorithm is already reached (l. 4).

Secondly, all remaining segments are assigned by `greedyAssignment` in a greedy manner (l. 6). The segment with the highest cost is assigned to the processor with the highest capacity. Afterward, the neighbor segments are assigned to the best suitable processors regarding communication and calculation capacities, and so on.

Algorithm 5 partIndirectlyAffectedNodes

```
1  Function Partitions partIndirectlyAffectedNodes
        (level, partitions, splitIndices,
        assigningUpperLevels)
2    for each segment in splitIndices do
3      results[segment] := voteForProcessor (segment,
            level, assigningUpperLevels)
4    sort(results)
5    for each segment in splitIndices do
6      proc := getBestProcessor(results[segment])
7      if isEligible(proc) then
8        procNodes  := getNodesInSegment(level, segment)
9        partitions := union (partitions, procNodes)
10   return partitions
```

If a node on the partitioning level is assigned to a certain partition, all of its sub-nodes are assigned to the same partition. After the partitioning of the levels below the partitioning level, the upper levels need to be assigned to the partitions created so far (see alg. 6).

Since the algorithm aims at assigning subtrees as big as possible to one partition, the partition for a node will be selected based on the partitions its child nodes have been assigned to (l. 6). Problems arise if the algorithm has to find a suitable partition for nodes that do not have children, i.e., for all leaves that are closer to the root node than the partitioning level. If the algorithm encounters this kind of nodes on a level, it calls the `partitionizeLevel` method again, in order to resolve the problem (l. 10).

The `partitionizeLevel` method now calls `part-IndirectlyAffectedNodes` (see listing 5), which distinguishes between two possible cases (see fig. 1 (I2)). On one hand, there could be very few leaf nodes surrounded by larger branches of already partitioned nodes (as depicted in situation a). Hence, the leaf nodes should be assigned to the processor that hosts the surrounding nodes, so that additional communication is avoided. On the other hand, it is also possible that the leaf nodes belong to a larger subtree (situation b). In this case, it is advantageous to assign the leaf nodes to another processor whose current load is too small. The `assigningUpperLevels` (see alg. 4, line 4) flag indicates the current phase of the algorithm, so that an advanced voting scheme, which is able to distinguish between both situations, can be activated. It is implemented in the `voteForProcessor` function. Having the leaves on the actual level assigned to partitions, `assignUpperLevels` can proceed toward the root node.

Algorithm 6 assignUpperLevels

```
1   Function Partitions assignUpperLevels (model,
        resources, levelIndex,
2     partitions)
3     levels := getLevels(model)
4     for l := levelIndex-1 downto 0 do
5       //assign nodes with children
6       partitions := union (partitions,childrenVote
            (levels[l+1]))
7       //assign nodes without children
8       udNodes := undecided (partitions, levels[l])
9       if size(udNodes) > 0 then
10        partitions := partitionizeLevel (model,
              resources, udNodes, partitions, true)
11    end for
12    return partitions
```

The algorithm described here is able to partition a given hierarchical model. But, as we already mentioned, we additionally need to consider constraints.

4.3 Considering Constraints

In contrast to the graph theoretical meaning of a constraint, we call the need to place a certain model on a particular host a constraint for partitioning, because the freedom to assign a model part to any partition is significantly reduced. Especially, if the constraint of a model has influence on its predecessors or successors (thus making them constrained as well). For being able to handle those constraints, we extended the algorithm described in the previous section. The `partition` method (see alg. 1) is the same as for the basic algorithm described above.

To preserve the idea of assigning all sub-nodes according to the partitioning results on the partitioning level, the top-down phase has now to ensure that these assignments are actually possible. Such an assignment is not possible, if two or more nodes in a subtree need to be assigned to two different partitions. In this case, there is a *conflict* that can only be resolved by choosing a lower partitioning level, therefore it needs to be checked by the `criteriaFullfilled` function (see alg. 7, line 2).

If nodes in the subtree are only constrained to one partition, their parent node on the partitioning level will be marked as being constrained to the same partition when entering the next phase of the algorithm.

Algorithm 7 criteriaFulfilled (with Constraints)

```
1   Function Boolean criteriaFullfilled (levels,
        levelIndex)
2     if conflictsExist(levels, levelIndex) then
3       return false
4     if size(levels[levelIndex]) > ParamNodeNum *
          numOfPartitions then
5       return true
6     if (levelIndex = size(levels)-1) or
          (size(levels[levelIndex+1]) <
          size(levels[levelIndex])) then
7       return true
8     return false
```

In the `partionizeLevel` function (see alg. 8), the constrained nodes have to be considered in a special way. At first, `partConstrNodes` (see alg. 9) is called to fill all partitions to which nodes on the partitioning level are constrained (l. 3). This is useful, because each processor should not only get its constrained nodes, but possibly enough other nodes on the partitioning level to suit its capacity. However, only nodes within a certain maximal distance to a node that already belongs to the actual partition are considered, and the overall cost of all nodes must not exceed the processor's share of calculation cost on the partitioning level.

Additionally, constraints regarding nodes in the upper levels should be taken into account. This is done by `part-IndirectlyAffectedNodes` (see alg. 5), which will now be called regardless of the phase the algorithm is in. The function itself is the same as in the basic algorithm, but `voteForProcessor` (l. 3) needs to consider the node constraints in the upper levels. This is necessary to minimize the communication cost, because the partition of a segment should be chosen – if possible – in a way that

Algorithm 8 partizionizeLevel (with Constraints)

```
Function Partitions partizionizeLevel (model,
    resources, level, partitions,
    assigningUpperLevels)
  if not assigningUpperLevels then
    partitions := partConstrNodes(level, resources)
  maxPartBlocks := min (size(resources), size(level))
  splitIndices  := getSplitIndicesByDistances (level,
      maxPartBlocks - 1)
  partitions    := partIndirectlyAffectedNodes(level,
      partitions, splitIndices,
      assigningUpperLevels)
  partitions    := greedyAssignment (model,
      resources, level, partitions, splitIndices)
  return partitions
```

nodes directly above the segment are constrained to the same partition.

Algorithm 9 partConstrNodes (with Constraints)

```
Function Partitions partConstrNodes (level,
    resources)
  processors := getProcessorsWithConstr (resources,
      level)
  for each proc in processor do
    maxCost := getCalcShare(proc)
    procNodes := getConstrNodesForProc (proc, level)
    actualCost := getCost(procNodes)
    while actualCost < maxCost do
      v := getAdditionalNode(level)
      if v != null and actualCost + getCost(v) <
          maxCost then
        procNodes := union (procNodes, v)
        actualCost := getCost(procNodes)
      else
        break
      end if
    end while
    partitions := union (partitions, procNodes)
  end for
  return partitions
```

After finishing these first two partitioning worksteps, the greedy assignment from the basic algorithm is called for the rest of the segments. The assignment of the upper levels is similar to the procedure of the basic algorithm, with one exception: Again, the calculation mechanism to determine the partition of a (parent) node (`childrenAndConstraintsVote`, see alg. 10, line 4) needs to take constraints for upper-level nodes into account.

5 EVALUATION

So far, only one infrastructure analyzer and one model analyzer have been implemented for the partitioning layer. Three partitioning algorithms, i.e., the KL algorithm, the geometric approach and the algorithm presented here, can be plugged into the system on demand. Those have been evaluated regarding the cut size and imbalance of the generated partition results. Figure 1(I3) shows a sample partition result created by the described partitioning algorithm.

Algorithm 10 assignUpperLevels (with Constraints)

```
Function Partitions assignUpperLevels (model,
    resources, levelIndex, partitions)
  levels := getLevels(model)
  for l := levelIndex-1 downto 0 do
    partitions := union (partitions,
        childrenAndConstraintsVote(levels[l+1]))
    udNodes := undecided (partitions, levels[l])
    if size(udNodes) > 0 then
      partitions := partitionizeLevel (model,
          resources, udNodes, partitions, true)
  end for
  return partitions
```

Since the new partitioning algorithm focuses on the minimization of communication cost, an important performance measurement is the number of edges between the partitions, i.e., the cut size of a partitioning result. On the other hand, the computational load has to be distributed equally over all processors.

The performance of the new algorithm was tested by creating 8-way partitions (i.e., partitions for 8 processors) for 200 randomly generated and labeled model trees which consisted of 20 to 500 nodes. Each tree had a branch factor of 4, which means that the average coupled model had 4 sub-models. This test was repeated three times for the new partitioning algorithm, under varying circumstances. For the first test, no random constraints had been declared, whereas the second and the third test added 4 and 8 constraints to the model trees, respectively.

In addition, the KL algorithm and the geometric approach have been tested under the same conditions. The results regarding processor imbalance and communication cost are shown in figure 1 (plots E1 and E2).

As can be seen in plot $E2$, the average load imbalance of partitions created by the new algorithm is rather high with $\approx 40\%$, i.e., the average difference between optimal and actual load was 40% of the optimal load. The imbalance is even worse for smaller graphs, because the partitioning algorithm makes not necessarily use of all existing processors. In contrast, both the KL and the geometric approach generate partitions which are balanced very well.

However, plot $E1$ illustrates that the communication costs introduced by the algorithm are rather small. Since at least 7 edges are needed to connect 8 processors, the results show that the unconstrained model trees are partitioned very efficiently regarding communication cost. As constraints force the algorithm to partition the tree in a differing way, introducing several random constraints enforces a slightly larger cut. In this case, both general graph partitioning approaches perform rather bad, because they do not exploit the model graph's tree property.

Additionally, we compared the new partitioning algorithm to the METIS (Karypis and Kumar 1995) package. METIS has the focus on balancing the load. As can be seen in plot $E4$, the load imbalance produced by METIS is rather low. But due to the fact that perfect load balance and minimal cut size are most often contradicting, METIS does not have the same low cut size as our new hierarchical partitioning algorithm. This is shown in plot $E3$.

The results show how the performances of the new partitioning algorithm, regarding imbalance and cut size, differ. Actually, imbalance and cut size cannot be optimized simultaneously in many cases: For example, consider a parent node with three child nodes, two of them being assigned to partition A and one to partition B. Assigning the parent to partition A minimizes communication cost (only one edge between the parent node and the child node in B), but may increase load imbalance (three nodes in partition A, only one in partition B). On the other hand, assigning the parent to partition B optimizes load balance, but increases communication cost. Because of the amount of messages which need to be exchanged for processing DEVS models, the new partitioning algorithm was designed to minimize communication cost in these cases.

6 SUMMARY

In this paper we described a flexible and extensible partitioning layer for JAMES II. The available infrastructure and the model structure are analyzed, and based on this information an algorithm will partition the model. Different algorithms for infrastructure analysis and model structure analysis can be plugged into the partitioning layer. However, the focus has been on realizing different partitioning algorithms that can be exchanged on demand. Three partitioning algorithms are currently supported, among which a newly developed partitioning algorithm holds a lot of promise, at least for hierarchically structured and processed models - the dominant type of model in JAMES II. Unlike other partitioning algorithms that have been designed for hierarchical structured models and an hierarchical execution of simulation, the algorithm takes constraints, e.g., certain models have to be executed on certain processors, into account. Thereby, the algorithm aims at reducing the negative effects introduced by the constraints on partitioning. The overall goal of the algorithm is to decrease the communication costs. Therefore, model branches are kept as complete as possible. The pay-off is a possible imbalance of working loads as has also been illustrated in our experiments with synthetic models.

First experiments with real models, which varied with respect to the number and heterogeneity of behavior and interaction pattern of sub-models, indicated that this imbalance should have little effect on the overall performance in many cases, and that the performance of the developed algorithm should surpass those that do not take the hierarchical structure of models and processing into account. However, this will be subject of further testing and experiments.

Although the partitioning algorithm is able to process larger models, i.e., several 10 thousands nodes, the efficiency of the implementation can still be improved. For the partitioning layer, further algorithms shall be implemented. This refers to partitioning algorithms, infrastructure analyzers, and model analyzers.

ACKNOWLEDGMENTS

This research is supported by the DFG (German Research Foundation).

REFERENCES

Bailey, M. L., J. V. J. Briner, and R. D. Chamberlain. 1994. Parallel logic simulation of vlsi systems. *ACM Comput. Surv.* 26 (3): 255–294.

Boukerche, A., and C. Tropper. 1994. A static partitioning and mapping algorithm for conservative parallel simulations. In *PADS '94: Proc. of the 8th workshop on Parallel and distributed simulation*, 164–172. New York, NY, USA: ACM Press.

Cuthill, E., and J. McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings Of The 1969 24th National Conference*, 157–172. New York, NY, USA: ACM Press.

Edmonds, B. 1999. *The evolution of complexity*, Chapter What is Complexity? - The philosophy of complexity per se with application to some examples in evolution. Dordrecht: Kluwer.

Fjällström, P.-O. 1998. Algorithms for graph partitioning: A survey. In *Linkoping Electronic Atricles in Computer and Information Science, 3.*

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1994. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, USA.

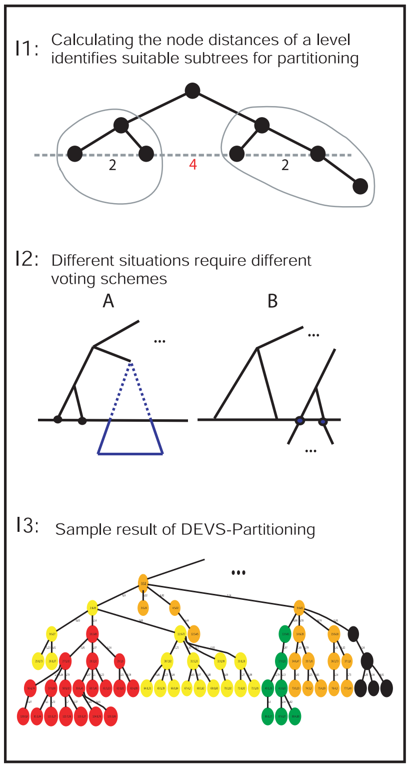
Himmelspach, J., and A. M. Uhrmacher. 2004a. A component-based simulation layer for JAMES. In *Proc. of the 18th Workshop on Parallel and Distributed Simulation (PADS), May 16-19, 2004, Kufstein, Austria*, 115–122.

Himmelspach, J., and A. M. Uhrmacher. 2004b, October. Processing dynamic PDEVS models. In *Proc. of the 12th IEEE Int'l Symposium on MASCOTS*, ed. D. DeGroot and P. Harrison, 329–336. Volendam, The Netherlands: IEEE Computer Society.

Karypis, G., and V. Kumar. 1995. *Metis: Unstrctured graph partitioning and sparse matrix ordering system, version 2.0.*

Kernighan, B. W., and S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. Journal* 49.

Kim, K. H., Y. R. Seong, T. G. Kim, and K. H. Park. 1995. Distributed optimistic simulation of hierarchical devs


I1: Calculating the node distances of a level identifies suitable subtrees for partitioning
2
4
2
I2: Different situations require different voting schemes
A
B
I3: Sample result of DEVS-Partitioning

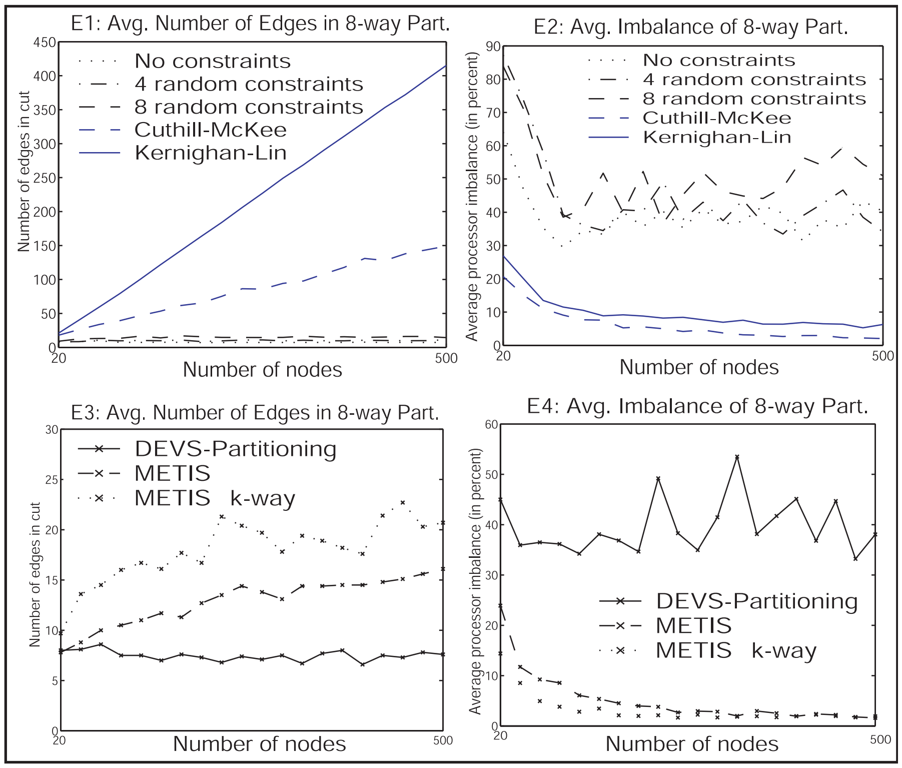


Figure 1: Illustration and Evaluation of the Presented Partitioning Algorithm

models. In *Summer Computer Simulation Conference 95*, 32–37. Ottawa, Canada.

Lemeire, J., B. Smets, P. Cara, and E. Dirkx. 2004. Exploiting symmetry for partitioning models in parallel discrete event simulation. In *PADS '04: Proc. of the 18th workshop on Parallel and distributed simulation*, 189–194. New York, NY, USA: ACM Press.

Li, L., H. Huang, and C. Tropper. 2003. Dvs: An object-oriented framework for distributed verilog simulation. In *17th Workshop on Parallel and Distributed Simulation*, 173–180. San Diego: IEEE Computer Society Press.

Nicol, D. M. 1998. Scalability, locality, partitioning and synchronization pdes. In *Proc. of the 12th Workshop on Parallel and distributed simulation*, 5–11: IEEE Computer Society.

Park, S., and B. P. Zeigler. 2003. Distributing simulation work based on component activity: A new approach to partitioning hierarchical DEVS models. In *1st International Workshop on Challenges of Large Applications in Distributed Environments*, 124.

Schloegel, K., G. Karypis, and V. Kumar. 2000. Graph partitioning for high performance scientific simulations.

Uhrmacher, A., M. Röhl, and J. Himmelspach. 2003. Unpaced and paced simulation for testing agents. In *Simulation in Industry, 15th European Simulation Symposium*, 71–80. Delft: SCS-European Publishing House.

Zeigler, B., H. Praehofer, and T. Kim. 2000. *Theory of modeling and simulation*. London: Academic Press.

AUTHOR BIOGRAPHIES

ROLAND EWALD holds an MSc in Computer Science from the University of Rostock. His research interests are in parallel and distributed simulation. He is currently a research scientist at the Modeling and Simulation Group at the University of Rostock.

JAN HIMMELSPACH holds an MSc in Computer Science from the University of Koblenz. His research interests are on designing flexible and efficient simulation systems. He is currently a research scientist at the Modeling and Simulation Group at the University of Rostock.

ADELINDE M. UHRMACHER is an Associate Professor at the Department of Computer Science at the University of Rostock and head of the Modeling and Simulation Group. Her research interests are in modeling and simulation methodologies, particularly agent-oriented modeling and simulation and their applications.