Optimization of cluster on-chip architectures

Master thesis

Frank Thiele

10.07.2013

Matriculation number: 8082758

Supervisor: Prof. Dr. Jörg Keller

Table of content

1. Introduction 1
2. Intel's Single-Chip Could Computer 5
3. Optimization of the merge sort algorithm for the Intel SCC 9
3.1. Merge sorting 9
3.2. Mapping approach 13
3.3. Optimization approach 15
4. Optimization of the Intel SCC for the merge sort algorithm 23
4.1. Motivation 23
4.2. ILP model adaptations 23
4.3. Tile and memory controller addressing 29
4.4. Optimizer selection 31
4.5. Optimization 33
4.5.1. The idea 33
4.5.2. Cooling function 34
4.5.3. Neighbor selection 36
4.5.4. When to stop 37
5. Execution results and analysis 39
5.1. Optimization with 4 MC 41
5.2. Optimization with 6 and 8 MC 55
5.3. Certain model configurations 59
5.4. Mirrored configuration 69
5.5. Lookout into further optimizations 80
5.6. Critical view into annealing 80
6. Investigation of further algorithms 81
6.1. Tiled-MapReduce 81
6.1.1. The ILP model 83
6.1.2. Results and analysis 91
6.1.3. Non-mirrored annealing results 99
6.2. Mesh communication 101
6.2.1. The ILP model 102
6.2.2. Parameter tuning 111
6.2.3. Factor selection 118

6.2.4. Results and analysis 119
7. Conclusion 135
References 137
Abbreviations 139
Eigenständigkeitserklärung 141

1. Introduction

This work covers the domain of many-core computing research. The computer industry and scientists all over the world are developing multi-core systems and adapted software for them in order to reach even higher performance figures with the increasing number of cores on a single processing system. This can easily be seen when buying a new PC or laptop. At least 2 cores are the minimum this device should have in order to support a sensible performance for the demanding multimedia applications. This trend also reached the graphic card market years ago. Nvidia and ATI improve their graphic card performance nowadays rather by doubling, tripling or quadrupling the count of so called shader units than by increasing the frequency or jump prediction. This is required for most of the high end computer games and simulators to make them run at acceptable frame rates. So the achievements of many-core research have already reached the consumer market.

Today, the performance per processing unit has become less important as the speedup is mainly generated by more units than by faster ones. There are several reasons for this change of minds. The rise of the temperature caused by higher frequencies is one of the major factors. The thermal design power of the single-core CPU "Pentium 4 HT 550" (2004) is about 115 Watt. This is quite much and hard to cool - especially as energy saving optimizations of that time were much worse compared to today's ones. But this CPU offered a clock rate of 3.4 GHz. Current CPUs like the "Intel i7-2700K" operate at comparable frequencies but with 4 or more cores and tend to consume less power. The increased number of cores and internal improvements make this CPU much faster than factor 4 in several benchmarks.

The challenge for software developers of high performance applications (e.g. computer games or big data tools) therefore became even higher. It is pretty complex to implement highly efficient but still correct parallel algorithms that make full use of all cores of the processing units the system offers. Of course, compiler and library developers put a lot of effort into extensions that are easier and saver to use. However, to develop a correct and fast parallel program is and will be a challenge for all software developers. This becomes even more obvious when taking a look at a typical server setup with several multi-core CPUs. For such configurations NUMA (non-uniform memory access) awareness becomes the key factor for efficient parallel software to maintain a high performance even when scaling the system. If not taken into consideration it might happen that by adding more cores to a process its performance slows down.

This work focuses on a many-core research processer from Intel, the so called SCC (Single-Chip Cloud Computer). It offers 48 cores placed on a single die. As the definitions of many and multi-core systems tend to differ from one author to another (some say at least 50 cores, others say many more than 16, see also [9]), this work defines the Intel SCC as many-core system. Each core of the Intel SCC is connected to an on-chip mesh network and can use this to communicate with other cores in a very fast way. With respect to external memory access the Intel SCC is a NUMA system as it uses 4 memory controllers. Each memory controller is responsible for 12 cores. The performance of the memory access is a function of the distance between the core and the memory controller. So to develop high performance software for this device is quite a big challenge.

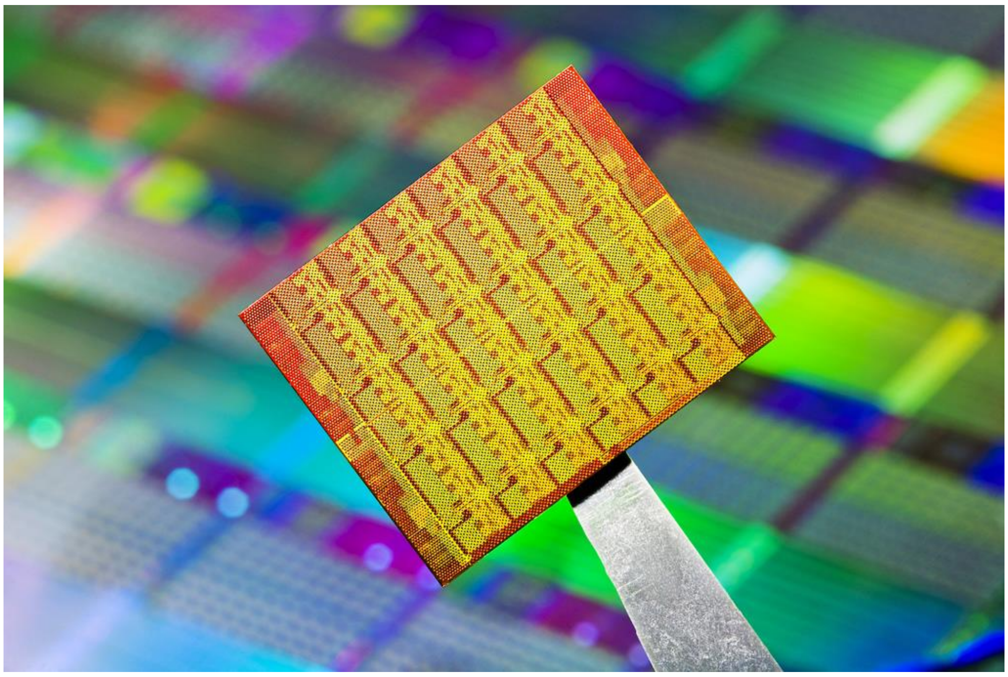


Figure 1.1 - Intel SCC die, 45nm technology, Intel.com

The purpose of this work is to investigate the influence of the memory controllers to a set of selected parallel algorithms that are executable on the Intel SCC. As the distance of the controllers to the cores is a crucial point, the placement of them on the Intel SCC die is taken a deeper look at. The original configuration is compared against a number of alternatives. This comparison is possible as for each algorithm a mathematical model with several cost factors is created. The model assumes the algorithm to consist of several tasks that run in parallel. The way those tasks are mapped to cores tends from being very inefficient to very promising. The best result based on the cost factors is the optimal one. This optimization is done by Integer Linear Programming (ILP) - a mathematical algorithm that can solve a number of inequalities that make up the model to optimality.

As already explained the best result depends on the cost factors. This work considers 3 cost factors: Memory distance for core to memory communication, communication distance for core to core communication and the computational load per core that expresses how many tasks are mapped onto a single core. As several factors are used the weight between them has a strong impact on the outcome of the optimization. For example if the weight of the computational load factor is very small, the weight of the distance costs becomes higher which typically results in a high load for some cores and no load for others. With respect to a real life system this would mean to have a processor with very fast cores but a pretty slow communication network. There it doesn’t make sense to use all cores as the communication of the intermediate results to another core is too expensive. The weight factors are varied in order to simulate the most likely conditions that might occur when running an implementation of the algorithm on a real many-core system.

The number of possible locations where the four memory controllers can be attached to is very high for a 48 core cluster. To avoid the calculation of the optimal placement of tasks for each memory controller configuration, Simulated Annealing is used in order to decrease the trial count drastically. As the optimization run for a single placement may take up to several hours depending on the complexity of the model, this approach is absolutely necessary. In addition to the location changes the number of controllers is also varied. Within this work the best memory controller locations for 4, 6, 8 and 12 memory controllers are investigated.

The result of the optimization runs is that the basic Intel SCC setup is not the overall winner which is kind of astonishing as one might think that the Intel labs are having a strong focus on performance. But as this die is only a proof of concept for future many-core CPUs and intended to be used by scientists only, the not optimal placement can be seen as a good challenge to further think about better designs for algorithms that circumvent such performance disadvantages by intelligent communication structures. A possible reason for this uncommon choice might be that the Intel designers decided to put the memory controllers to the non-optimal position with respect to performance because of more important hardware requirements like a short distance to die edges or less electrical interferences to and from other components of the die.

Another and also expected result is that the addition of more memory controllers speeds up the performance of all algorithms. This can be explained best with the average distance of a node to a memory controller. The more memory controllers are given, the less the average distance of each core to the external memory becomes.

This work is organized as following. The next chapter introduces the Intel SCC and its properties in more detail. Chapter 3 explains the first algorithm to be optimized which is merge sorting. It gives some links to papers that are the base for this thesis and demonstrates the ILP model used there. Chapter 4 is about the adaptation of this model to make it possible to vary the memory controller count and location. Also, the application of Simulated Annealing can be found there. The succeeding chapter 5 summarizes the results of the optimization runs for the merge sort model. Within this chapter the optimization is first done using the whole cluster and then for performance reasons only using a quarter of it. Chapter 6 applies all the ideas of chapter 3 to 5 to two further algorithms and compares their results to the previous ones. The first algorithm is Tiled-MapReduce and the second one is a mesh communication network.

2. Intel’s Single-Chip Could Computer

The challenge for programmers and researchers today is to experiment with many-core systems to generate solution for current and future problems faster and more efficiently than ever before. This research became more important in the last years due to the challenge of still being able to double the performance of a CPU in the typical 18 months period (referring to David House). The "doubling of performance every 18 months" phrase is based on Moore's law who predicted the doubling of the transistor numbers on an integrated circuit every two years and David House who combined this with the increasing speed of the transistors itself. In 2010 Intel announced the end of that predicted development at the end of 2013 [2]. They stated that the transistor density can then only be doubled every three years. The currently most promising way forward back to the old performance development seems to be a combination of the classical more and better transistors together with improved logic on the CPU (e.g. branching prediction) and the addition of more cores on a single die. So by implementing such CPUs and finding an effective high-performance implementation of important and commonly used algorithms one can find the way making sure that future software can still become quicker and quicker the same way we're used to since decades.

The Intel SCC ("single-chip cloud computer") is a many-core system even though its core count is not that high. Predecessors of it have already included up to 80 cores [11]. It is a so called cluster on-chip system and was published in the end of 2009 as prove of concept (POC) for researchers. It offers 48 computing cores which are placed on the die within 24 tiles. The tiles are arranged in a mesh of 4 rows and 6 columns. In this context a tile can be seen as a modern multi-core CPU with two cores. The tiles are connected using a packet based mesh network. Each core is attached to a L1 cache of size 32KiByte (16/16 for data/code) and L2 cache of 256KiByte. An interesting thing about the caching is that there is no synchronization between the caches of different cores. This needs to be implemented by software means. The advantage of this decision beside less power consumption and less hardware complexity is the enforcement of message passing as this performs much better than the communication using the external memory. The cores itself are pretty old fashioned (IA32 P54C, published in 1993) and therefore a significant speedup compared to modern CPUs can only be reached by heavily making use of all the 48 cores. The die is manufactured in 45nm technology and can consume up to 125W of power.

The routing of data packages between the tiles is done using X-Y-routing which is implemented by routers on the die. The main memory access and node to node communication is controlled by the mesh interface unit (MIU) that sends requests to and handles requests from the router. The core to core communication itself uses an extra on-tile buffer of size 16KiByte which is called MPB (message passing buffer). The buffer is read and write accessible from all cores of the mesh.

The memory attached to the die is connected to 4 routers only. So the tiles next to the router are closest to the memory. Those tiles are located on the left and right side in each second row (seen from top to down).

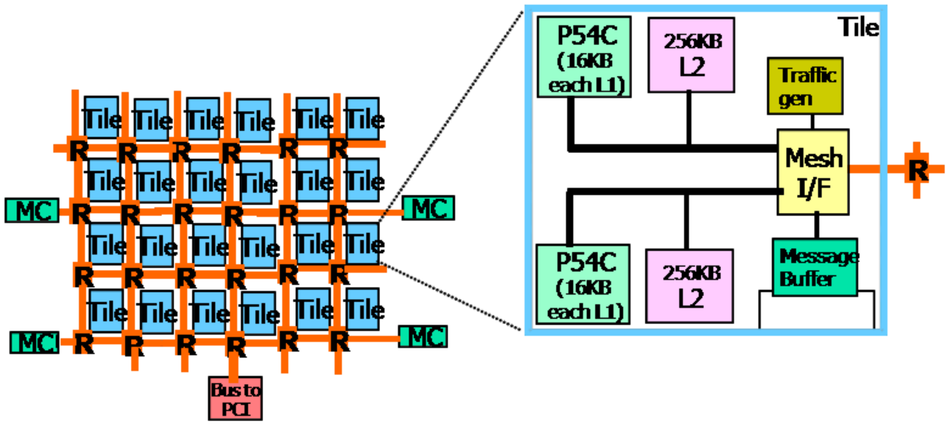


Figure 2.1 - Top level tile architecture, Intel.com

The address width of the system is 36Bit so that up to 64GiByte of RAM can be used by all cores. This address space is referred to as system address space. A core itself can only address 4GiByte within the core address space of 32Bit. The MIU is responsible for translating local core addresses to and from global system addresses.

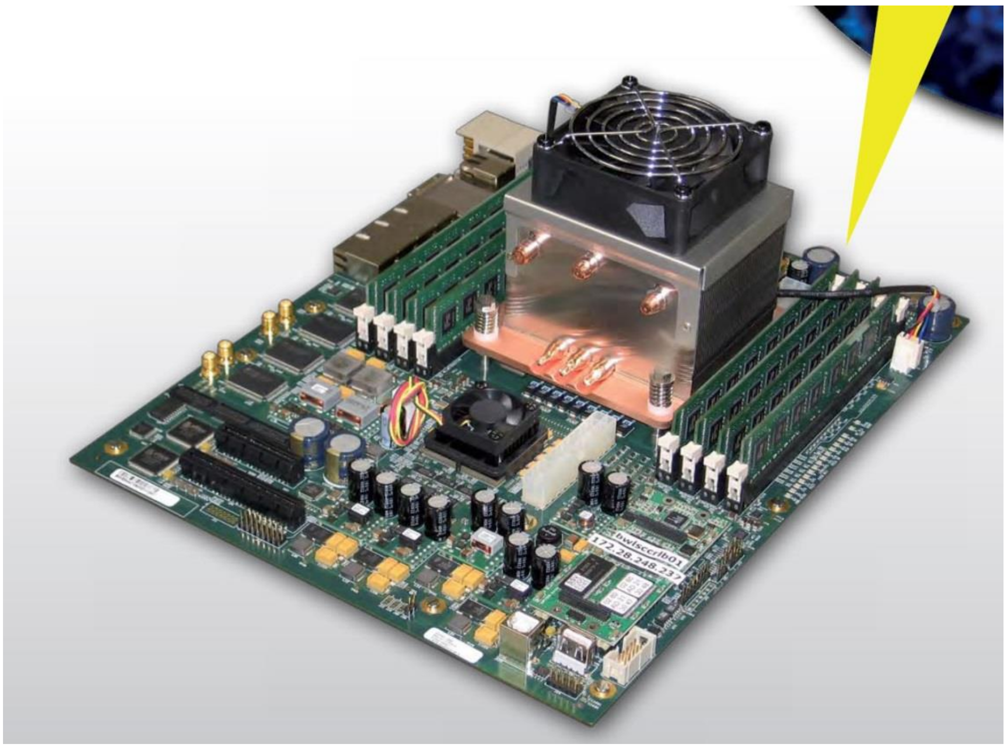


Figure 2.2 - The Intel SCC on a special main board, Intel.com

The programming and general control of the Intel SCC is done by a management console PC (MCPC). The "Rocky Lake system FPGA" connects the Intel SCC system with a normal PC by a PCIe link. The FPGA can also act as I/O hub. This enables services like disk or Ethernet access. So one can create a cluster of Intel SCC hosts or offer very fast algorithm implementations via a web service to the local organization. In the picture 2.2 the FPGA is located under the small fan, the Intel SCC die under the big fan and the PCIe and Ethernet port can be found where the external devices are typically connected to. There is also a so called "Board management controller" (BMC). This runs a Linux system on a little ARM CPU and offers services like boot strapping the FPGA. It can be connected to via Telnet or SSH (see [10]). On the Intel SCC itself a special Linux distribution is running, too.

3. Optimization of the merge sort algorithm for the Intel SCC

The next chapters introduce an Intel SCC adaptation of the merge sort algorithm (see [4]) found by a group of researchers from the FernUniversität Hagen (Germany) in cooperation with the Linköping University (Sweden).
The first sub chapter briefly summarizes the algorithm and the succeeding ones describe the required changes and retrieval of an optimized mapping of the algorithm onto the Intel SCC cores.

3.1. Merge sorting

The goal of merge sorting is to sort a number of sortable objects, e.g. ordinary integers. In the case of binary merge sorting this is done by dividing the original block of data into sub blocks of half the size (binary). If the new blocks have the size 2, the sorting is trivial and can be done directly. Otherwise, the new blocks are again divided. Two sorted blocks are then merged together based on the sorting order into one block of their aggregated size.

This approach leads to a merge tree - in case of binary merge sorting to a binary merge tree that is illustrated with the following graph. A merge task represents a node of the tree. In the figure, the merge tasks are not painted. Only the blocks to be sorted are displayed.

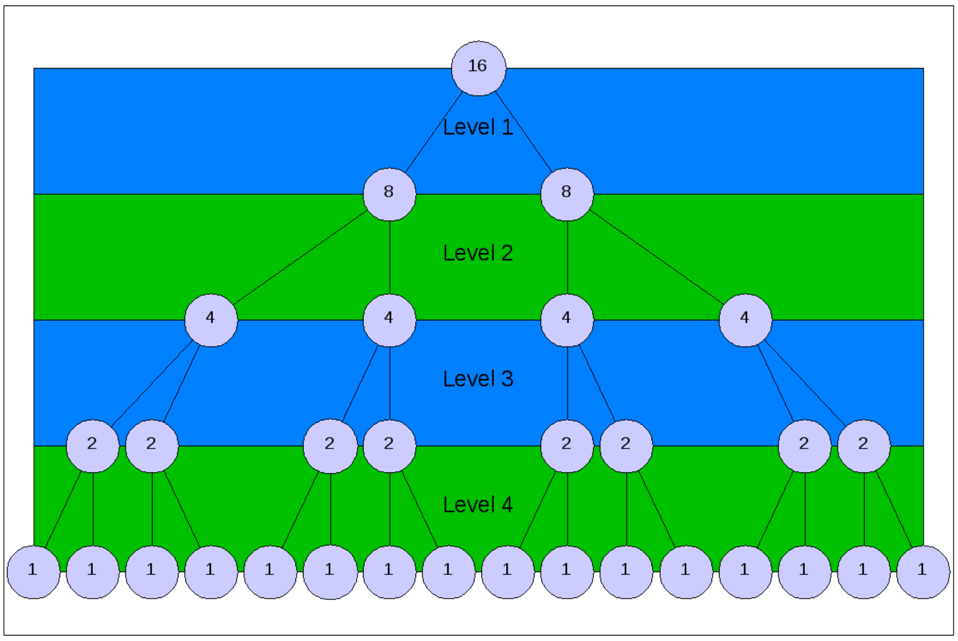


Figure 3.1A - Binary merge tree with 4 levels

In a classical non-parallel way the algorithm is implemented recursively. Using Perl, the code could look like shown in the following code block:

```
sub mergeSort {
   my @numbers = @_;

   my $count = scalar @numbers;

   if ($count > 2) {
      my $half = int($count/2);

      my @leftNumbers = @numbers[0..$half-1];
      my @rightNumbers = @numbers[$half..$count-1];

      my $sortedLeftNumbers = mergeSort(@leftNumbers);
      my $sortedRightNumbers = mergeSort(@rightNumbers);

      my @sortedNumbers = ();
      for my $value (@sortedLeftNumbers) {
         while (defined $sortedRightNumbers[0] && $value > $sortedRightNumbers[0]) {
            my $rightValue = shift @sortedRightNumbers;
            push @sortedNumbers, $rightValue;
         }
         push @sortedNumbers, $value;
      }
      push @sortedNumbers, @sortedRightNumbers;

      return @sortedNumbers;
   }
   elsif ($count == 2) {
      if ($numbers[0] > $numbers[1]) {
         return (
            $numbers[1],
            $numbers[0],
         );
      }
      else {
         return @numbers;
      }
   }

   return @numbers;
}
```

Figure 3.1B - Perl example of sequential merge sorting

For the classical parallel execution the merge tree becomes important because the merge sort algorithm is then split into phases based on the levels of the tree. Within a phase one level of the merge tree is processed.

In case of a 4 level binary merge tree (so 16 elements to be sorted), the fourth level is the first one to be processed. All 16 blocks of size one are sorted into 8 blocks of size 2. This is done in parallel by 8 workers. The next phase can be supported by 4 workers that process in parallel the 8 sorted two-element blocks into 4 sorted blocks of 4 elements each. This is done by merging the two sets together. The next level is level 2. Here, the 4 blocks are merged into 2 blocks by 2 workers. And the last phase handles level 1. Here, only a single worker merges the 2 blocks of size 8 into a single sorted result block of size 16.

There exist some approaches to further use all cores even though the number of blocks within a phase is smaller than the worker count divided by 2 (for a binary tree).

In order to reduce the number of levels and the overhead in the first merges (little number of objects to be merged), the merge sort algorithm can be extended to sort the highest levels in the first phases using another sorting algorithm. This becomes important later when taking a look at the Intel SCC adaptations. An interesting point of the "complete" merge tree is the fact that even for small object numbers there are normally less workers than leafs so that the nodes of the leaf level are not processed in parallel. The same applies for the higher levels where more destination blocks than workers are present.

If one assumes p workers, then the normal parallel merge sort tree that is processed level-wise has p leafs so that all nodes of the leaf level can be processed in parallel. For each leaf there are 2 blocks of sortable objects in the binary case. This makes 2*p blocks of unsorted objects. Those blocks need to be sorted before starting to merge them in parallel. In the best case each block fits into the workers local memory (e.g. cache). If this requirement is fulfilled, a worker can sequentially sort (e.g. quick sort) a block without accessing the external memory (excepting the initial read from and final store to memory). After all 2*p blocks are sorted in parallel (in two phases) the parallel merge sort can start at the leaf level where all p workers merge 2*p sorted blocks into p sorted blocks.

If the 2*p blocks are too large for the local memory of each processing unit, then one can normally not avoid accessing the external memory. How this can be achieved anyway is described in the following. The merge tree is further extended until for each leaf the size of its input blocks is less or equal than the local processing memory. Then the merge tree has m leafs and m is greater than p so that there are not enough workers to process the new leaf level in parallel. The way forward leads to the introduction of sub merge trees starting at the highest level with less or equal than p nodes (the "regular leaf level"). The leaf count of each sub merge tree must be less or equal then p (otherwise, the sub merge tree must be extended/split, too). Then every sub merge tree can be sorted by p workers in parallel as described above. The leaf nodes of the sub merge trees are sorted locally in parallel. Once all sub merge trees are processed, the regular leaf level can be processed in parallel.

3.2. Mapping approach

In [3] one can find in detail how the merge sort algorithm was efficiently adapted and mapped to the Intel SCC. The following gives an overview of the ideas and the solution found.
Implementing the parallel execution explained in chapter 3.1 is not very promising for one reason - the memory access performance. The memory bandwidth of the Intel SCC is limited, especially the write performance is very poor (less than 11MiByte per second and per core, see [5]). As the phase-wise execution requires the data to be fully read and fully written back into a new memory area, this approach is not feasible.
The idea was to use pipelining which has the advantage that it can make full use of the much faster core to core communication. Pipelining describes the processing of data by a number of stream processing units in series. Stream processing means that an input data stream can be separated into logical blocks and each block can be processed by the processor independent of the next data block. The processing result can then be streamed in blocks to the next processing unit which can directly consume the data in parallel to its input units generating the next data blocks. This is referred to as pipelining. The following graphic shall demonstrate this from a high-level perspective.

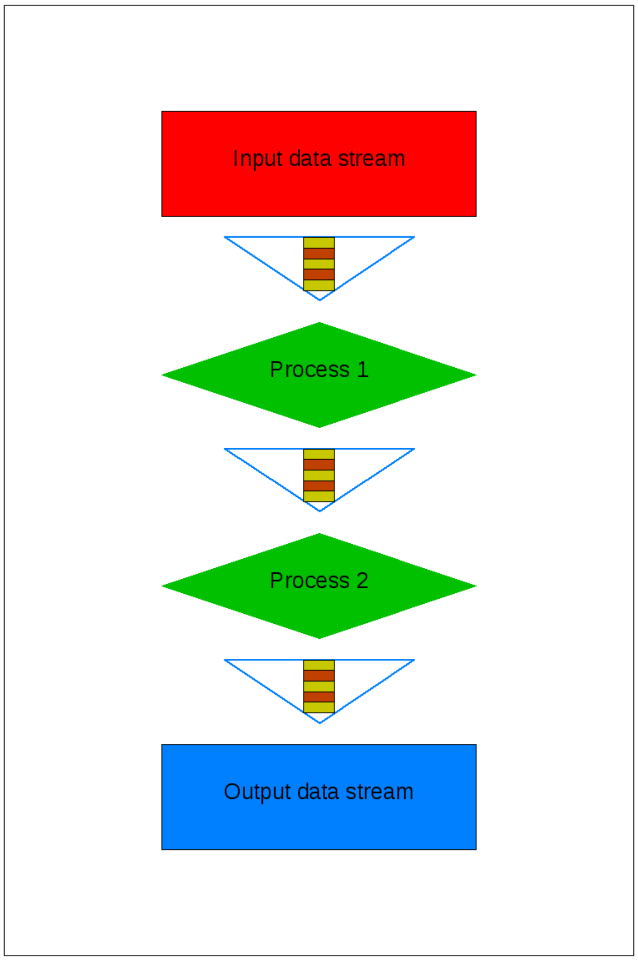


Figure 3.2 - A simple pipeline with 2 processes

Process 1 reads the data stream block wise and outputs the resulting stream in blocks to process 2 that further processes the results and generates an output data stream, too.

This processing schema has the advantage that no external memory is required between the processing units. All data is holed in caches and registers. On the Intel SCC this means that the mesh network can be used to transfer the stream data from node to node directly. The main memory is so only accessed once when reading the unsorted data from and once again when writing the sorted data back to memory. This puts the memory bottleneck only to the border of the pipeline making the overall pipeline much faster as the waiting time for the memory operations is not aggregated any more from node to node.

The pipeline processes were then defined to be the merge tasks of the merge tree. In level 4 of figure 3.1A there are 8 merge tasks. The result of those can be streamed to the 4 merge tasks of level 3 and so on.

Depending on the level the merge task is defined for, it has more or less work to do. For each level, the sum of the work load is equal. Assuming now the idea to map the tasks equally (with respect to load) to the cores of the Intel SCC, one would require a 48 level merge tree where all tasks of a level are mapped to a single core. This makes up a total of 281e12 tasks. As each task requires some management data this way is not feasible.

The way forward was then to cut the Intel SCC into 4 parts each being attached to one of the four memory controllers. So, each part has 2 rows and 3 columns of tiles, so 6 tiles or 12 cores in total. This can be used to map two independent 6 level trees with 63 tasks each to the tiles/cores of the divided cluster. The merge sort of the resulting 12 merge trees so results in 12 sorted sets of data. This can then be sorted separately using a non-pipelined merge sort.

But what level shall be put to which core? Does it make sense to put tasks of different levels to a single core? The answers to these questions are given in the next chapter.

As indicated in chapter 3.1 one can presort the input in order to reduce the size of the tree. As the 48 level merge tree requires too many tasks, the presort needs to be done for the pipelined merge sort, too. The paper describes quick sort as the algorithm to be used to presort the data for the highest level. This means for the leaf level that instead of sorting 2 elements 2 sorted sets of n elements are to be merged. Assuming a 6 level merge tree the number of tasks of the leaf level is 32. This means that 64 sorted sets of equal size are to be generated by quick sort.

3.3. Optimization approach

Chapter 3.2 explained that pipelining the merge sort tasks is better than simply mapping them onto the Intel SCC cluster nodes and leaving the data transfer unchanged. How pipelining can be done as good as possible and what the optimal mapping of each task to a node is will be explained in this chapter.

The basic idea is to design a mathematical model describing the influence of different cost factors on the overall performance. This model is then optimized and the best solution found which represents the Pareto optimal mapping of tasks to nodes can then be implemented. By using a mathematical optimization program called integer linear programming one can use a number of linear equalities and inequalities to describe and solve the model. There are a number of implementations (or solvers) for such models available, e.g. CPLEX and Gurobi. These solvers also support floating point variables and partly non-linear inequalities. Both have their own modeling language but there exists a framework called AMPL that is capable of mapping a common language into both models so one can easily compare different solvers with each other. The work [3] showed that for the required model Gurobi is much faster than CPLEX.

Integer linear programming (ILP) tries to minimize or maximize an object function. The object function depends on variables that are bounded by linear constraints that are typically formulated using inequalities. So by finding constraints for the cost factors having an influence on the performance, one can define an object function summing up those costs. The goal is to find the minimum or maximum value of the object function called the objective. In our example this means to be able to see the perfect mapping of tasks onto the cores for which the total costs are minimized.

The following cost functions have been found in [3]:

- Maximum computational load: This defines the highest load per node. This makes sense as the node with the highest load requires the most time to finish its tasks.
- Memory distance costs: [4] revealed that the distance of a node to the memory controller has significant impact on the read and write performance. The faster the data can be transferred, the faster the sorting can be finished. So by summing up the distances of the tasks mapped to an Intel SCC node that communicate with the main memory, one can approximate this influence.
- Node communication costs: As for the memory distance, the latency from node to node has an influence on the data throughput. This latency becomes higher with higher distances. So by summing up the distances of tasks communicating with each other (father and its sons), one can approximate the influence.

As the relative share of the 3 different cost functions depends on the implementation and hardware capabilities it is unknown. So one needs to make assumptions and formulate those using factors. This leads to the introduction of epsilon (eps) and zeta. Both have a floating point value between 0 and 1. Eps defines the relation between the maximum computational load and the distance costs. This means that the 2 distance related cost factors are grouped together. If eps is 1 then the computational load is the only relevant cost factor. If eps is 0 then only the distance is taken into account. Zeta defines the relation between memory and node to node distance costs. If zeta is 1 then the memory distance costs are the only relevant distance cost factor. Of course, the results of an optimization with two different eps and zeta values can be very different. Testing those different results shows up what eps and zeta values are more realistic for a given system. But this is not part of this work.

An important idea of the work was to define co-tasks. Co-tasks exist for the root node and for the leaf tasks of the merge tree. The co-tasks are used to model the calculation of the distance from the root and leaf tasks to the memory controllers. Due to pipelining the root and leaf tasks are the only tasks that communicate with the external memory.

The following AMPL model presents the result of the mentioned thoughts. It does not take into consideration that only one quarter shall be looked into. It assumes the full Intel SCC and is presented here as it is the base for chapter 4. It contains some fixes to the original one used in [3].

```
option show_stats 1;
option omit_zero_rows 0;
option omit_zero_cols 0;
option eexit 1; #bail out on exit
option solver gurobi_ampl;
option gurobi_options 'outlev=1 timing=1';
```

Figure 3.3.1 - AMPL model for full cluster optimization - part 1

The first code block is about setting run time options that affect the output and behavior of the solver. As Gurobi has been found to be a fast solver, it is bound here as the solver to be used when running the model.

```
param k integer;
param b integer;
set V = 1..(b**k-1)/(b-1);
set Vinner = 1..(b**(k-1)-1)/(b-1);
set Vext = 0..(2*(b**k)-(b**(k-1))-1)/(b-1);
set Vco = ((b**k-1)/(b-1)+1)..(2*(b**k) - (b**(k-1)) -1)/(b-1);
set Vleaf = (b**(k-1)-1)/(b-1)+1..(b**k-1)/(b-1);
set B = 1..b;
```

Figure 3.3.2 - AMPL model for full cluster optimization - part 2

The pipelined merge sort model has some parameters. Two of them are "k" and "b". "k" represents the levels of the tree. "b" represents the forking factor. It is 2 in all further chapters what means the tree is always binary. So each father task has two sons. Both parameters are of type integer.

The introduced parameters are then used to define so called sets. Sets are comparable to arrays and one can iterate over them. The sets defined here are used later. The syntax is related to the programming language Perl where a list of all numbers from 1 to 10 is generated with the expression "(1..10)".

The set "V" defines the set of all tasks belonging to a normal merge tree, starting with the root task 1 and ending with the last leaf task. The other "V*" sets are extending the task set V or form a subset of it or are at least related to V. "Vinner" contains all the tasks of "V" having sons. This means that leaf tasks are not part of this set. "Vext" contains all tasks of "V" plus the co-tasks of the root and leaf tasks. "Vco" contains the co-tasks of the leaf tasks. So "Vco" is a subset of "Vext" but not of "V". "Vleaf" contains only leaf tasks. "Vleaf" plus "Vinner" together form "V".

And the last set defined is "B" which is holding the set of links of an anonymous father to is sons. For the case "b=2" each father can have 2 sons. So "B" contains the (input) link 1 and 2.

```
param NRows integer;
param NCols integer;
set Rows = 1..NRows;
set Cols = 1..NCols;

param secondMIC binary;
```

Figure 3.3.3 - AMPL model for full cluster optimization - part 3

The first two parameters describe the Intel SCC cluster layout. "NRows" holds the row count and "NCols" holds the column count of the tile cluster. The single cores of each tile are not modeled. Derived from the parameters the sets "Rows" and "Cols" are defined and contain an entry for each row respectively column of the cluster.

The last parameter "secondMIC" is also about the Intel SCC configuration. This is a flag (so of type binary) expressing whether or not the rightmost Intel SCC cluster column is attached to memory controllers. This flag was introduced to experiment with other memory controller settings and is normally set to 1 (true).

```
param eps in [0,1.0];
param zeta in [0,1.0];
param work {1..(b**k-1)/(b-1)} in [0,1.0];
```

Figure 3.3.4 - AMPL model for full cluster optimization - part 4

The remaining merge sort parameters are given here. The meaning of eps and zeta has already been discussed. They are defined as floating point variables between 0 and 1.

The "work" parameter is defined as array of type floating point. It contains the work or load per task. The task definition is the same as for the set "V". This means for each normal task (no co-tasks) an entry exists. This also implies that a co-task has no load. It is just used to model the memory distance of its corresponding root or leaf task. The addressing of tasks in the work and some other later defined arrays works by numbering the tasks like in the "V" set and using those numbers as array indexes. The zero index of the arrays is normally in contrast to most programming languages 1 instead of 0.

The root entry "work[1]" typically has the load 1. The entry value of its two sons 1 and 2 is typically 0.5 as the number of objects (not blocks) received from the father is the same as the sum of the objects the sons have sorted. This means the load per tree level is equal. This is made sure by the executor program filling this parameter array.

```
var x{v in Vext, row in Rows, col in Cols} binary;
var quad{v in V, q in 1..NRows} binary;
```

Figure 3.3.5 - AMPL model for full cluster optimization - part 5

Now, variables are introduced. Those contain the equality and inequality argument values and are changed by the optimizer so that the objective function is improved.

The first variable is "x". It is a variable set and models the mapping of each task onto the cluster tiles. This means that "x" must be a three dimensional array. The first dimension carries the task index, and the other two the row and column number of the cluster. The tasks to be mapped are all tasks including co-tasks. This is why "Vext" is given. The mapping value is a flag. If it is 1 the task is mapped to the given row and column. If it is 0 it is not mapped to the given row and column of the cluster.

The "quad" variable set helps to bind the co-tasks to the same quadrant as the corresponding task.

```
var sumDistComm in interval[0.0,10000.0];
var sumDistMem  in interval[0.0,10000.0];
var maxCompLoad in interval[0.0,10000.0];

minimize obj:
  eps*maxCompLoad
  + (1-eps)*(1-zeta)*sumDistComm
  + (1-eps)*(zeta)*sumDistMem;
```

Figure 3.3.6 - AMPL model for full cluster optimization - part 6

Here, the cost variables are defined. The objective function named "obj" is stated here. It combines the cost variables with the weight factors eps and zeta. The minimum value (keyword "minimize") of the object function shall be found and represents the objective.

```
subject to MappingOnce {v in Vext}:
  sum {row in Rows, col in Cols} x[v,row,col] = 1;
```

Figure 3.3.7 - AMPL model for full cluster optimization - part 7

The constraints are called "subjects" in the AMPL modeling language. Here the first can be seen. It is about the constriction that each task must be mapped exactly to one tile. Expressed with linear equations this means that for each task (see subject definition "v in Vext") the sum of all entries (meaning all positions in the cluster expressed by row and column number) of the mapping variable set "x" is equal to 1. So there is no unmapped or multiple times mapped task.

```
subject to DefineMaxCompLoad { row in Rows, col in Cols }:
  maxCompLoad >= sum{v in V} work[v] * x[v,row,col];
```

Figure 3.3.8 - AMPL model for full cluster optimization - part 8

The variable "maxCompLoad" is set here. Therefore, for each tile (row and column, see subject definition) the mapping variable set "x" is checked whether a normal merge task is mapped to it. This "fact" (can be 0 or 1) is weighted by multiplying it with the load for that task ("work" array). The single results are summed up by "sum{v in V}". Please note that the sum is then only set as an upper bound to "maxCompLoad" (">="). But as the objective function shall be minimized, the "greater or equals to" becomes effectively an equals ("="). This is typical for ILP solvers.

```
subject to notinthisrow0even { r in 1..NRows/2 }:
    sum { c in Cols } x[0,2*r,c] <= 0;
subject to notinthisrow0odd { r in 1..NRows/2 }:
    sum { c in 2..NCols-secondMIC } x[0,2*r-1,c] <= 0;

subject to notinthisrowcoeven { v in Vco, r in 1..NRows/2 }:
    sum { c in Cols } x[v,2*r,c] <= 0;
subject to notinthisrowcoodd { v in Vco, r in 1..NRows/2 }:
    sum { c in 2..NCols-secondMIC } x[v,2*r-1,c] <= 0;
```

Figure 3.3.9 - AMPL model for full cluster optimization - part 9

Those subjects are all about co-task mapping constraints. The first two force the root co-task (index 0) to be mapped only to the first and last column of each odd row (counting starts with 1 at bottom) of the cluster. There, the memory controllers are attached to the router of the so addressed tile. This becomes a little bit more complicated by the parameter "secondRow" defining whether the last column contains a memory controller or not. If it is 0, the root task cannot be mapped to the last column. "Cannot be mapped" is expressed as "x[0, row, column] <= 0". All entries that are not touched by the sum around this expression are not bounded and can therefore contain the root task (if not further restricted by later subjects).

The other two subjects are equal to the first two subjects but here it is about the leaf co-tasks. The difference can be found in the task index and in the subject header that now also contains the leaf set "Vco" to iterate over.

In general, one can see that using the "NRows" parameter one can influence the memory controller count as for each odd row at least one memory controller exists.

```
subject to DefineQuadrant {v in V, memcrow in 1..NRows/2, memccol in 0..1} :
  quad[v,2*memcrow-1+memccol] <= sum { r in memcrow*2-1..memcrow*2,
                     c                                                      in
(memccol*floor(NCols/2)+1)..(memccol*floor(NCols/2)+NCols/2) } x[v,r,c];
subject to DefineQuadrant2 {v in V}:
  sum { memcrow in 1..NRows/2, memccol in 0..1 } quad[v,2*memcrow-1+memccol] >= 1;
```

Figure 3.3.10 - AMPL model for full cluster optimization - part 10

This code block fills the quad variable set. It aims on setting the entry "quad[v, z]" to 1 for a normal task v if the task v is mapped to the quadrant z. z is a value from 1 to "NRows" where:

- z is 1 for row 1 and 2 and for the left row part [column 1 to NCols/2 (integer division)]
- z is 2 for row 1 and 2 and for the right row part [column NCols/2+1 to NCols]
- z is 3 for row 3 and 4 and for the left row part [column 1 to NCols/2 (integer division)]
- z is 4 for row 3 and 4 and for the right row part [column NCols/2+1 to NCols]
- ...

The mathematical expression for z can be found in the subjects. If there are 4 rows, then there are 4 quadrants. 2 additional "quadrants" can be found for 6 rows and so on.

The first subject defines the upper bound and the second the lower bound. The first therefore depends on the task mapping variable set "x". If a task is not mapped to the quadrant, the entry shall be less or equal to 0 (as binary this means equal to 0). If the task is mapped to the quadrant, the entry can be less or equal to "1". And the second subject is comparable to the subject "MappingOnce". It requires the "quad" variable set to contain exactly one non-zero entry for each task of "V".

The next figure shows where the quad variable set is used.

```
subject to force1 { memcrow in 1..NRows/2, memccol in 0..1, v in Vleaf }:
    x[ v+b**(k-1), memcrow*2-1, memccol*(NCols-1) + 1 ]
      <= quad[v,memcrow*2-1+memccol];
subject to force2 { memcrow in 1..NRows/2, memccol in 0..1 }:
    x[ 0, memcrow*2-1, memccol*(NCols-1) + 1 ]
      <= quad[1,memcrow*2-1+memccol];
```

Figure 3.3.11 - AMPL model for full cluster optimization - part 11

Subject "force1" restricts the leaf co-tasks (leaf task index + number of nodes in last level) to be mapped to the same quadrant as its corresponding leaf task. The second subject applies the same rule to the root co-task.

```
var yh{v in Vext, r1 in Rows, r2 in Rows} binary;
var yv{v in Vext, c1 in Cols, c2 in Cols} binary;

subject to Horizontal {u in Vinner, i in B, r1 in Rows, r2 in Rows}:
  yh[b*(u-1)+i+1, r1, r2] >= sum{c in Cols} x[b*(u-1)+i+1, r1, c]
                           + sum{c in Cols} x[u,           r2, c] - 1;
subject to Vertical {u in Vinner, i in B, c1 in Cols, c2 in Cols}:
  yv[b*(u-1)+i+1, c1, c2] >= sum{r in Rows} x[b*(u-1)+i+1, r, c1]
                           + sum{r in Rows} x[u,           r, c2] - 1;
```

Figure 3.3.12 - AMPL model for full cluster optimization - part 12

The node to node communication distance calculation is calculated in two phases. In the first phase, for each task "child(u, i) := b*(u-1)+i+1" it is stored from which row "r1" to which row "r2" and from which column "c1" to which column "c2" it is sending data. "i" is the child link number (see set "B"). "u" is the father node that receives the data from its children. The information is stored as a flag in the variable sets "yh" (horizontal) and "yv" (vertical). The mapping of each task to a tile and the function "child(u, i)" to get the communication path are the input parameters for the derivation. The derivation works as following:

- If a son of "u" is mapped to row "r1" and "u" is mapped to row "r2", then the entry of "yv[child(u, i), r1, r2]" is greater or equal 1. This means that the son of u is sending from row r1 to row r2 data.
- If a son of "u" is mapped to row "r1" and "u" is not mapped to row "r2", then the entry of "yv[child(u, i), r1, r2]" is greater or equal 0.
- If a son of "u" is not mapped to row "r1" and "u" is mapped to row "r2", then the entry of "yv[child(u, i), r1, r2]" is greater or equal 0.
- If a son of "u" is not mapped to row "r1" and "u" is not mapped to row "r2", then the entry of yv[child(u, i), r1, r2] is greater or equal 0 (normally -1, but binary values are allowed only).

"Greater or equal" becomes "equal" due to the global minimization of the object function. The same applies for the "yh" variable set.

The node to node communication exists only for the destination tasks of the set "Vinner". Excluded are the leaf tasks as they have no children other than co-tasks that send data to them.

The second phase of the node to node communication distance calculation is to deduce the distance from the communication information variable sets. This can be seen in the figure 3.3.15.

The leaf tasks have no children. But they get their data from the leaf co-tasks and so from the memory. The root task sends its data via the root co-tasks to the memory. As already stated the co-tasks are just a helper to model the distance to memory. The co-tasks are mapped to the tile that is closest to the memory controller. If one now derives the distance from co-task to its corresponding task and sums this up, the memory distance is the result. The same approach as for node to node communication can be used here. The first phase of that is shown in the following two figures. The last phase can be found in figure 3.3.16.

```
subject to Horizontal0 {i in B, r1 in Rows, r2 in Rows}:
  yh[1, r1, r2] >= sum{c in Cols} x[1, r1, c]
                 + sum{c in Cols} x[0, r2, c] - 1;
subject to Vertical0 {i in B, c1 in Cols, c2 in Cols}:
  yv[1, c1, c2] >= sum{r in Rows} x[1, r, c1]
                 + sum{r in Rows} x[0, r, c2] - 1;
```

Figure 3.3.13 - AMPL model for full cluster optimization - part 13

The source of the memory communication in level 1 is the root. The receiver is task 0, the root co-task. So as like as in figure 3.3.12 the communication information is stored in "yh" and "yv".

```
subject to HorizontalCoTask {v in Vleaf, i in B, r1 in Rows, r2 in Rows}:
  yh[v+b**(k-1), r1, r2] >= sum{c in Cols} x[v+b**(k-1), r1, c]
                          + sum{c in Cols} x[v,          r2, c] - 1;
subject to VerticalCoTask {v in Vleaf, i in B, c1 in Cols, c2 in Cols}:
  yv[v+b**(k-1), c1, c2] >= sum{r in Rows} x[v+b**(k-1), r, c1]
                          + sum{r in Rows} x[v,          r, c2] - 1;
```

Figure 3.3.14 - AMPL model for full cluster optimization - part 14

The sources of the memory communication in the level k are the leaf co-tasks sending data to the leaf tasks in level k. So as like as in figure 3.3.12 the communication information is stored in "yh" and "yv".

```
subject to DefineSumDistComm:
  sumDistComm =
          # inner task to parant task
                sum {u in Vinner, i in B, r1 in Rows, r2 in Rows}
                   yh[b*(u-1)+i+1, r1, r2]  *  abs(r2 - r1)  *  work[b*(u-1)+i+1]
              + sum {u in Vinner, i in B, c1 in Cols, c2 in Cols}
                   yv[b*(u-1)+i+1, c1, c2]  *  abs(c2 - c1)  *  work[b*(u-1)+i+1];
```

Figure 3.3.15 - AMPL model for full cluster optimization - part 15

Having stored what task is sending data from one place to another, the information is evaluated in this subject for all tasks that use a pipelined communication. If the task is sending data from row "r1" to "r2" then the positive difference of "r1" and "r2" is weighted by multiplication with the load of the source. This weights the sent data with its distance. The overall sum represents the weighted node to node communication distance.

```
subject to defineSumDistMem:
  sumDistMem =
          # root task to co-root task
                sum {i in B, r1 in Rows, r2 in Rows}
                   yh[1, r1, r2]  *  abs(r2 - r1)  *  work[1]
              + sum {i in B, c1 in Cols, c2 in Cols}
                   yv[1, c1, c2]  *  abs(c2 - c1)  *  work[1]
          # co-leaf task to leaf task
              + sum {v in Vleaf, i in B, r1 in Rows, r2 in Rows}
                   yh[v+b**(k-1), r1, r2]  *  abs(r2 - r1)  *  work[v]
              + sum {v in Vleaf, i in B, c1 in Cols, c2 in Cols}
                   yv[v+b**(k-1), c1, c2]  *  abs(c2 - c1)  *  work[v];
```

Figure 3.3.16 - AMPL model for full cluster optimization - part 16

Similar to before, this subject is about distance calculation. But this time it is about the root and leaf tasks and their distance to the off-chip memory.

4. Optimization of the Intel SCC for the merge sort algorithm

4.1. Motivation

The Intel SCC has been designed to allow developers and scientists to test their theoretical models and ideas on hardware that is likely to be commonly used in the next years. Unfortunately, this hardware is fixed. One cannot predict how an implementation would behave if the amount of cores is doubled, what impact it would have if the mesh network performance is changed or the network itself becomes a hierarchical network with core and sub networks.

In order to investigate the influence of some of the possible parameters the following chapters explain how the merge sort AMPL model can be changed and used to determine the effect of varying those parameters.

4.2. ILP model adaptations

In contrast to the original model where the memory controller positions are fixed to 2 respectively 4 positions ("secondMIC" parameter) for 4 rows, the new model has been extended to allow the memory controllers to be placed freely. With the new model, the memory controller location and count can be configured. If required, the memory controllers can even be placed within the inner area of the board. As the preceding model was already capable of configuring different row and column numbers, there is no upgrade required.

The extensions also include some changes with respect to memory distance calculations which are required due to the relaxation of the memory controller locations.

The following figures show the AMPL model bit by bit.

```
#include "tree-scc-map-latency-mem4-v3.define"
option show_stats 1;
option omit_zero_rows 0;
option omit_zero_cols 0;
option eexit 1; // bail out on exit
option solver gurobi_ampl;
option gurobi_options GUROBI_OPTIONS; // This parameter is defined in the *.define
file
```

Figure 4.2.1 - Extended merge sort model - part 1

The initial option setup is equal to the chapter 3 model. The only difference is the C preprocessor directive "#include" that is parsed and interpreted before the model is run. It is used to automatically create dynamic code like a time limit after which the solver stops optimizing and returns the intermediate results.

```
param k integer;
param b integer;

set V = 1..(b**k-1)/(b-1);
set Vinner = 1..(b**(k-1)-1)/(b-1);
set Vext = 0..(2*(b**k)-(b**(k-1))-1)/(b-1);
set Vco = ((b**k-1)/(b-1)+1)..(2*(b**k) - (b**(k-1)) -1)/(b-1);
set Vleaf = (b**(k-1)-1)/(b-1)+1..(b**k-1)/(b-1);

set B = 1..b;

param NRows integer;
param NCols integer;
set Rows = 1..NRows;
set Cols = 1..NCols;

param eps in [0,1.0];
param zeta in [0,1];
param secondMIC binary;
param rootweight in [0,1];
param wscale in [0,b];
param work {1..(b**k-1)/(b-1)} in [0,1];
```

Figure 4.2.2 - Extended merge sort model - part 2

This is equal to chapter 3 and requires no further explanation.

```
param memoryControllerCount integer >= 1;
set memoryControllers = 1..memoryControllerCount;
param memoryControllerMapping {group in memoryControllers, r in Rows, c in Cols};
param rootGroup;
```

Figure 4.2.3 - Extended merge sort model - part 3

This code block is new and presents the implementation of the variable memory controller count and location. It is realized with the parameter set "memoryControllerMapping" that describes to what tile a memory controller (or better: router of the tile) is placed to. The parameter "memoryControllerCount" and the corresponding set "memoryControllers" have also been introduced for that. The parameter set "memoryControllerMapping" contains for each memory controller (called group here) a two dimensional array that has a "1" at exactly that tile entry where the memory controller is connected to. This must be made sure by the caller of the model defining the content of the parameters.

Another parameter defined here is the "rootGroup" parameter. This will be explained later.

```
var x{v in Vext, row in Rows, col in Cols} binary;
var yh{v in Vext, r1 in Rows, r2 in Rows} binary;
var yv{v in Vext, c1 in Cols, c2 in Cols} binary;
```

Figure 4.2.4 - Extended merge sort model - part 4

The meaning of those variable sets has already been described in chapter 3 and requires no further explanation.

```
var sumDistComm in interval[0.0,10000.0];
var sumDistMem  in interval[0.0,10000.0];
var maxCompLoad in interval[0.0,10000.0];

minimize obj:
  eps*maxCompLoad
  + (1-eps)*(1-zeta)*sumDistComm
  + (1-eps)*(zeta)*sumDistMem;
```

Figure 4.2.5 - Extended merge sort model - part 5

This is equal to chapter 3 and requires no further explanation.

```
subject to MappingOnce {v in Vext}:
  sum {row in Rows, col in Cols} x[v,row,col] = 1;

subject to DefineMaxCompLoad { row in Rows, col in Cols }:
  maxCompLoad >= sum{v in V} work[v] * x[v,row,col];
```

Figure 4.2.6 - Extended merge sort model - part 6

This is equal to chapter 3 and requires no further explanation.

```
// co-tasks are mapped to the memory controller nodes.
subject to zeroTaskIsMappedToMemoryControllerNode {col in Cols, row in Rows}:
#ifdef USE_ROOT_GROUP
    x[0, row, col] <= memoryControllerMapping[rootGroup, row, col];
#else
    x[0,     row,     col]     <=     sum     {memGroup     in     memoryControllers}
memoryControllerMapping[memGroup, row, col];
#endif
```

Figure 4.2.7 - Extended merge sort model - part 7

Here, the root co-task location restriction is modeled depending on whether the "USE_ROOT_GROUP" feature is enabled or not (via C preprocessor directive, so defined by caller). The first runs of the model revealed a significant performance improvement of the model run time by permanently setting the root co-task to a dedicated memory controller node. This fixation is now configurable and can be turned off if required. So if the "USE_ROOT_GROUP" feature is enabled, the root co-task (0) is forced to a certain memory controller tile (to be more precise: to the router of that tile). In this case, the parameter "rootGroup" is evaluated. If the feature is not enabled, this parameter is ignored and the subject is just about mapping the root co-task to exactly one of the configured memory controller tiles.

The quad variable set has been removed as it further complicates the model.

```
subject to leafCoTasksAreMappedToMemoryControllerNode {v in Vco, col in Cols, row
in Rows}:
    x[v,     row,     col]     <=     sum     {memGroup     in     memoryControllers}
memoryControllerMapping[memGroup, row, col];
```

Figure 4.2.8 - Extended merge sort model - part 8

The constraint of mapping the leaf co-tasks to the memory controller tiles is implemented here. This is implemented differently than in chapter 3 as here is no rule known any more to express the memory controller positions (before: every odd row contains 1 or 2 memory controllers). Now the "memoryControllerMapping" parameter set is summed over for each tile and controller. If for a certain tile the entry of a memory group is 1 then the leaf co-task can be mapped to that tile. Otherwise this is not allowed.

```
subject to Horizontal {u in Vinner, i in B, r1 in Rows, r2 in Rows}:
  yh[b*(u-1)+i+1, r1, r2] >= sum{c in Cols} x[b*(u-1)+i+1, r1, c]
                           + sum{c in Cols} x[u,           r2, c] - 1;
subject to Vertical {u in Vinner, i in B, c1 in Cols, c2 in Cols}:
  yv[b*(u-1)+i+1, c1, c2] >= sum{r in Rows} x[b*(u-1)+i+1, r, c1]
                           + sum{r in Rows} x[u,           r, c2] - 1;
```

Figure 4.2.9 - Extended merge sort model - part 9

This can also be found in chapter 3. Both subjects are about the first part of the node to node distance calculation.

```
subject to Horizontal0 {r1 in Rows, r2 in Rows}:
  yh[1, r1, r2] >= sum{c in Cols} x[1, r1, c]
                 + sum{c in Cols} x[0, r2, c] - 1;
subject to Vertical0 {c1 in Cols, c2 in Cols}:
  yv[1, c1, c2] >= sum{r in Rows} x[1, r, c1]
                 + sum{r in Rows} x[0, r, c2] - 1;
```

Figure 4.2.10 - Extended merge sort model - part 10

This figure and the next one are equal to chapter 3. They are about the first part of the distance to memory calculation.

```
subject to HorizontalCoTask {v in Vleaf, r1 in Rows, r2 in Rows}:
  yh[v+b**(k-1), r1, r2] >= sum{c in Cols} x[v+b**(k-1), r1, c]
                          + sum{c in Cols} x[v,          r2, c] - 1;
subject to VerticalCoTask {v in Vleaf, c1 in Cols, c2 in Cols}:
  yv[v+b**(k-1), c1, c2] >= sum{r in Rows} x[v+b**(k-1), r, c1]
                          + sum{r in Rows} x[v,          r, c2] - 1;
```

Figure 4.2.11 - Extended merge sort model - part 11

```
subject to DefineSumDistComm:
  sumDistComm =
          // inner task to parant task
                sum {u in Vinner, i in B, r1 in Rows, r2 in Rows}
                   yh[b*(u-1)+i+1, r1, r2]  *  abs(r2 - r1)  *  work[b*(u-1)+i+1]
              + sum {u in Vinner, i in B, c1 in Cols, c2 in Cols}
                   yv[b*(u-1)+i+1, c1, c2]  *  abs(c2 - c1)  *  work[b*(u-1)+i+1];
```

Figure 4.2.12 - Extended merge sort model - part 12

The final subject that calculates the sums of the node to node communication costs is equal to the one of chapter 3. No changes were required. The same applies to the last figure below where the distance to memory is finally aggregated.

```
subject to defineSumDistMem:
  sumDistMem =
          // root task to co-root task
                sum {r1 in Rows, r2 in Rows}
                   yh[1, r1, r2]  *  abs(r2 - r1)  *  work[1]
              + sum {c1 in Cols, c2 in Cols}
                   yv[1, c1, c2]  *  abs(c2 - c1)  *  work[1]
          // co-leaf task to leaf task
              + sum {v in Vleaf, r1 in Rows, r2 in Rows}
                   yh[v+b**(k-1), r1, r2]  *  abs(r2 - r1)  *  work[v]
              + sum {v in Vleaf, c1 in Cols, c2 in Cols}
                   yv[v+b**(k-1), c1, c2]  *  abs(c2 - c1)  *  work[v];
```

Figure 4.2.13 - Extended merge sort model - part 13

4.3. Tile and memory controller addressing

The location of the memory controllers can now be changed freely. In order to make clear to what tile the memory controller is attached to without using an illustration an addressing schema is required.

This can be done by giving each tile an address. Within this work the tiles are named using one of the following two addressing schemes:

- The tile in the top left vertex is named "1,1" or "0". The left tile in the second row is addressed with "2,1" or "6" assuming a column count of 6.
    - The first address contains two numbers separated by a comma:
        - The first number is the row index of the cluster (counting from top to down, starting with 1).
        - The second number is the column index of the cluster (counting from left to right, starting with 1).
    - The second address is the linear transformation of the first address type. It can be calculated as following: "(rowIndex - 1) * columnCount + columnIndex - 1".
- The left tile in the second row is addressed with "2,1" or "6" assuming a column count of 6.
- The right tile in the second row is addressed with "2,6" or "11" assuming a column count of 6.
- And so on…

Applied to the Intel SCC this means:

- There are tiles from "0" (top left vertex) to "23" (bottom right vertex).
- The tiles "6", "11", "18" and "23" have a memory controller attached to their router.

If the memory controller of the bottom right vertex "23" ("4,6") is moved to the top right position, then the new address of it would be "5" or "1,6".

4.4. Optimizer selection

The selection of the optimizer is crucial with respect to the time within the optimal solution of a model is found. At least this is the first idea one could have when trying to find a good solver for the merge sort model. But finding the optimal solution is not the only important property that should be considered when comparing different optimizers and versions. The approach used to find the best model is explained in detail later. For now, it is enough to say that the best configuration of a huge set of possible memory controller positions for the model is approximated. Inherent to this principle is not to find the best but a good local solution which is described in the next chapters. With respect to that one can define the perfect optimizer for the merge sort model to be the one with the fastest and best candidate. That means the intermediate results are good enough compared to a given value and do not necessarily need to be optimal.

As already stated in the previous chapters Gurobi is a very fast solver for the merge sort model. This is why this work will not consider other optimizers. Instead of that, only different versions of Gurobi are compared against each other. As the model is written in the description language of AMPL only the published versions of the combination of AMPL and Gurobi are tested. There are many more Gurobi standalone versions which cannot be considered here.

The following table represents the results of 6 runs for 4 different versions of AMPL with Gurobi. Each run uses a certain memory controller configuration. As each run for a given configuration behaves very much equal when repeated only one trial is used. The first 4 digits of the "setup" column define the controller positions, the last one the root memory controller index ranging from 1 to 4 (as 4 memory controllers are used).

Table 4.4.1 - Results of a comparing run between different Gurobi versions

VERSION	SETUP	VALUE	TIME	5%	3%	1%	0%
4.6.1a	0_1_6_12_1	1.1875	1800.01s	1762s	1762s	1800s	1800s
5.0.0	0_1_6_12_1	1.1875	1800.06s	73s	76s	1368s	1368s
5.0.2	0_1_6_12_1	1.2109375 (+2.0%)	1800.48s	77s	82s	82s	109s
5.1.0	0_1_6_12_1	1.203125 (+1.3%)	1800.02s	19s	19s	46s	1591s
4.6.1a	0_1_6_12_2	1.1875	1384.13s	55s	64s	1095s	1095s
5.0.0	0_1_6_12_2	1.203125 (+1.3%)	1800.03s	15s	19s	65s	92s
5.0.2	0_1_6_12_2	1.203125 (+1.3%)	1800.04s	15s	20s	66s	93s
5.1.0	0_1_6_12_2	1.1875	997.41s	42s	42s	42s	53s
4.6.1a	2_3_20_21_1	1.2421875	961.07s	60s	841s	889s	889s
5.0.0	2_3_20_21_1	1.2421875	888.92s	99s	785s	785s	839s
5.0.2	2_3_20_21_1	1.2421875	919.48s	63s	63s	851s	876s
5.1.0	2_3_20_21_1	1.25 (+0.6%)	1802.61s	77s	80s	164s	164s
4.6.1a	2_3_20_21_2	1.2421875	976.19s	53s	66s	123s	879s
5.0.0	2_3_20_21_2	1.2421875	934.65s	49s	97s	869s	884s
5.0.2	2_3_20_21_2	1.2421875	809.45s	43s	679s	765s	765s
5.1.0	2_3_20_21_2	1.2421875	982.21s	24s	850s	850s	850s
4.6.1a	6_12_11_17_1	1.2734375	961.64s	106s	800s	800s	830s
5.0.0	6_12_11_17_1	1.3359375	1800.02s	57s	57s	1546s	1546s

VERSION	SETUP	VALUE	TIME	5%	3%	1%	0%
		(+4.9%)					
5.0.2	6_12_11_17_1	1.2734375	1179.37s	705s	1059s	1113s	1113s
5.1.0	6_12_11_17_1	1.2734375	1414.19s	141s	141s	141s	157s
4.6.1a	6_12_11_17_2	1.2734375	1264.88s	38s	131s	131s	131s
5.0.0	6_12_11_17_2	1.2734375	1800.01s	676s	676s	743s	899s
5.0.2	6_12_11_17_2	1.2734375	1800.02s	674s	694s	694s	709s
5.1.0	6_12_11_17_2	1.2734375	929.40s	25s	35s	823s	905s

The intermediate values reached while running are displayed in the range columns, e.g. 38s in the 5% column means that after 38s the intermediate solution was within a 5% range of the final (but not necessarily optimal) solution of the run. Red markers show that the time limit of 30 minutes was hit. Green markers express that the value is the best of the current range. Orange markers are used to show that the range was hit faster than for the green ones but the overall result compared against is worse (see value column).

In order to compare the results against each other, the green and orange markers are summed up where green markers are weighted with the factor 1 and orange ones with 0.5. The overall winner out of that is Gurobi v5.1.0 with 10 points. The second winner with 9 points is version 5.0.0. The others have less than 7 points.

As the number of tests is pretty small and the weighting factors are kind of arbitrary the informational value can be questioned. But due to time limitations this should be acceptable.

4.5. Optimization

In this chapter it shall be explained how the Intel SCC can be optimized. A precondition for an optimization is the knowledge about what can be changed in order to achieve better results at all. Therefore, the "setup" of the model to be optimized needs to be defined. This setup includes the definition of the hardware (cluster) and software (merge sort) parameters:

- hardware specific parameters
    - memory controller count (4 for Intel SCC)
    - row count (4 for Intel SCC)
    - column count (6 for Intel SCC)
- merge sort specific parameters
    - eps (weight of computation over communication costs)
    - zeta (weight of memory controller distance over node distance costs)
    - k (number of merge tree levels)
    - work load per task

In principle one can now try to find the best result for a fixed setup by changing the hardware layout within the requirements of the setup. In this case, the memory controller positions are modifiable as only their count is defined. But running the model with all possible values takes too much time. For example for a model with 4 rows, 6 columns and 4 memory controllers there are 4 x 6 = 24 positions where a memory controller can be placed. As there are 4 memory controllers the number of possible combinations is 24! / (20!*4!) = 10626. This needs to be multiplied by 4 as for each combination the root co-task can be placed to one out of four controllers. In sum about 42k alternatives to be calculated. Assuming only very optimistic 100 seconds for each run would result in nearly 50 days of calculations.

Assuming that memory controllers can only be attached to the edges of the cluster the situation becomes more relaxed as for 16 possible locations only about 8.5 days of processing time are required. But this is still too much as the run needs to be done several times, e.g. for different eps and zeta values or to make sure that the result is really very good.

As running the model for all possible memory controller positions resulting from a given setup is unacceptable the following chapters describe one possible way to find good but not optimal solutions.

4.5.1. The idea

The basic assumption is that the optimal configuration won't get much worse just by slightly changing the memory controller positions. This also applies to configurations of the setup that are not optimal. There is always a neighboring configuration B of A so that B is either as good as A or at least pretty close to A.

This does not automatically apply to the root controller node (where the root co-task is fixed to) for odd configurations. An odd configuration means here an odd memory controller distribution. But to make the optimization as easy as possible this can be accepted.

There are a number of optimization algorithms that could be used to find a good solution by searching them randomly and testing their neighborhood for even better values. One of those is Simulated Annealing. Because of its easy application to the problem it has been chosen to find the optimal solution for a certain model setup.

Based on the idea of the controlled annealing of metal to minimize the occurrence of defects this algorithm uses a start temperature that cools down from iteration to iteration. Here, iteration refers to an optimization run that produces a specific result for the current configuration of the model setup. From iteration to iteration a neighbor of the current best configuration is selected and tested. The current best configuration is adapted if the current configuration is either equal or better than the current best configuration or if the probability p of the difference (to the current best) and the current temperature is higher than a randomly chosen value in the range of 0 and 1.

The further the annealing process progresses the lower the temperature and therefore the lower the chance to accept a worse configuration as the current best configuration becomes. The function of p is predefined as e^(-diff/T).

How to cool down, how to select a neighbor and when to stop is to be decided based on the specific model and its parameters that shall be optimized. The next chapters explain how this is done within this work.

4.5.2. Cooling function

Two options for the temperature cooling are investigated. The first is the linear temperature curve. It uses a start temperature and decreases this linearly with the current iteration number down to zero.

The second curve is based on a power function. It also uses this start temperature but decreases the temperature by dividing the start value by the current iteration number.

The following figure shall demonstrate both temperature functions inserted in the probability function e^(-diff/T). The first 3 examples in the legend use the power function, the last 3 ones use linear cooling. The difference is set to 0.1 as this corresponds to a difference in the subject of about 10 to 15% for the later introduced test. Early test runs of this work showed that the best result found was about 10 to 15% better than the result for the Intel configuration. So setting the value to 0.1 results in a probability function than can be interpreted as the likelihood that a very good configuration is replaced by a result that is just as good as the original Intel configuration.

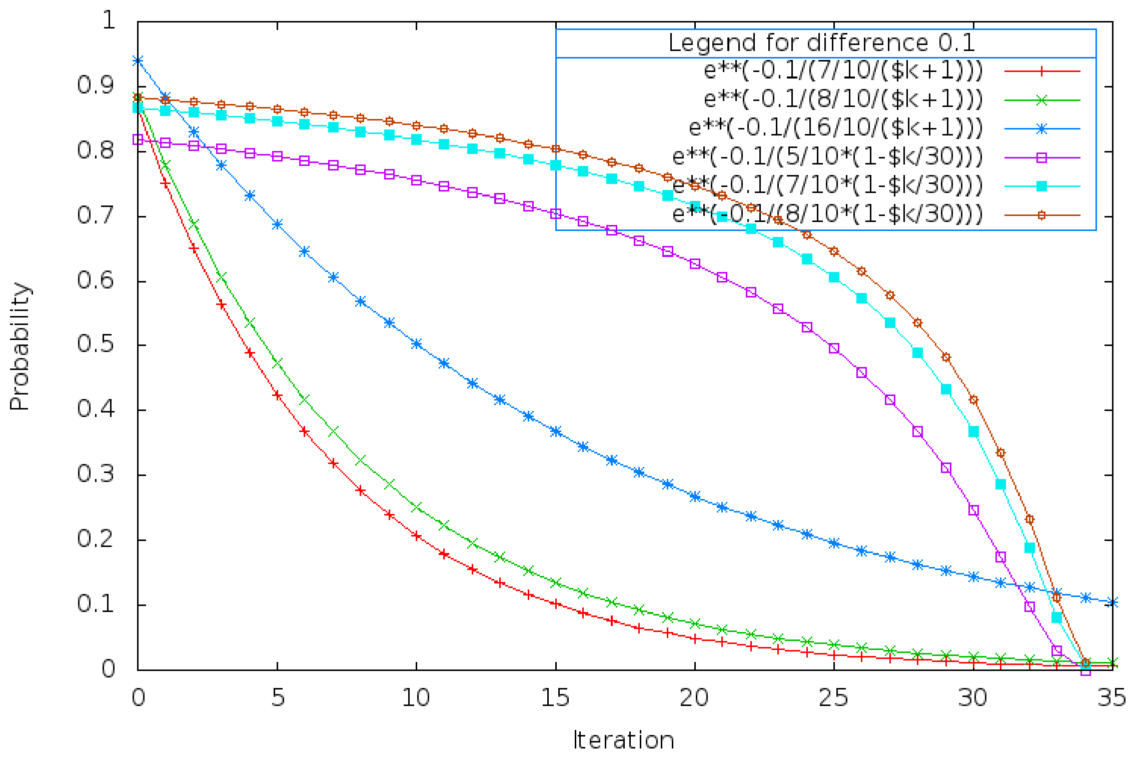


Figure 4.5.2.1 - Graph displaying the probabilities for linear and power function based temperature annealing

One can see that the different options lead to two different behaviors for the same start temperature and a given difference. The linear function leads to steady changes for a long time whereas the power function represents the opposite - worse values become much earlier unacceptable. The higher the start temperature is set the higher the iteration count becomes for the same probability.

Experiments with a small statistic show that for the Intel SCC merge sort model the power function with temperature 8 (the second in the legend of above) delivers more frequently good results. The following table summarizes those tests. In this table the header is to be read like that: "P_8" means that the power function is used with a start temperature of 8. "L_7" means that the linear function is used with a start temperature of 7.

Table 4.5.2.1 - Results of annealing runs with both linear or power function and 2 different start temperatures, 10 times each

Subject cluster	P_8	P_16	L_5	L_7
(1.12 - 1.15)	1	-	-	-
(1.15 - 1.18)	2	-	3	1
(1.18 - 1.20)	6	3	2	8
(1.20 - 1.23)	1	5	3	1
(1.23 - 1.26)	-	2	2	-

The outcome of this experiment is normally only valid for the special model setup used (Intel SCC, eps=0.5, zeta=0.5, k=7). To speed up the required calculations, the algorithm and starting temperature is used anyway for all the other possible model setups having the same number of merge tree levels and memory controllers. This might have an impact on the degree of improvement over the original model. But as long as it can be improved this approach is acceptable.

If the mentioned parameters change the start temperature and iteration count are adapted by a little rule of thumb: "Higher parameter numbers require higher temperatures and more iterations". This fuzzy rule can be used and applied to generate a plot like in the previous figure to define the values. E.g. when running models with 6 memory controllers, the temperature is doubled to 16 but the iteration count is only increased by 50% to 60. For 8 memory controllers, the temperature is set to 24 and 70 iterations are used. In both cases, the function curve looks similar to the one with a start temperature of 8.

One other side note: By extending the memory controller count or the number of merge tree levels, more and more inequalities are to be considered by Gurobi. This typically requires a higher waiting time. The waiting time is also adapted with the same rule as for the temperature and iteration count and like there it cannot be considered as a perfect adaptation.

4.5.3. Neighbor selection

The neighborhood of a configuration needs to be defined in order to select a neighbor of it. This can be done as following:

- A configuration X of a setup specifies the not already fixed model parameters. They are:
    - The memory controller indexes MC[i] where i is a number between 1 and the memory controller count n. One such index is of the range [0, rowCount*colCount-1] and represents the location of the memory controller.
    - The root index r that describes to what memory controller the root co-task is attached. So r is a number between 0 and the memory controller count n minus 1 (see also chapter 4.2).
- The neighborhood N(X) of a configuration X can then be defined as the set of configurations for which a parameter value differs by maximum one unit compared to the same parameter of X.
- For the memory controller indexes and the root index this needs to be extended by the inclusion of the wrap around case: The difference between the maximum and the minimum value of a parameter is 1.

This definition includes the configuration in its own neighborhood.

There are two different parameters classes: Memory controller and root co-task index. For each class an array can be defined that contains all parameter values of a class.

Now one can define the neighbor selection with the following Perl program:

```
    sub getNeighbors {
        my ($parameterClassConfig, $probility, @valuesBefore) = @_;

        my $max = $parameterClassConfig->{Max};
        my $min = $parameterClassConfig->{Min};
        my @newValues = ();
        my %valueIsUsed = ();

        for my $currentValue (@valuesBefore) {
            my $newValue = _changeByOneUnit($probility, $currentValue);
            $newValue = _normalizeValue($newValue, $min, $max); # wrap around

            my $shallInc = _shallDo(0.5) ? 1 : 0;
            while ($valueIsUsed{$newValue}) {
                $newValue += $shallInc ? 1 : -1;
                $newValue = _normalizeValue($newValue, $min, $max); # wrap around
            }

            push @newValues, $newValue;
            $valueIsUsed{$newValue} = 1;
        }

        return @newValues;
    }
```

Figure 4.5.3.1 - Neighbor selection algorithm

The input of that algorithm is the parameter class configuration which contains the minimum and maximum values for the parameters and a probability value and the value array that represents the current configuration. The result contains a neighbor. This can be the same as the input configuration. The algorithm makes sure that the values of the array are distinct as it doesn't make sense to map two memory controllers to a single node. This function is used to get a neighbor for the two parameter classes. In the implemented final algorithm the probability value used is 0.7.

An optimization applied here is the avoidance of calculating a result for the same or a mirrored configuration twice. If this case is detected, the probability value used to change the neighbor values is increased to 1. Also, the neighbor selection uses another base configuration. Instead of using the current best result, the last neighbor found is then used to find the next neighbor. This maximizes the likelihood to get another unique neighbor and minimizes the calculation time.

4.5.4. When to stop

The stop conditions for the annealing algorithm applied are:

- time limit for the whole optimization, e.g. 1 hour
- iteration limit, e.g. 30 iterations

The "better than X" limit has not been implemented as this could hide even better configurations.

5. Execution results and analysis

Now the best model configuration for a model setup is tried to be found. The goal is to achieve better values than for the original Intel SCC model configuration. In theory, memory controllers (MC) can be placed anywhere around the cluster. This work simulates in this chapter a realistic hardware layout by putting the memory controllers to the edges of the cluster.

The tasks of the model are mapped to the tiles of the cluster and not to the cores of a tile. This is because otherwise the model would consist of 8 rows and 6 columns and some other changes related to memory controller placement would be required. As this leads to higher complexity and solution times, the calculation is simplified by just mapping tasks to tiles. More details can be found in chapter 3.2.

The following sub chapters describe the retrieval of a cluster configuration that is better than the Intel SCC. This is done in 4 different ways.

First, the best configuration is tried to be found using the full cluster. This has the disadvantage that because of the high complexity of a 24 level merge tree only a 7 level merge tree is used which results in a very poor load distribution. But the result is anyway interesting as the best memory controller setup found is an odd one. The second chapter 5.2 extends the chapter 5.1 by adding more memory controllers to the cluster and investigates the influence on the performance.

Chapter 5.3 is about the manual definition of the memory controller positions. The reason behind this manual selection is to find an even configuration that is better than the original Intel SCC configuration.

The next chapter uses another approach in order to get a perfect cluster usage. Therefore, chapter 5.4 introduces the mirrored configuration which automatically results in even memory controller configurations. The best results of the optimization runs for the merge sort model are shown here.

5.1. Optimization with 4 MC

The following tables and figures present and summarize the output of two annealing runs. The following model setup was used: eps={0.1, 0.5, 0.9} and zeta={0.1, 0.5, 0.9}.

A1) communication costs predominate computation costs AND node distance costs predominate memory controller distance costs

Table 5.1.A1 - results of eps=0.1, zeta=0.1

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	0.7	7	0	0
Best configuration	0.7	7	0	0

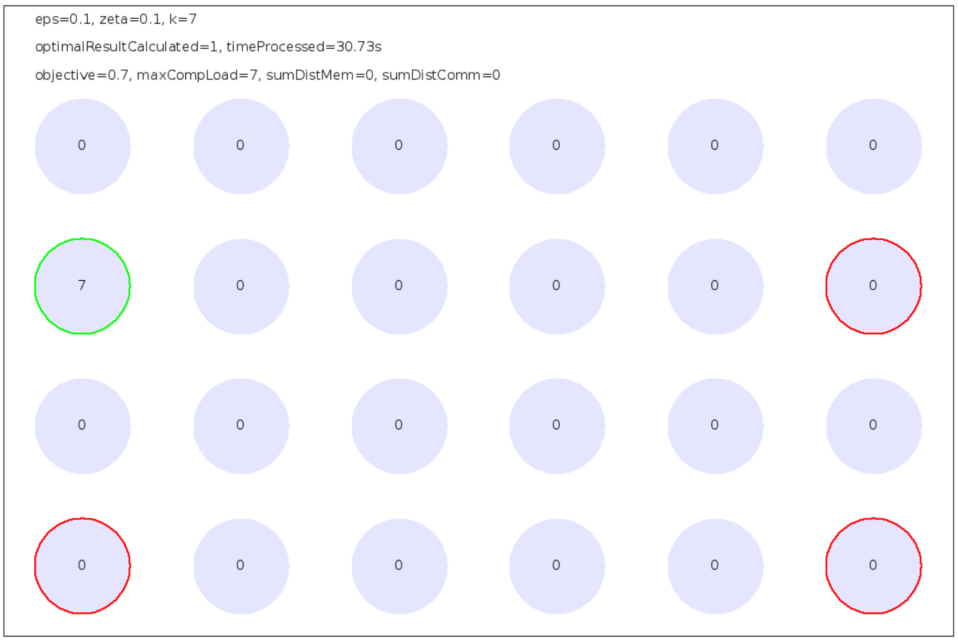

eps=0.1, zeta=0.1, k=7
optimalResultCalculated=1, timeProcessed=30.73s
objective=0.7, maxCompLoad=7, sumDistMem=0, sumDistComm=0

Figure 5.1.A11 - Intel SCC task and memory controller mapping

This drawing was automatically generated from the result dump of the Gurobi model execution. Its 4 rows and 6 columns represent the node structure of the Intel SCC. One node stands for 2 cores or one tile. The colored ones are the tiles with an attached memory controller. The green one is the root node meaning the one node where to root co-task is mapped to. As there is no communication between the nodes for this configuration, there is no such information available here. The legend at the top part summarizes the run details. Most are self-explaining. The parameter "optimalResultCalculated" tells whether the Gurobi run did fully finish the linear optimization or was stopped after a time limit (in most of such runs 2 hours).

As the computation costs are not important here, the relation between the communication and the memory controller distance becomes the key factor. If the communication costs relation is as high as here, it doesn't matter where the memory controllers are placed. All tasks are mapped to a single node to enforce the low node communication costs subject.

A2) communication costs predominate computation costs AND node distance costs are weighted equally to memory controller distance costs

Table 5.1.A2 - results of eps=0.1, zeta=0.5

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	0.7	7	0	0
Best configuration	0.625	4	0.5	0

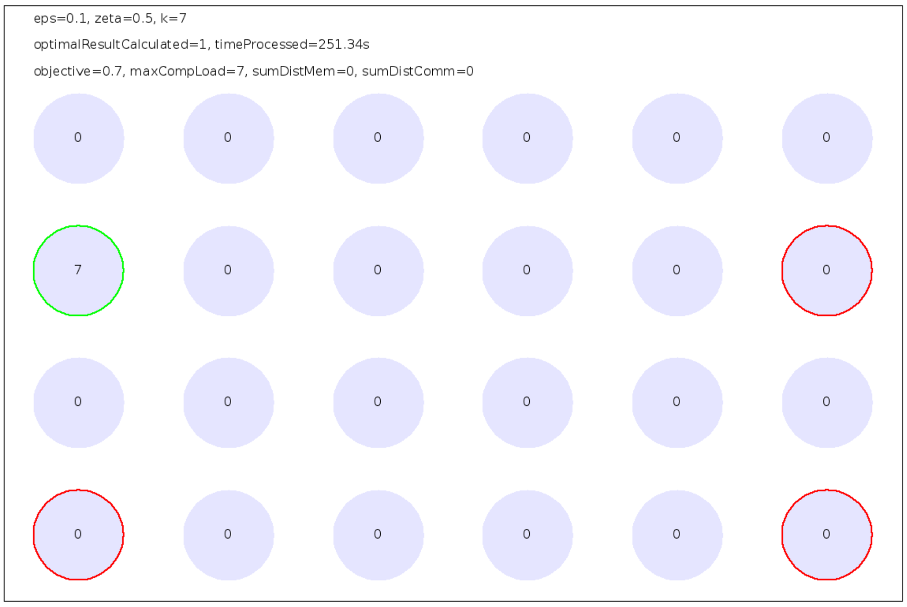

eps=0.1, zeta=0.5, k=7
optimalResultCalculated=1, timeProcessed=251.34s
objective=0.7, maxCompLoad=7, sumDistMem=0, sumDistComm=0
7
0

Figure 5.1.A21 - Intel SCC task and memory controller mapping

One can see that there is no difference to the setup with zeta=0.1.

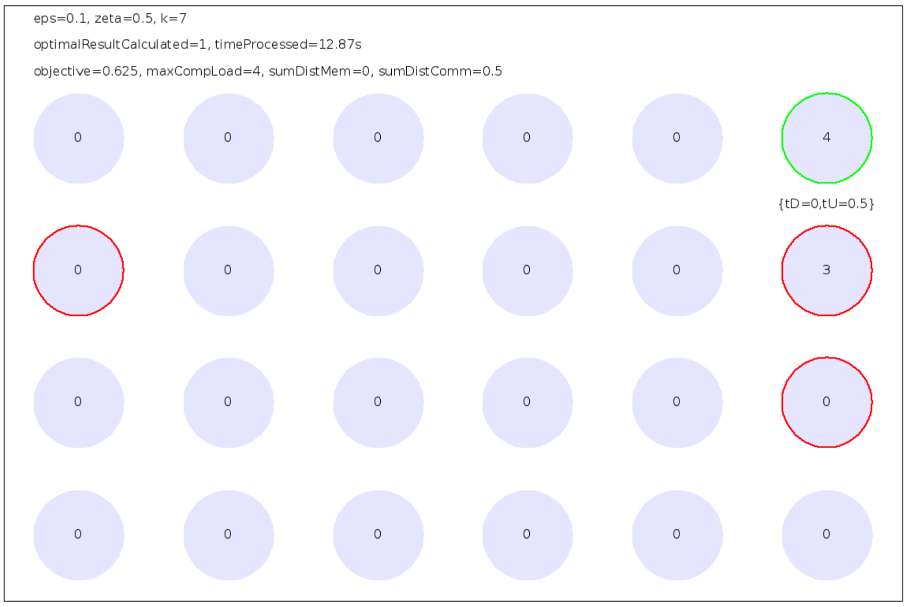

eps=0.1, zeta=0.5, k=7
optimalResultCalculated=1, timeProcessed=12.87s
objective=0.625, maxCompLoad=4, sumDistMem=0, sumDistComm=0.5
4
{tD=0,tU=0.5}
3
0

Figure 5.1.A22 – Best task and memory controller mapping

This picture demonstrates the communication part as well. "tU" means to upper nodes, "tD" means downward, "tL" to left nodes and "tR" means to send data to the right tile. The communication path illustrated in the figures also includes the memory controller communication part which is 0 here but becomes important later.

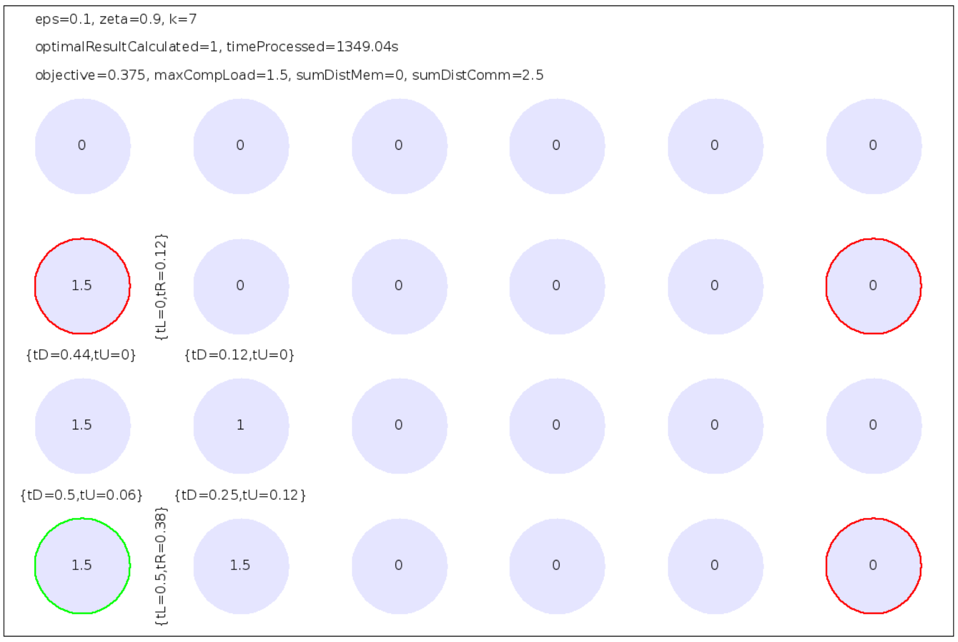
Compared to zeta=0.1 this result makes more use of parallelization as 3 memory controllers are next to each other and the communication costs part has less influence over the other subject parts. Of course, two memory controllers next to each other would be sufficient as the third one is not mapped to by any task. As the importance of a low memory controller distance is now even bigger, the mapping of leaf tasks and the root task remains the same. They are still bounded to the memory controller tiles.

The result of optimizing the cluster is a lower subject and therefore faster sort algorithm for this configuration.

A3) communication costs predominate computation costs AND memory controller distance costs predominate node distance costs

Table 5.1.A3 - results of eps=0.1, zeta=0.9

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	0.375	1.5	2.5	0
Best configuration	0.3159375	1.5	1.84375	0



```
eps=0.1, zeta=0.9, k=7
optimalResultCalculated=1, timeProcessed=1349.04s
objective=0.375, maxCompLoad=1.5, sumDistMem=0, sumDistComm=2.5
```

Figure 5.1.A31 - Intel SCC task and memory controller mapping

Now one can see that if the node distance costs are making up only 1 tenth of the distance costs, also the merge tree mapping of the original Intel SCC leads to some communication between the nodes.

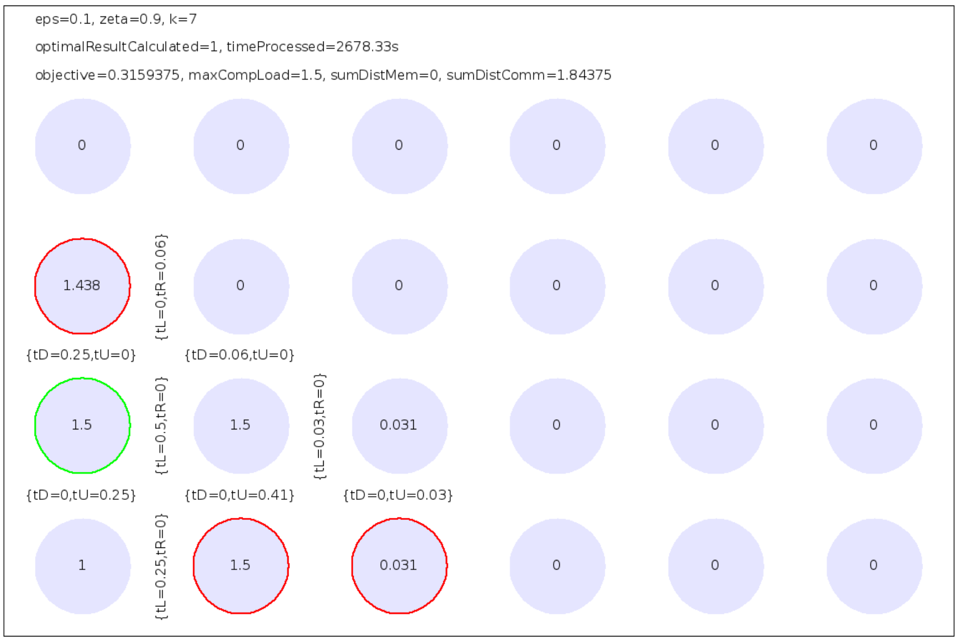

```
eps=0.1, zeta=0.9, k=7
optimalResultCalculated=1, timeProcessed=2678.33s
objective=0.3159375, maxCompLoad=1.5, sumDistMem=0, sumDistComm=1.84375
```

Figure 5.1.A32 - Best task and memory controller mapping

The important point to note here is the usage of all memory controllers. As they are close enough to each other they are all used.

The memory controllers of the best cluster configuration are pretty close to each other and they are lying in the lower left edge of the cluster. Like above, the values of the optimized cluster are better than the one of the Intel SCC.

B1) communication costs are weighted equally to computation costs AND node distance costs predominate memory controller distance costs

Table 5.1.B1 - results of eps=0.5, zeta=0.1

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	1.3875	1.5	1.34375	0.65625
Best configuration	1.3875	1.5	1.34375	0.65625

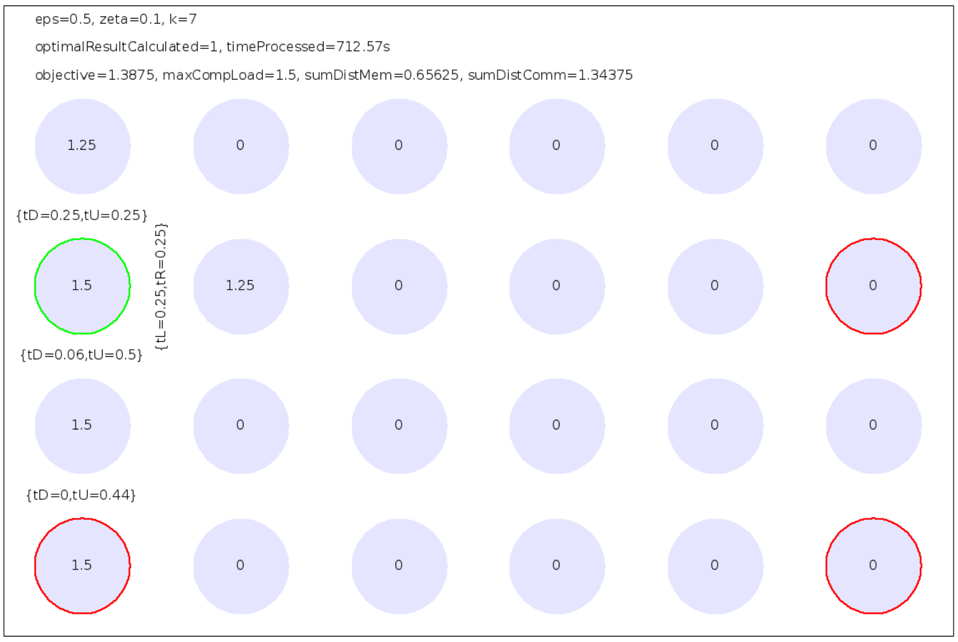

eps=0.5, zeta=0.1, k=7
optimalResultCalculated=1, timeProcessed=712.57s
objective=1.3875, maxCompLoad=1.5, sumDistMem=0.65625, sumDistComm=1.34375
1.25
0
{tD=0.25,tU=0.25}
{tL=0.25,tR=0.25}
1.5
1.25
{tD=0.06,tU=0.5}
1.5
{tD=0,tU=0.44}
1.5

Figure 5.1.B11 - Intel SCC task and memory controller mapping

Compared to the figures before where eps was set to 0.1, the memory distance costs are non-zero now. This means, that the root and leaf tasks are mapped to nodes other than the memory controller. This is caused by the higher share of computation costs over distance costs.

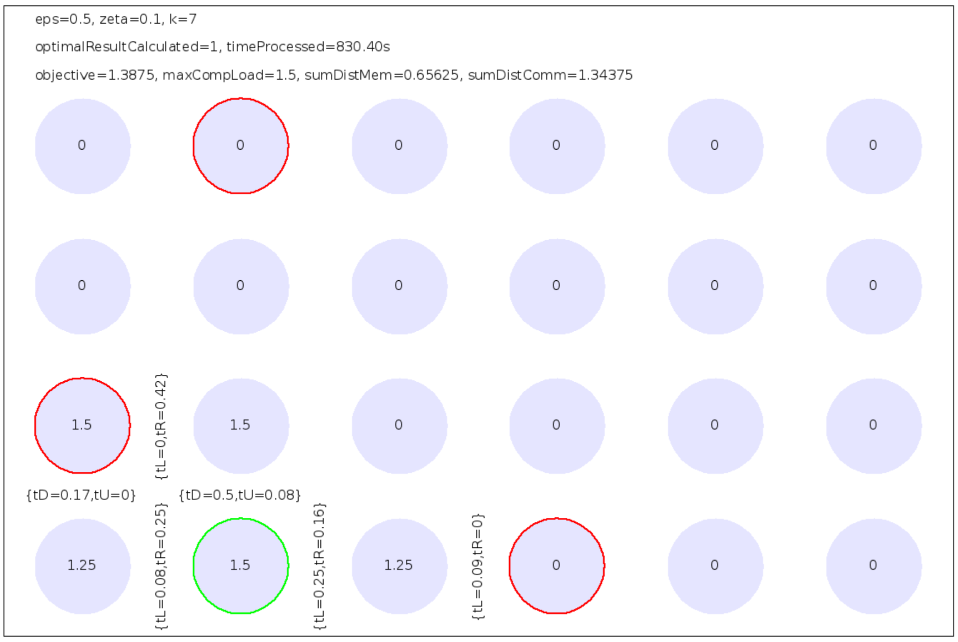


Figure 5.1.B12 - Best task and memory controller mapping

Although the mapping is different and the memory is attached to other nodes, the subject was not improvable. For this merge tree setup, the Intel SCC is optimal.

B2) communication costs are weighted equally to computation costs AND node distance costs are weighted equally to memory controller distance costs

Table 5.1.B2 - results of eps=0.5, zeta=0.5

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	1.25	1.5	1.40625	0.59375
Best configuration	1.1875	1.5	1.5	0.25

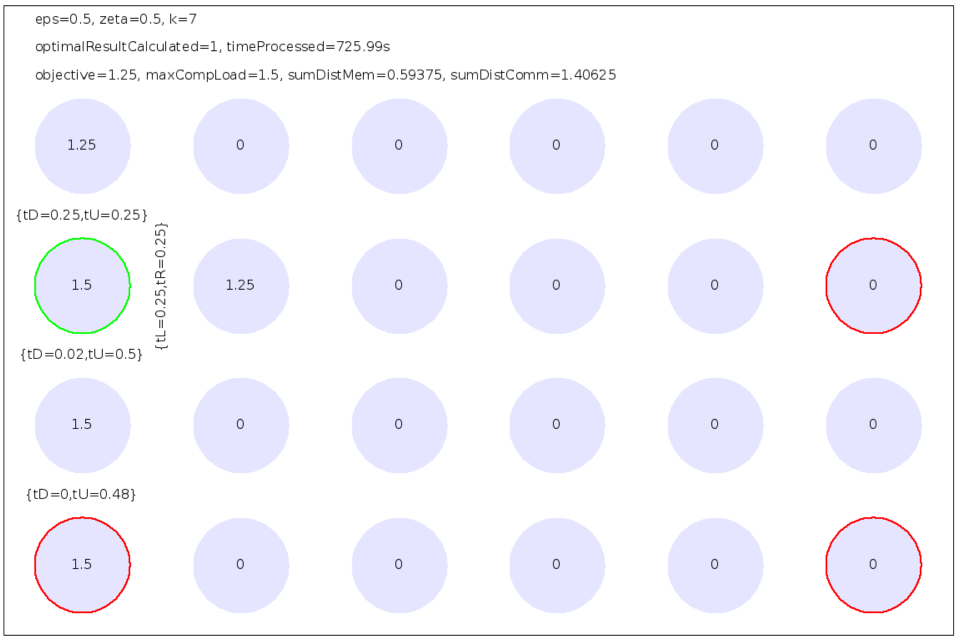


Figure 5.1.B21 - Intel SCC task and memory controller mapping

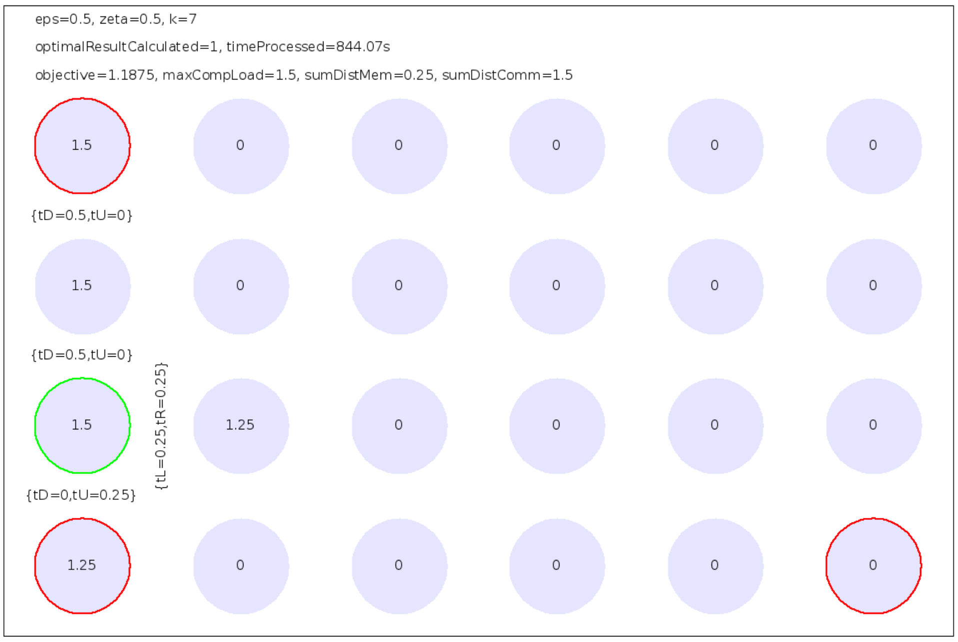

eps=0.5, zeta=0.5, k=7
optimalResultCalculated=1, timeProcessed=844.07s
objective=1.1875, maxCompLoad=1.5, sumDistMem=0.25, sumDistComm=1.5
1.5
0
0
0
0
0
{tD=0.5,tU=0}
1.5
0
0
0
0
0
{tD=0.5,tU=0}
1.5
{tL=0.25,tR=0.25}
1.25
0
0
0
0
{tD=0,tU=0.25}
1.25
0
0
0
0
0

Figure 5.1.B22 - Best task and memory controller mapping

By moving one memory controller from the right to the left side of the cluster, the merge tree mapping is improvable. This also leads to higher memory distance costs but the overall subject is better.

B3) communication costs are weighted equally to computation costs AND memory controller distance costs predominate node distance costs

Table 5.1.B3 - results of eps=0.5, zeta=0.9

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	0.69375	1	3.875	0
Best configuration	0.6375	1	2.75	0

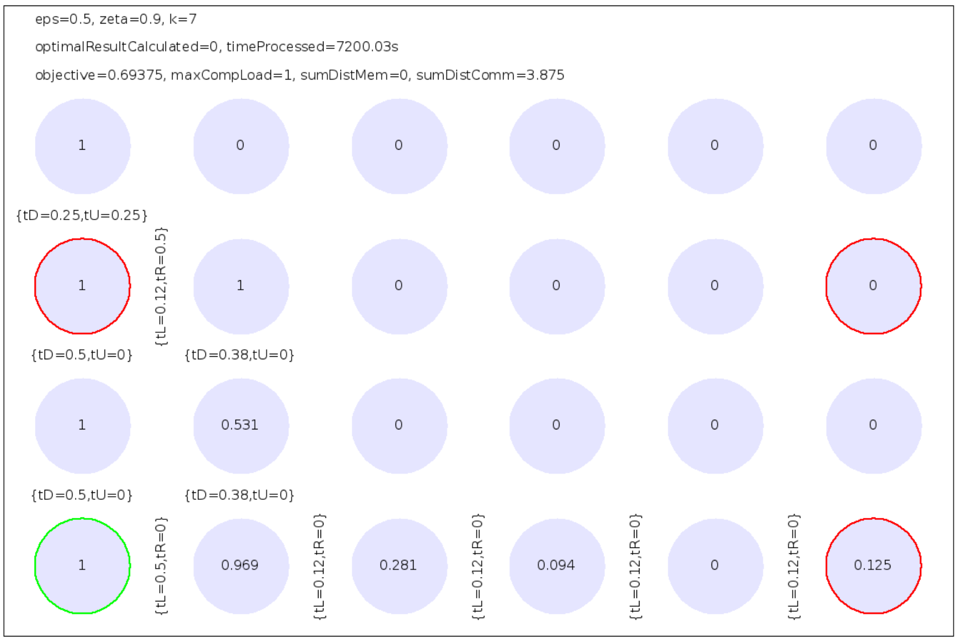

eps=0.5, zeta=0.9, k=7
optimalResultCalculated=0, timeProcessed=7200.03s
objective=0.69375, maxCompLoad=1, sumDistMem=0, sumDistComm=3.875

Figure 5.1.B31 - Intel SCC task and memory controller mapping

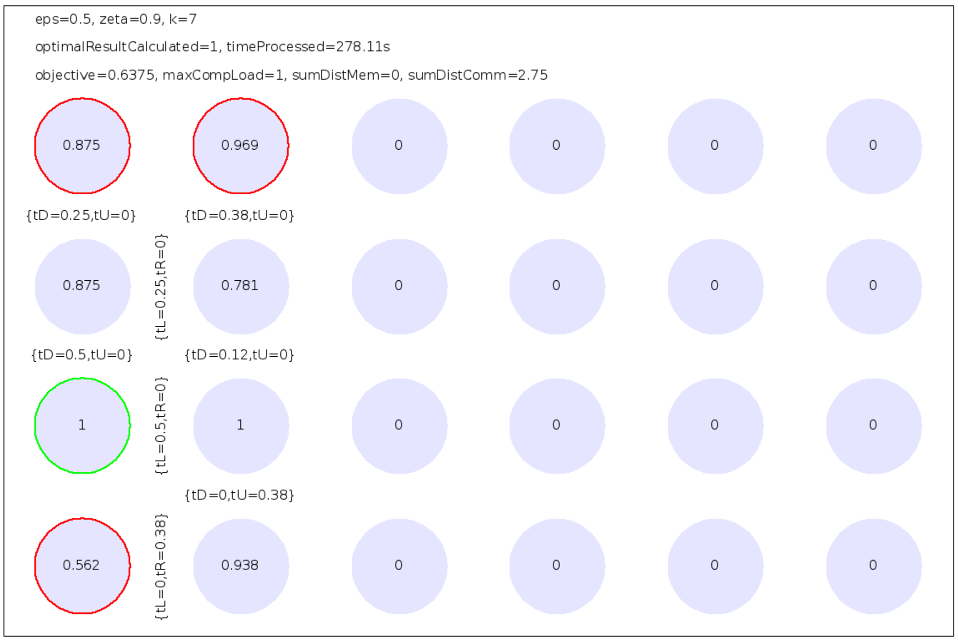


Figure 5.1.B32 - Best task and memory controller mapping

Compared to B1 and B2, the memory controller distance costs are now zero again. This is caused by the higher share of that cost part.

Another interesting change is that for both configurations the maximum computational load decreases from 1.5 to 1. This is caused by the lower importance of the node distance costs over the other ones.

By moving all memory controllers to one half of the cluster, the subject can be improved compared to the Intel configuration.

C1) computation costs predominate to communication costs AND node distance costs predominate memory controller distance costs

Table 5.1.C1 - results of eps=0.9, zeta=0.1

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	1.095	1	2	1.5
Best configuration	1.0925	1	2	1.25

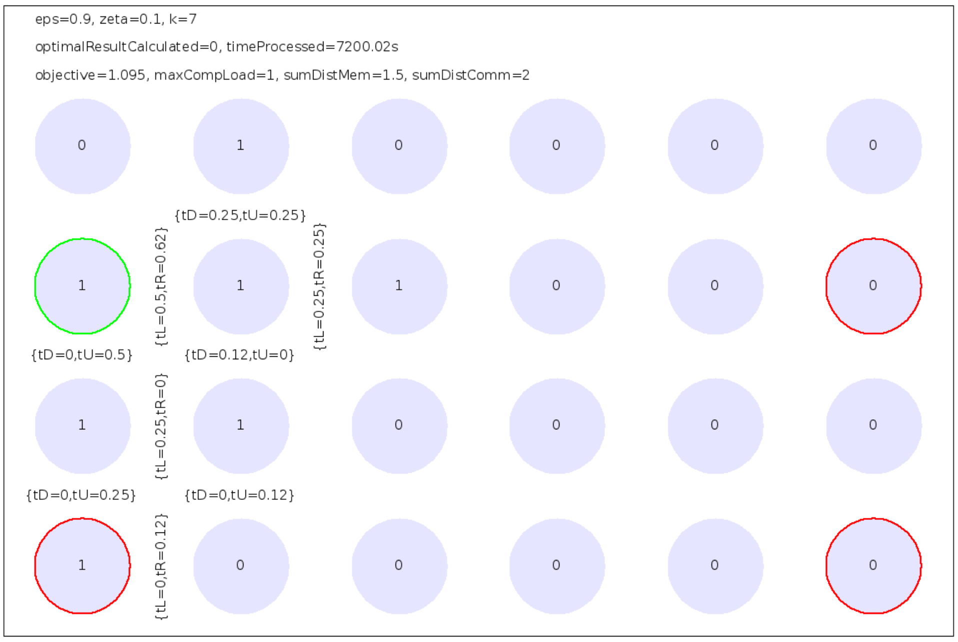


Figure 5.1.C11 - Intel SCC task and memory controller mapping

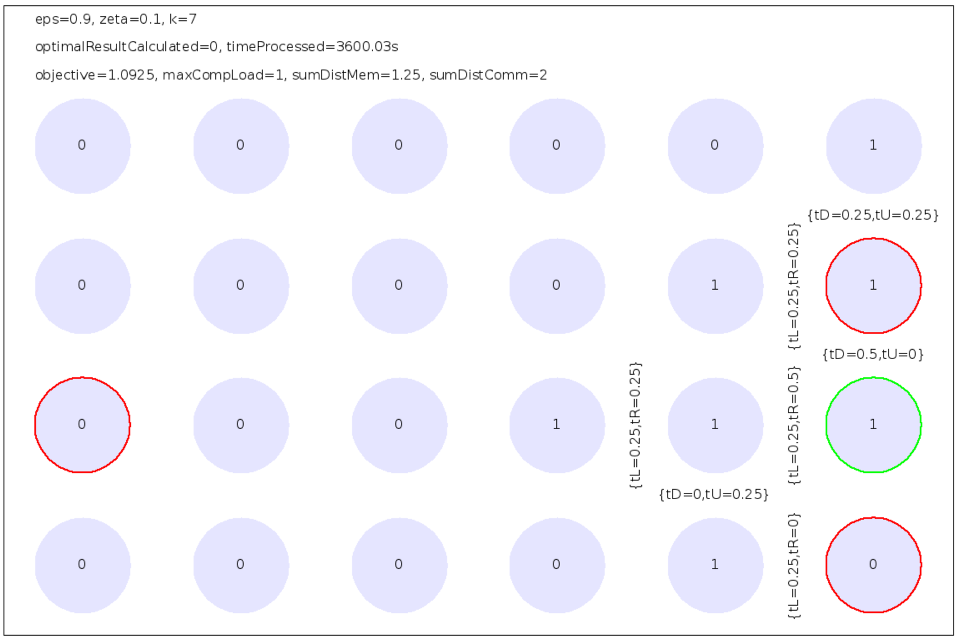


Figure 5.1.C12 - Best task and memory controller mapping

The higher the computation costs share, the more nodes are used by the merge tree. This became already visible for B1 to B3 and improved here as well.

The overall subject over the Intel SCC configuration was only improved a little by moving one of the memory controllers to the right side of the cluster.

C2) computation costs predominate to communication costs AND node distance costs are weighted equally to memory controller distance costs

Table 5.1.C2 - results of eps=0.9, zeta=0.5

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	1.071875	1	2.8125	0.625
Best configuration	1.0375	1	2.5	0.25

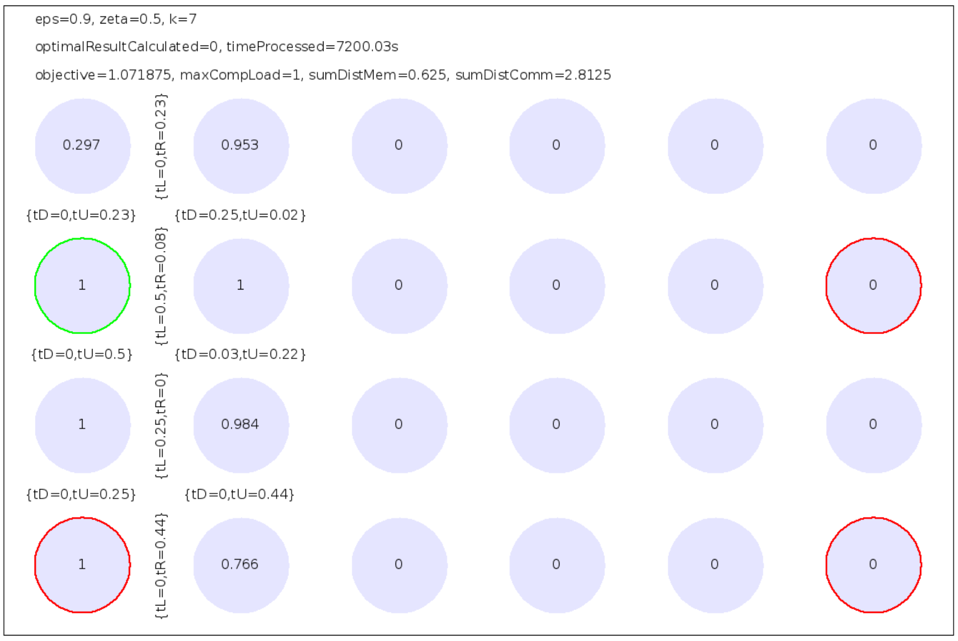


Figure 5.1.C21 - Intel SCC task and memory controller mapping

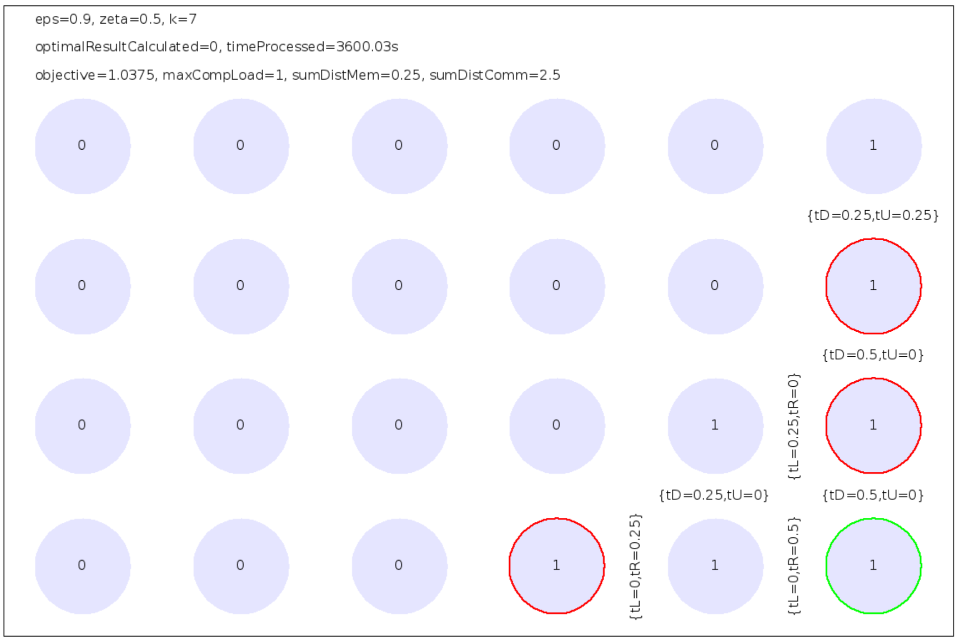


Figure 5.1.C22 - Best task and memory controller mapping

The Intel SCC configuration makes more use of the cluster (8 instead of 7 nodes are used), but the overall subject of the best configuration is better. Both memory distance and node distance costs have decreased over the original configuration.

C3) computation costs predominate to communication costs AND memory controller distance costs predominate node distance costs

Table 5.1.C3 - results of eps=0.9, zeta=0.9

	Subject	maxCompLoad	sumDistComm	sumDistMem
Intel SCC	0.93875	1	3.875	0
Best configuration	0.9275	1	2.75	0

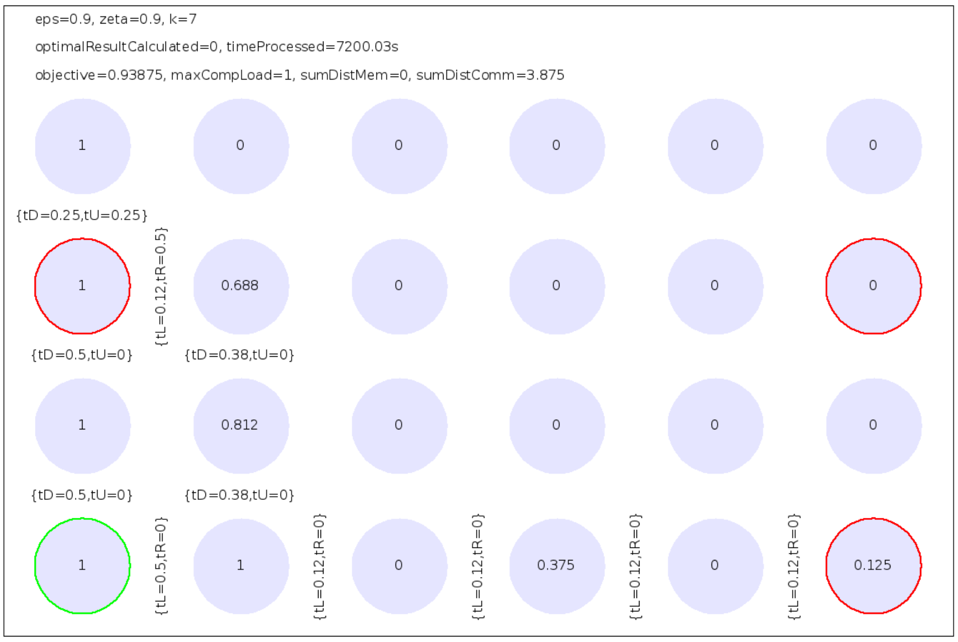

eps=0.9, zeta=0.9, k=7
optimalResultCalculated=0, timeProcessed=7200.03s
objective=0.93875, maxCompLoad=1, sumDistMem=0, sumDistComm=3.875

Figure 5.1.C31 - Intel SCC task and memory controller mapping

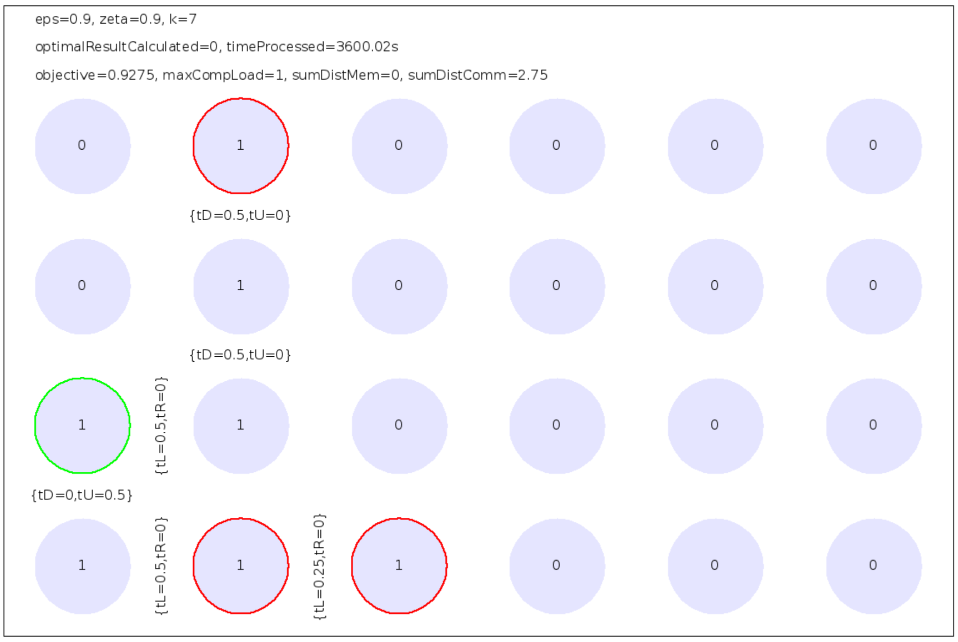


Figure 5.1.C32 - Best task and memory controller mapping

As for C2, the usage of the cluster nodes is better than for the best configuration found. Now, 9 nodes are used instead of 7. But like before, the best configuration found has a better subject, but the gap is smaller than before. As for B3, the higher share of memory distance costs leads to this part of the costs to be zero. All leaf tasks and the root task are mapped to the tiles with an attached memory controller.

Results

In 7 out of 9 cases the Intel SCC configuration was improvable. In the cases where the memory controller distance costs have only a very small influence (zeta=0.1), in two out of 3 cases there was no improvement reachable. What one also can see is that the annealing run results are not optimal. For example when looking into the best configurations for C1 and C2 one can see that if the best configuration found in C2 is applied to the setup defined in C1, this would lead to an even better result as all 4 instead of only 3 memory controllers would be used.

The distribution of the tasks over the cluster is very poor. The higher the share of computation costs, the better the distribution becomes. But even the best configuration uses only 7 nodes whereas the Intel SCC configuration uses up to 9 nodes (see C3). The reason for that is the little k value of 7. In the best case the computational load is equally distributed over the mapped tiles. If so, the load per node is 1 as the root tasks load is 1 and this is why in the best case only 7 nodes are mapped to by merge tasks.

Trying to increase the k value anyway to 9 for the original configuration leads to the following load distribution:

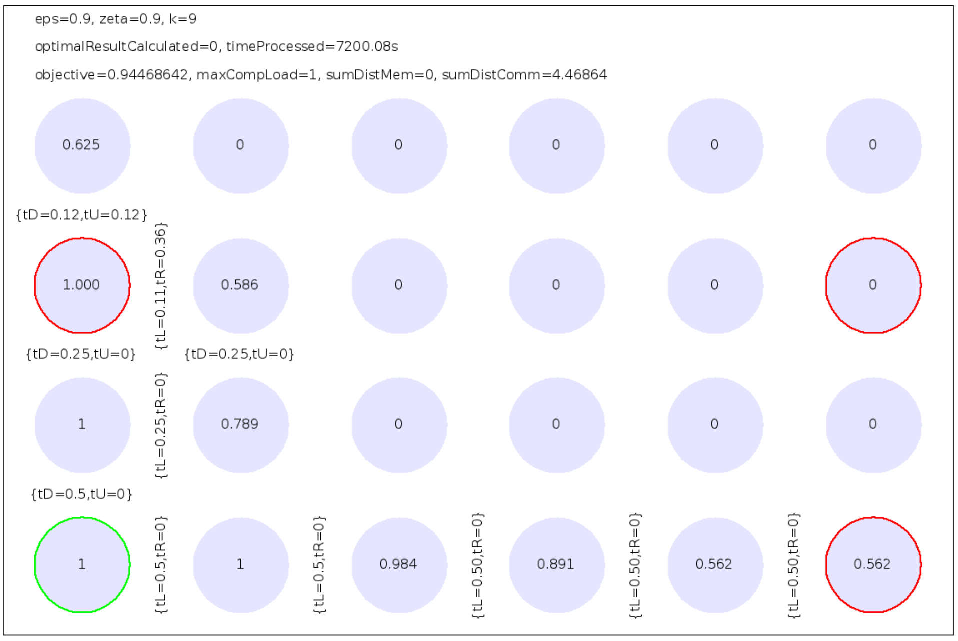


Figure 5.1.k=9 - Intel SCC configuration result for k=9

As the load is not fully equally shared over the mapped nodes, the number of mapped nodes is 11 instead of 9. But the best configuration would probably have an equal load distribution (see C2/3) so that only 9 nodes would be used.

So by setting k to 22 or higher, an equal distribution could be reached. But this doesn't make sense as then too many merge tasks (4.194.303 = 2^K-1) are to be maintained in a real implementation so that this idea can be skipped. Another approach increasing the cluster utilization is to divide the cluster into sub clusters. This is done in chapter 5.4.

5.2. Optimization with 6 and 8 MC

As a next step in further improving the architecture one can add more memory controllers to the cluster.

The following table summarizes a selected set of results of the annealing runs with 6 and 8 memory controllers. Selected means that results with no change to chapter 5.1 are omitted here. This refers to the setups 0.1_0.1 and 0.1_0.5. The row header represents the different combinations of eps and zeta, e.g. "0.1_0.9" means that eps is 0.1 and zeta is 0.9. The table values are encoded with the following schema: $subject_$maxCompLoad_$sumDistComm_$sumDistMem.

Table 5.2 - Results for higher MC numbers

Costs	Intel	4 MC	6 MC	8 MC
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.2988_1.75_1.375_0	0.285_1.5_1.5_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.375_1.5_1.344_0.406	see MC6
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.117_1_2.344_0.125	see MC6
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.625_1_2.5_0	0.623_1_2.469_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.088_1_2_0.75	1.085_1_2_0.5
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.023_1_2.219_0.25	1.025_1_2.469_0.031
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.925_1_2.5_0	see MC6

The improvement with 6 and 8 memory controllers over the configuration with 4 memory controllers for eps=0.1 and zeta=0.9 is misleading as the one with 8 memory controllers uses only 4 of them actively. The only reason for the best configuration with 4 memory controllers to not have the same result is the nature of the annealing algorithm not to try every single neighbor of the current best solution. The following figure demonstrates that.

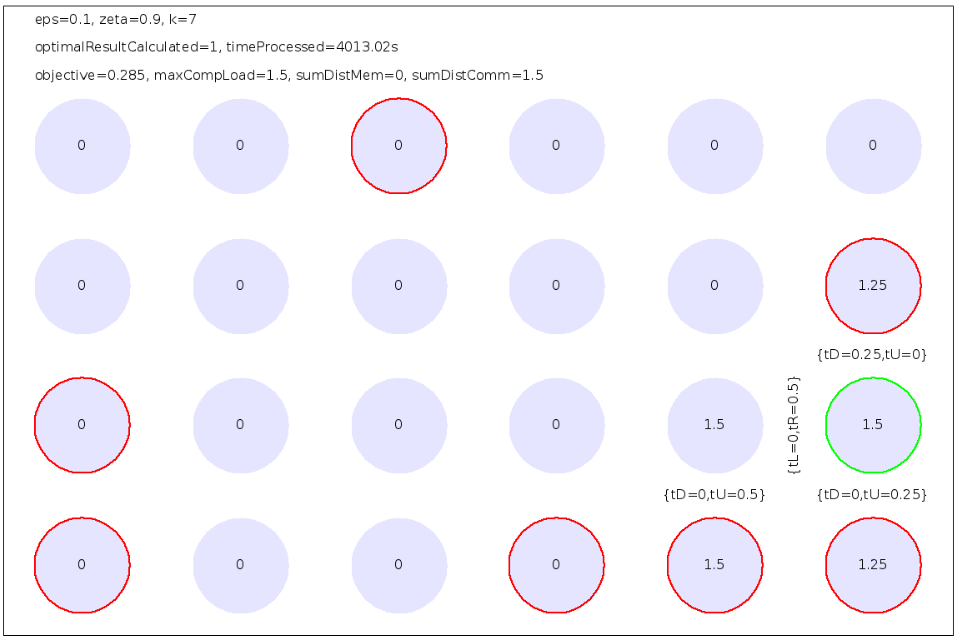
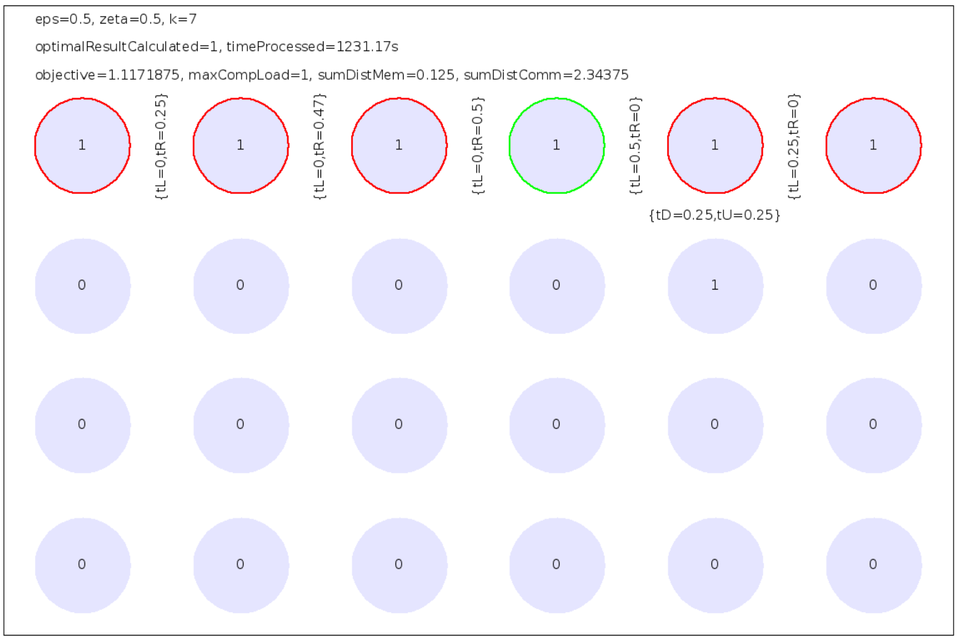

eps=0.1, zeta=0.9, k=7
optimalResultCalculated=1, timeProcessed=4013.02s
objective=0.285, maxCompLoad=1.5, sumDistMem=0, sumDistComm=1.5
0
1.25
1.5
{tD=0.25,tU=0}
{tL=0,tR=0.5}
{tD=0,tU=0.5}
{tD=0,tU=0.25}

Figure 5.2.A3_8 - Best task and memory controller mapping for 8 MC

One can see that only 4 memory controllers out of 8 are used. So the same result could be achieved with only 4 memory controllers, too. The same applies to the runs with eps=0.5 and zeta=0.1 as well as to eps=0.9 and zeta=0.1.

The following picture illustrates the result of a real improvement with 6 memory controllers.


eps=0.5, zeta=0.5, k=7

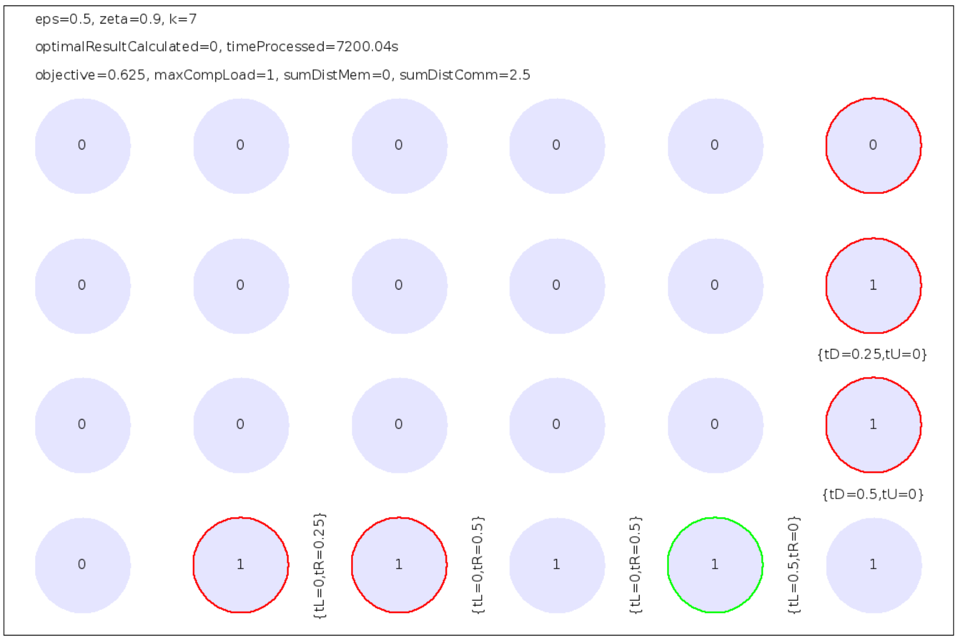
optimalResultCalculated=1, timeProcessed=1231.17s

objective=1.1171875, maxCompLoad=1, sumDistMem=0.125, sumDistComm=2.34375

Figure 5.2.B2_6 – Best task and memory controller mapping for 6 MC

Here one can see that the placement of all 6 memory controllers in the first row of the cluster is superior over configurations with 4 MC.

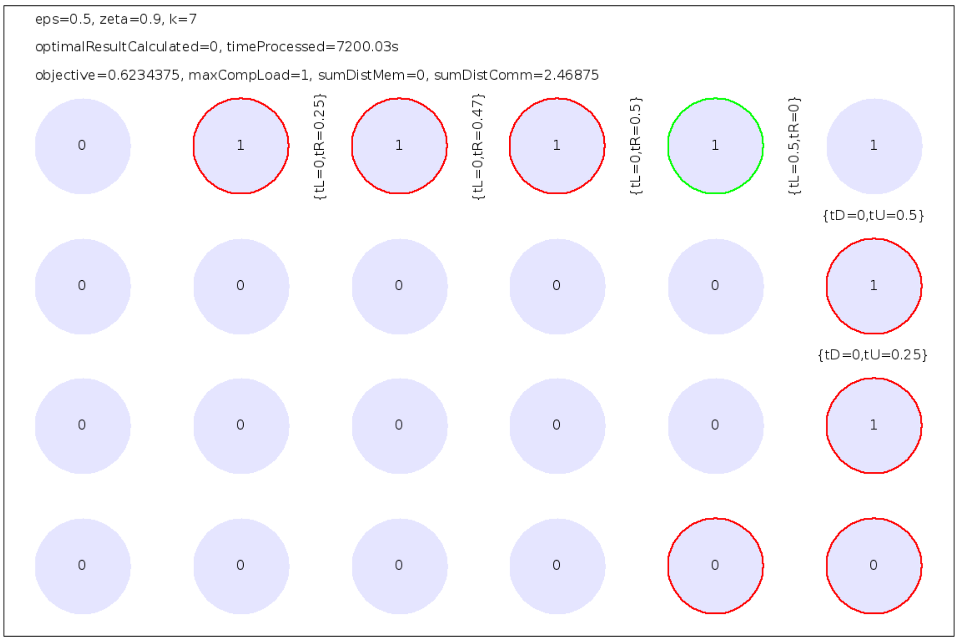
The following two figures present the best configuration found for eps=0.5 and zeta=0.9.


eps=0.5, zeta=0.9, k=7

optimalResultCalculated=0, timeProcessed=7200.04s

objective=0.625, maxCompLoad=1, sumDistMem=0, sumDistComm=2.5

Figure 5.2.B3_6 - Best task and memory controller mapping for 6 MC

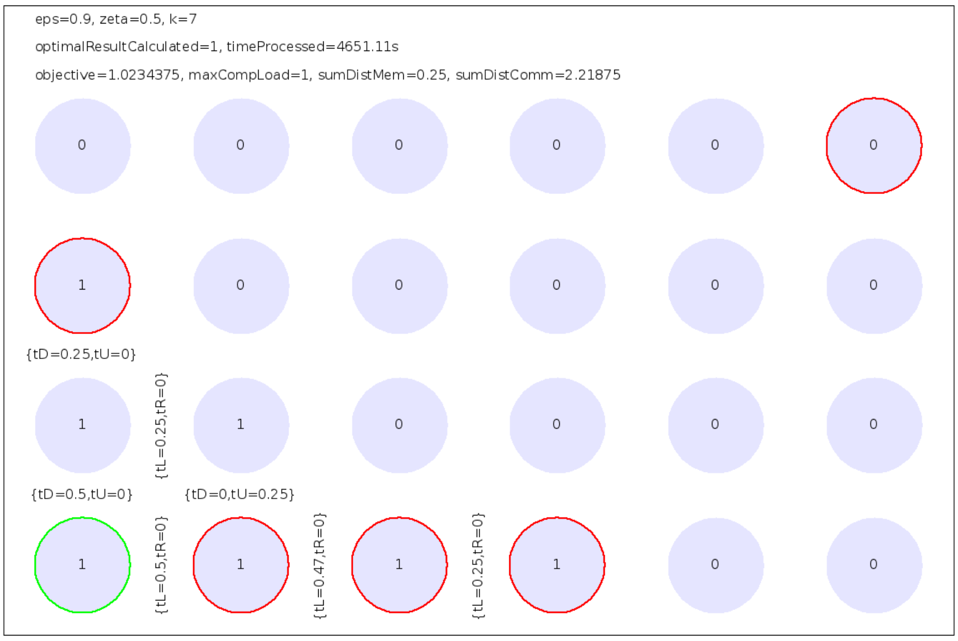

eps=0.5, zeta=0.9, k=7

optimalResultCalculated=0, timeProcessed=7200.03s

objective=0.6234375, maxCompLoad=1, sumDistMem=0, sumDistComm=2.46875

Figure 5.2.B3_8 - Best task and memory controller mapping for 8 MC

The next picture only shows the result for eps=0.9 and zeta=0.5 with 6 memory controllers as the annealing run with 8 memory controllers has a worse result.


eps=0.9, zeta=0.5, k=7

optimalResultCalculated=1, timeProcessed=4651.11s

objective=1.0234375, maxCompLoad=1, sumDistMem=0.25, sumDistComm=2.21875

Figure 5.2.C2_6 - Best task and memory controller mapping for 6 MC

The last picture shows the best configuration for eps=0.9 and zeta=0.9 with 6 memory controllers only as the annealing run with 8 memory controllers has the same subject.

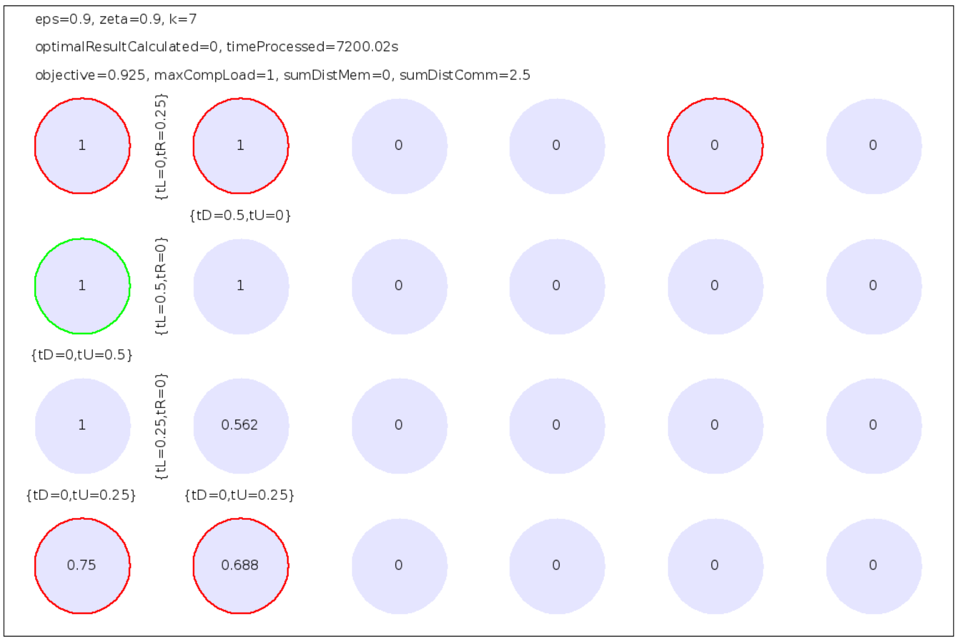


Figure 5.2.C3_6 - Best task and memory controller mapping for 6 MC

The addition of two memory controllers only makes sense under certain circumstances. For this model, there one can find 7 setups (eps and zeta) where an improvement is achievable, especially for eps=0.5 and zeta=0.5 the improvement is significant. But adding even more memory controllers seems not to have the same effect. In the simulation a speedup is found in only 3 out of 9 cases. Of course, this could be caused by the fact that the annealing algorithm has not enough time (100% more time is given for 8 MC compared to 6 MC) to sufficiently investigate the neighborhood. But the time must be limited in order to get results at all.

Later investigations in chapter 5.4 demonstrate, that adding more than 6 memory controllers to the cluster has actually a very positive impact - at least if a lot more tasks are to mapped.

5.3. Certain model configurations

Another approach in finding a better configuration is to manually define a "good" one and test it. The following figures and tables present the outcome of that idea. The summary can be found after the result listing.

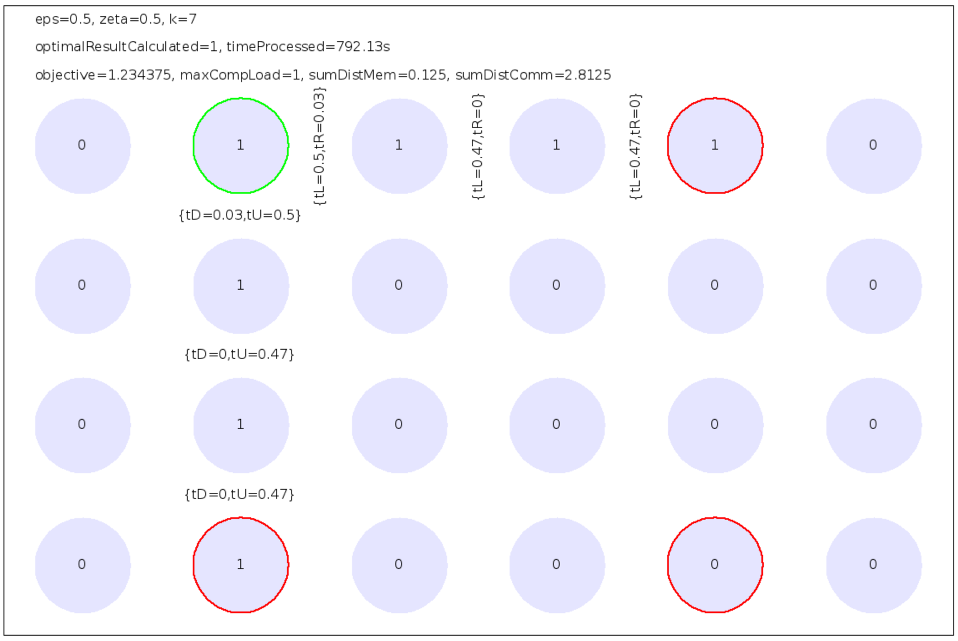
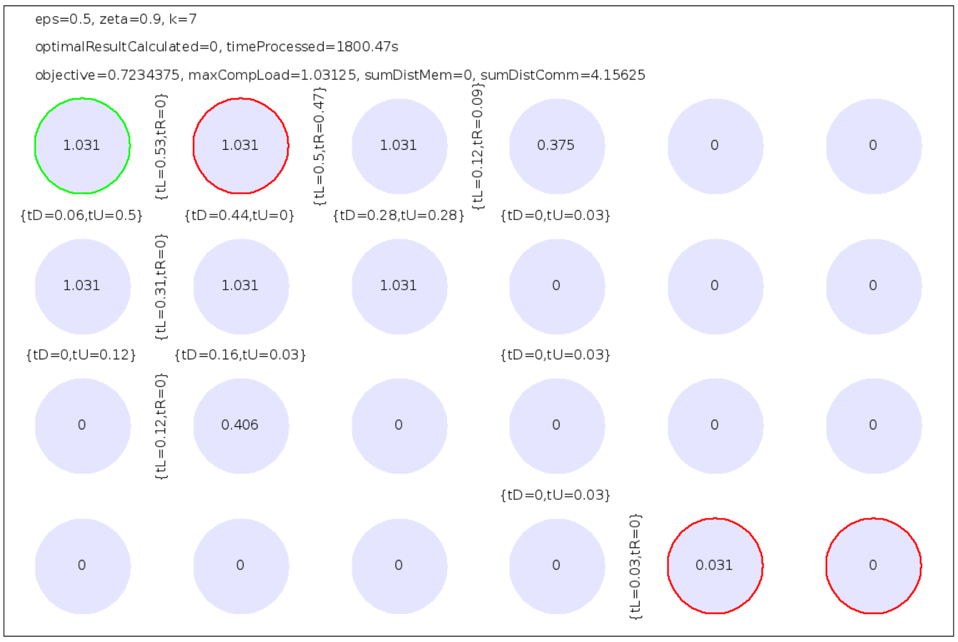

eps=0.5, zeta=0.5, k=7
optimalResultCalculated=1, timeProcessed=792.13s
objective=1.234375, maxCompLoad=1, sumDistMem=0.125, sumDistComm=2.8125
{tL=0.5,tR=0.03}
{tL=0.47,tR=0}
{tL=0.47,tR=0}
{tD=0.03,tU=0.5}
{tD=0,tU=0.47}
{tD=0,tU=0.47}

Figure 5.3.A - MC positions 1_4_19_22, root is 1

The row headers and the values of that table are to be read as like as in chapter 5.2. Green marked entries in the "current" column represent values that are better than the Intel SCC configuration. Red ones stand for the opposite.

Table 5.3.A - MC positions 1_4_19_22, root is 1

	Intel	Best	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.7_7_0_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.37_1_3_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.417_1.5_1.344_1.25
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.234_1_2.813_0.125
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.65_1_3_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.099_1_2.063_1.375
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.047_2.813_0.125
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.93_1_3_0


eps=0.5, zeta=0.9, k=7

optimalResultCalculated=0, timeProcessed=1800.47s

objective=0.7234375, maxCompLoad=1.03125, sumDistMem=0, sumDistComm=4.15625

Figure 5.3.B - MC positions 0_1_22_23, root is 0

Table 5.3.B - MC positions 0_1_22_23, root is 0

	Intel	Best	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.371_2.25_1.625_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.438_1.5_1.344_1.656
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.328_1.438_1.813_0.625
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.723_1.031_4.156_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.111_1_2.188_1.438
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.088_1_2.344_1.406
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.943_1_4.25_0

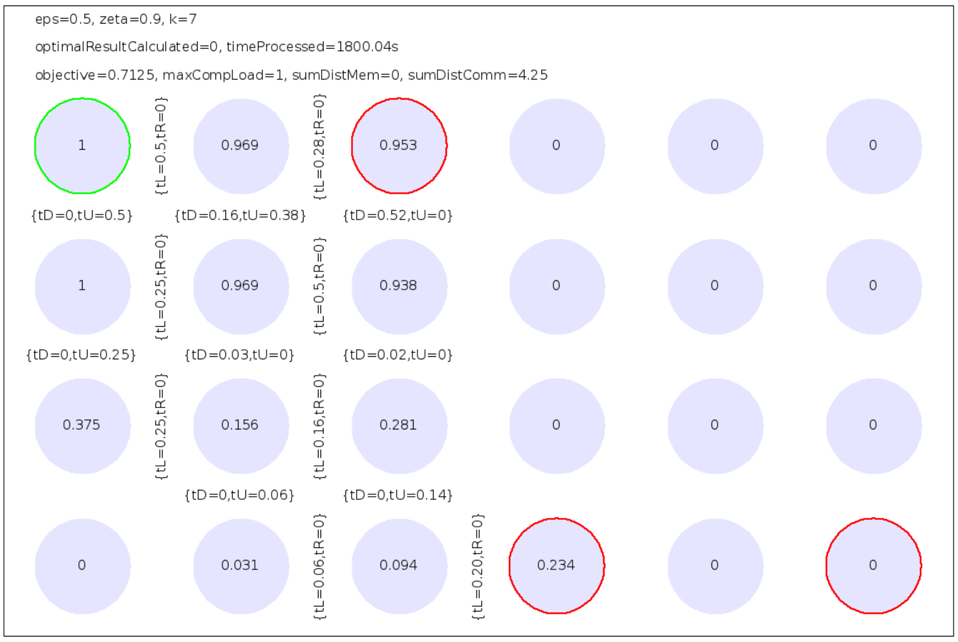

```
eps=0.5, zeta=0.9, k=7
optimalResultCalculated=0, timeProcessed=1800.04s
objective=0.7125, maxCompLoad=1, sumDistMem=0, sumDistComm=4.25
```

Figure 5.3.C - MC positions 0_2_21_23, root is 0

Table 5.3.C - MC positions 0_2_21_23, root is 0

	Intel	Best	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.7_7_0_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.375_1.5_2.5_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.447_1.5_1.344_1.844
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.344_1.438_2.25_0.25
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.713_1_4.25_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.11_1_2_3
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.075_1_2.719_0.781
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.939_1_3.875_0

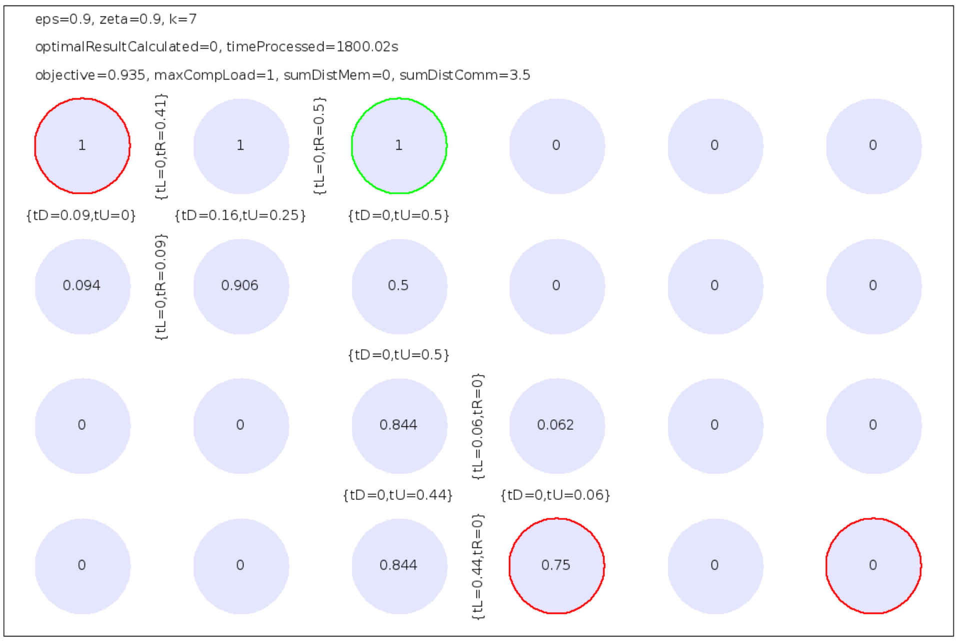


Figure 5.3.D - MC positions 0_2_21_23, root is 2

Table 5.3.D - MC positions 0_2_21_23, root is 2

	Intel	Best	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.7_7_0_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.369_1.438_2.5_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.359_1_2.625_0.813
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.675_1_3.5_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.095_1_2_1.5
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.078_1_2.906_0.656
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.935_1_3.5_0

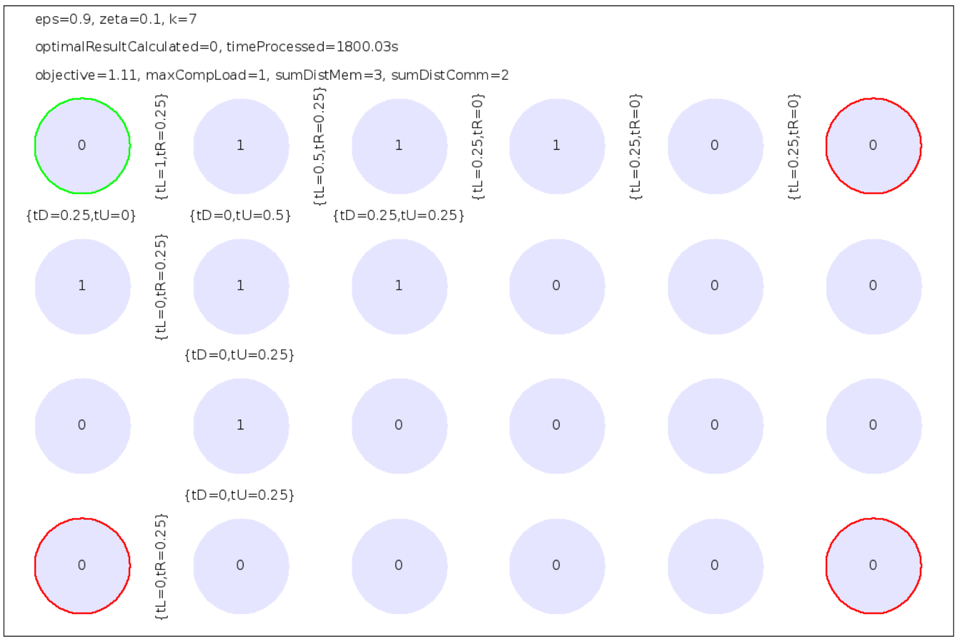


Figure 5.3.E - MC positions 0_5_18_23, root is 0

Table 5.3.E - MC positions 0_5_18_23, root is 0

	Intel	Best with 4 MC	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.7_7_0_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.41_2.188_2.125_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.438_1.5_1.344_1.656
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.375_1.5_2_0.5
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.7_1_4_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.11_1_2_3
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.075_1_2.5_1
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.94_1_4_0

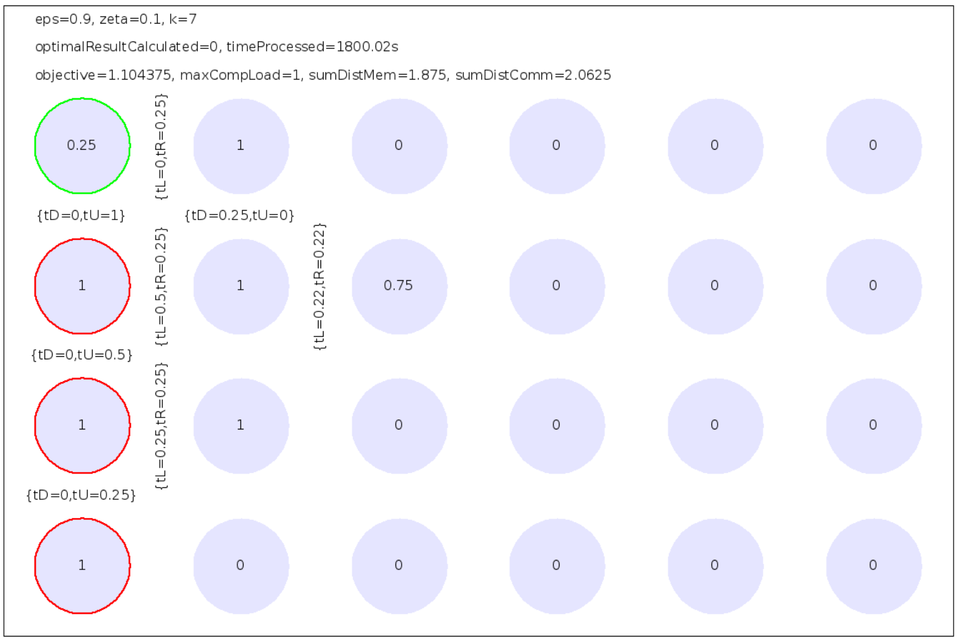
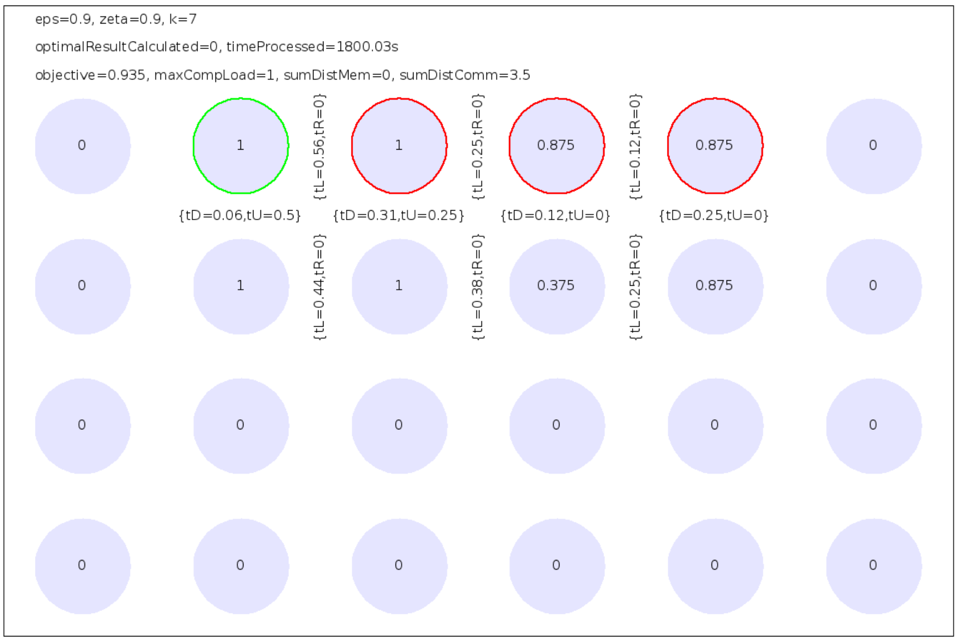


Figure 5.3.F - MC positions 0_6_12_18, root is 0

Table 5.3.F - MC positions 0_6_12_18, root is 0

	Intel	Best with 4 MC	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.346_1.375_2.313_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.417_1.5_1.344_1.25
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.234_1_2.438_0.5
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.663_1_3.25_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.104_1_2.063_1.875
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.048_1_2.219_0.75
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.933_1_3.25_0


eps=0.9, zeta=0.9, k=7

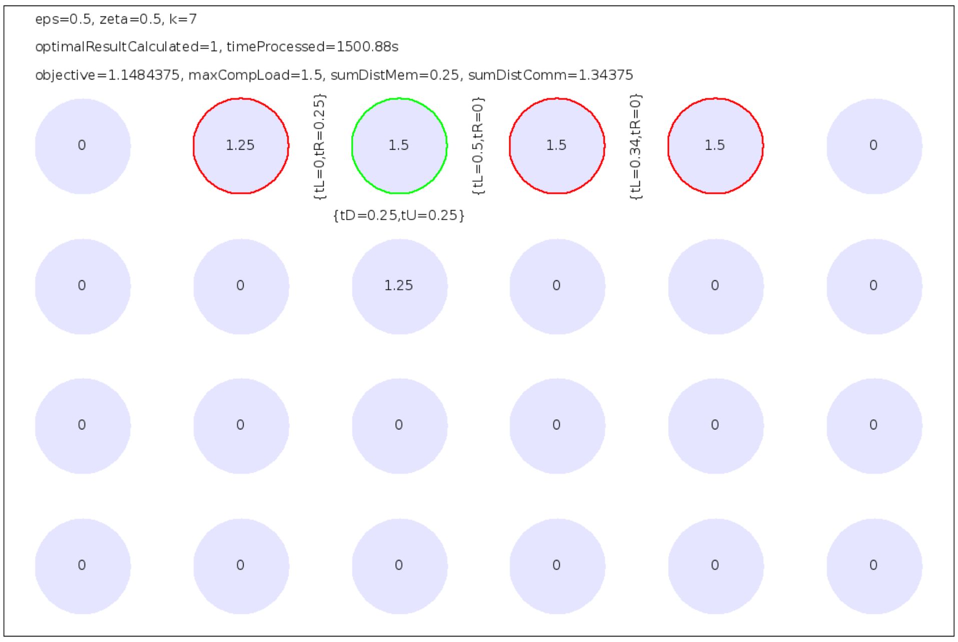
optimalResultCalculated=0, timeProcessed=1800.03s

objective=0.935, maxCompLoad=1, sumDistMem=0, sumDistComm=3.5

Figure 5.3.G - MC positions 1_2_3_4, root is 1

Table 5.3.G - MC positions 1_2_3_4, root is 1

	Intel	Best with 4 MC	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.346_1.375_2.313_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.380_1.5_1.344_0.5
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.234_1.313_2.031_0.281
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.669_1_3.375_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.093_1_2_1.25
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.048_1_2.422_0.547
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.935_1_3.5_0


eps=0.5, zeta=0.5, k=7

optimalResultCalculated=1, timeProcessed=1500.88s

objective=1.1484375, maxCompLoad=1.5, sumDistMem=0.25, sumDistComm=1.34375

Figure 5.3.H - MC positions 1_2_3_4, root is 2

Table 5.3.H - MC positions 1_2_3_4, root is 2

	Intel	**Best with 4 MC**	**Current**
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.299_1.75_1.375_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.367_1.5_1.344_0.25
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.148_1.5_1.344_0.25
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.638_1_2.75_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.097_1_2.094_0.844
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.036_1_2.406_0.313
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.928_1_2.75_0

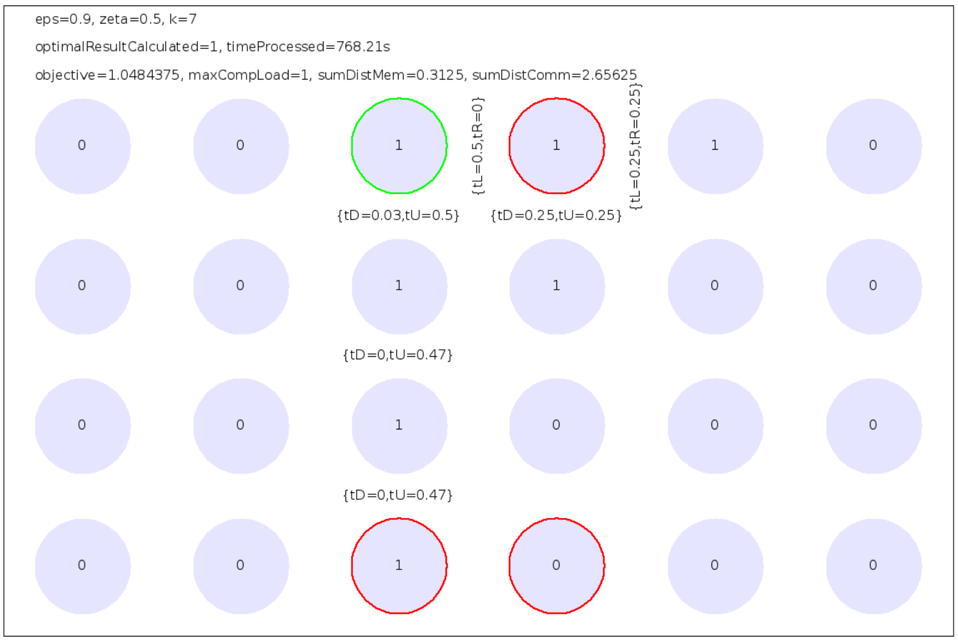


Figure 5.3.I - MC positions 2_3_20_21, root is 2

Table 5.3.I - MC positions 2_3_20_21, root is 2

	Intel	Best with 4 MC	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.368_2.188_1.656_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.392_1.5_1.344_0.75
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.25_1_2.547_0.453
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.65_1_3_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.103_1_2_2.25
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.048_1_2.656_0.313
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.93_1_3_0

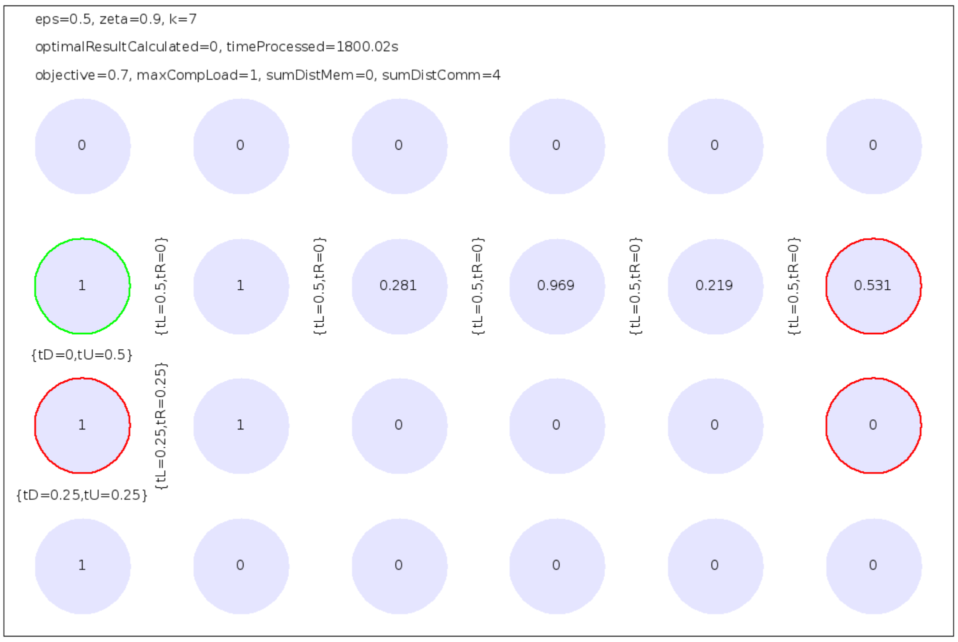


Figure 5.3.J - MC positions 6_11_12_17, root is 6

Table 5.3.J - MC positions 6_11_12_17, root is 6

	Intel	Best with 4 MC	Current
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.375_1.5_2.5_0	0.316_1.5_1.844_0	0.371_2.25_1.625_0
0.5_0.1	1.388_1.5_1.344_0.656	1.388_1.5_1.344_0.656	1.459_1.5_1.344_2.094
0.5_0.5	1.25_1.5_1.406_0.594	1.188_1.5_1.5_0.25	1.273_1.5_1.516_0.578
0.5_0.9	0.694_1_3.875_0	0.638_1_2.75_0	0.7_1_4_0
0.9_0.1	1.095_1_2_1.5	1.093_1_2_1.25	1.095_1_2_1.5
0.9_0.5	1.072_1_2.813_0.625	1.038_1_2.5_0.25	1.072_1_2.75_0.688
0.9_0.9	0.939_1_3.875_0	0.928_1_2.75_0	0.94_1_4_0

One can see that like in the previous chapters most of the processing nodes are not used. Like there this is caused by the small tree depth. For certain algorithm setups (eps and zeta values) the majority of the configuration results are better than the Intel configuration. But with the exception of one configuration all defined solutions are worse than the original Intel SCC configuration for at least one tested eps and zeta combination.

The mentioned exception is displayed in figure 5.3.G. All memory controller nodes are centered in the first row. With this configuration, the model outperforms the Intel model in 8 out of 9 algorithm setups. There is also an interesting effect visible. If the root co-task position is changed by one tile (see next figure 5.3.H), the results become even better. In 4 cases they are even better than the annealing results (which is caused by the fact that the algorithm is searching for good solutions randomly and therefore not all local minimums can be found). The only drawback is that for eps=0.9 and zeta=0.1 the result is not as good as the Intel one. But the combined result summarizing the two possible root co-task locations is the overall winner. The configuration where all memory controllers are centered in the first row of the cluster is under all tested circumstances better than or at least equal to the Intel SCC.

Of course, for more tree levels (and other algorithms) the optimal memory controller placement can and will look pretty much different.

5.4. Mirrored configuration

The previous investigations of chapter 5 have one major issue. The configurations tested resulted in clusters that are not fully used and therefore the outcome is at least questionable. The poor distribution of tasks is because the minimum load of all nodes is defined by the root task load which is 1. As for each level of the merge sort tree the load is 1, the overall load is the number of levels of the tree (k) that needs to be mapped to the nodes. Assuming equally distributed tasks (so the load per node is pretty much equal, see also chapter 3) the ideal load per node is 1 as the minimum load is defined by the root node. So for a total load of k to be mapped to nodes, the ideal number of nodes is k, too. So by setting the value k to the number of nodes, the distribution problem could be solved. But as more tree levels lead to even more tasks (2^(k) - 1) and more tasks to more inequalities, such a change is practically impossible as more inequalities lead to a very high calculation time per configuration. If each calculation takes hours, an annealing run can take days and possibly requires too much memory. This is unacceptable.

In order to cope with that the same approach as reported in chapter 3.2 of dividing the model into logical units and mirror them to get the final results is used here. This means to divide the big global merge sort tree into sub trees. So by concentrating on a smaller part of the cluster the same k value leads to a much better utilization.

A side effect of this principle is that it leads to a full optimization because of two reasons. First there are only a few locations left where the memory controllers can be placed at. Adding some constraints like "only on edges" and "no duplicates" even more reduces this little number. Secondly, each of those configuration options is faster calculated because fewer inequalities (compared to a full cluster) are taken into consideration and so the Gurobi solver can very likely fully solve each model in an acceptable time.

Before starting to cut the cluster into various parts and optimizing them, it makes sense to filter the different possible divisions in order to get fewer results to investigate. If one defines the full usage of all nodes as filter criteria, a lot of the possibilities can be skipped because if too many nodes are within a sub cluster, then the k value becomes too high and therefore the calculation time is not acceptable any more.

This single filter criterion is sufficient as only one division remains! It is to cut the cluster into 4 quadrants: each part has 2 rows and 3 columns - like in the paper [3]. This makes up 6 nodes or tiles per cluster. In order to see the influence of k, the optimization runs are done with the following values of k: 5, 6 and 7.

The memory controller count of the Intel SCC per sub cluster is one. So for the optimization, the same value is tested. But in order to see the impact of more memory controllers, 2 and 3 memory controllers are investigated as well.

The following table summarizes the results for k set to 5. It is to be read as like as in chapter 5.2.

Table 5.4.A - Results for runs with k set to 5

	Intel	Best	2 MC	3 MC
0.1_0.1	0.5_5_0_0	0.5_5_0_0	0.5_5_0_0	0.5_5_0_0
0.1_0.5	0.5_5_0_0	0.5_5_0_0	0.5_5_0_0	0.5_5_0_0
0.1_0.9	0.38_2_2_0	0.38_2_2_0	0.305_1.25_2_0	0.276_1.75_1.125_0
0.5_0.1	1.375_1.5_1.25_1.25	1.375_1.25_1.5_1.5	1.331_1_1.75_0.875	1.313_1_1.75_0.5
0.5_0.5	1.344_1.688_1.438_0.563	1.344_1.188_2.188_0.813	1.125_1.25_2_0	1.063_1_1.875_0.375
			1.125_1.25_1.625_375	1.063_1_1.75_0.5
				1.063_1.25_1.625_0.125
0.5_0.9	1.069_1.688_1.688_0.313	1.069_1.188_2.188_0.813	0.65_1_3_0	0.619_1_2.375_0
0.9_0.1	1.075_1_1.75_1.75	1.075_1_1.75_1.75	1.066_1_1.75_0.875	1.063_1_1.75_0.5
0.9_0.5	1.075_1_1.75_1.75	1.075_1_2_1.5	1.031_1_1.75_0.875	1.013_1_1.75_0.5
				1.013_1_1.875_0.375
0.9_0.9	1.02_1_3_1	1.015_1_2.5_1	0.93_1_3_0	0.924_1_2.375_0

The table contains for some setups more than one value. This is because different configurations are found that have the same subject but differing sub cost. The following things can be seen:

- Load balancing
  - For the runs with high communication cost share the load is totally unbalanced for all configurations. All tasks are mapped to a single node. The only exception are the runs with a dominating memory controller distance share (zeta=0.9). But even there the tasks are not equally spread over all nodes.
  - The runs with an equal cost share between communication and computation costs show in many cases with 2 and 3 memory controllers that the maximum computational load per node is 1. This does not mean that the tasks are equally distributed as the total load for k=5 is smaller than the number of nodes. But as it cannot get better, this pretty good.
  - For the run with a dominating computation cost share the load is balanced perfectly for all configurations if one considers the little k value.
  - The entries marked green are the runs where the maximum computational load per node is 1.
- Improvements for the best configuration with 1 MC
  - For k=5 the improvement is mostly related to a better load balancing (less maximum computation costs). This is of course traded against higher costs for communication.
  - In all but one case the overall costs are equal to the Intel SCC result. Only in one case the result can be improved.
- Improvements for the best configuration with 2 MC
  - The memory controller distance costs can be decreased. Sometimes this cost part becomes 0.
  - The load balancing tends to get better. Only one exception can be found.
  - The communication costs are increased in many cases.
  - In general: In 7 out of 9 runs the overall costs are better and sometimes dramatically better than the best configuration with 1 memory controller.
- Improvements for the best configuration with 3 MC
  - In contrast to the runs with 2 MC here is one more case leading to a worse load balancing. But as before, the higher MC count tends to improve the balancing.
  - The same observations as like as for the runs with 2 MC applies here. But the overall costs are even better in those 7 runs.

Table 5.4.B - Results for runs with k set to 6

	Intel	Best	2 MC	3 MC
0.1_0.1	0.6_6_0_0	0.6_6_0_0	0.6_6_0_0	0.6_6_0_0
0.1_0.5	0.6_6_0_0	0.6_6_0_0	0.575_3.5_0.5_0	0.575_3.5_0.5_0
0.1_0.9	0.38_2_2_0	0.38_2_2_0	0.33_1.5_2_0	0.29_2_1_0
0.5_0.1	1.425_1.5_1.25_2.25	1.375_1.5_1.25_1.25	1.35_1.5_1.25_0.75	1.338_1.5_1.25_0.5
0.5_0.5	1.422_1.344_1.688_1.313	1.375_1.5_1.25_1.25	1.219_1_2.25_0.625	1.125_1_2.25_0.25 1.125_1_2.188_0.313
0.5_0.9	1.1_1.5_2.5_0.5	1.084_1.344_2.344_0.656	0.65_1_3_0	0.631_1_2.625_0
0.9_0.1	1.113_1_2.125_2.125	1.108_1_2.125_1.625	1.099_1_2.125_0.75	1.095_1_2.125_0.375
0.9_0.5	1.113_1_2.125_2.125	1.088_1_2.125_1.625	1.044_1_2.438_0.438	1.025_1_2.125_0.375 1.025_1_2.188_0.313
0.9_0.9	1.026_1_3.625_1	1.018_1_2.75_1	0.93_1_3_0	0.926_1_2.625_0

In contrast to the runs with k=5 before, the maximum load value of 1 here really refers to an equal load distribution over the nodes. The following figure shall prove that.

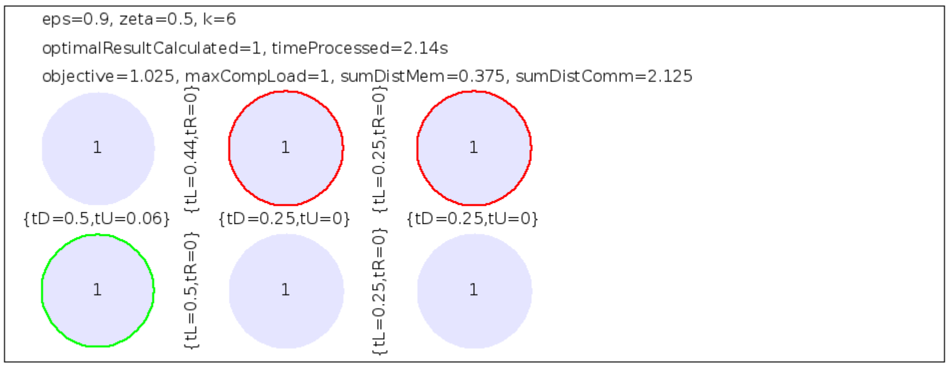


Figure 5.4.A - Equal task distribution

The observations for k=5 are applicable here as well with some little exceptions. One of those minor differences is related to the load balancing for eps=0.5 with 2 and 3 memory controllers.

Table 5.4.C - Results for runs with k set to 7

	Intel	Best	2 MC	3 MC
0.1_0.1	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0	0.7_7_0_0
0.1_0.5	0.7_7_0_0	0.7_7_0_0	0.625_4_0.5_0	0.625_4_0.5_0
0.1_0.9	0.414_2.344_2_0	0.414_2.344_2_0	0.355_1.75_2_0	0.316_1.5_1.844_0
0.5_0.1	1.463_1.5_1.344_2.156	1.422_1.5_1.344_1.344	1.388_1.5_1.344_0.656	1.375_1.5_1.344_0.406
0.5_0.5	1.5_1.438_1.781_1.344	1.422_1.5_1.469_1.219	1.25_1.5_1.5_0.5	1.188_1.5_1.406_0.344
0.5_0.9	1.117_1.547_2.797_0.453	1.092_1.172_2.672_0.828	0.734_1.188_2.813_0	0.720_1.188_2.531_0
0.9_0.1	1.272_1.172_2.188_2	1.265_1.172_2.188_1.313	1.257_1.172_2.188_0.563	1.255_1.172_2.188_0.359
0.9_0.5	1.258_1.172_2.25_1.813	1.230_1.172_2.391_1.109	1.192_1.172_2.375_0.375	*1.181_1.172_2.300_0.234
0.9_0.9	1.167_1.172_3.594_0.844	1.160_1.172_2.672_0.828	*1.086_1.172_2.844_0.031	*1.082_1.172_2.563_0.016

As before, the observations compared to k=5 are comparable. But there is one special exception. Due to k=7 the theoretical perfect load per node of 1.167 (7/6) is higher. But in contrast to the observations of the previous runs the perfect load cannot be achieved here. Instead, the load balancing is "close to perfect" for the best results. There the maximum computational load is 1.172. The following picture is a sample for the last best run with 3 MC. Here we can see that the load per node is in four cases 1.172 and for the others 1.156. In another not shown result (eps=0.9, zeta=0.5, k=7) five nodes have a load value of 1.172 each and only a single node has a load of 1.141. A better grouping seems not to be possible for k=7.

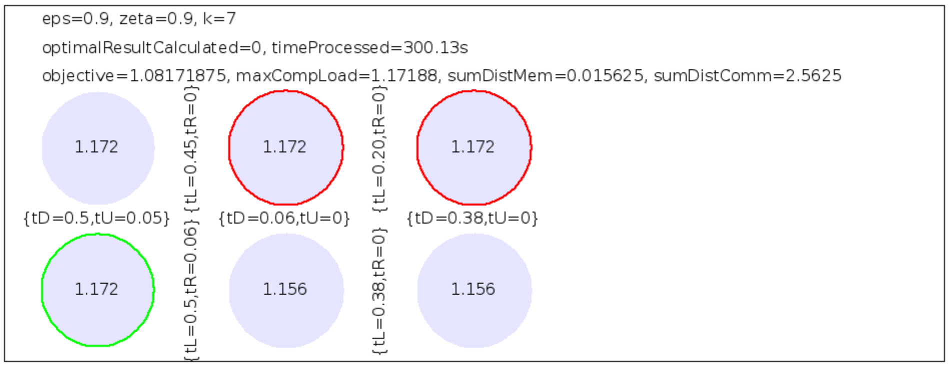
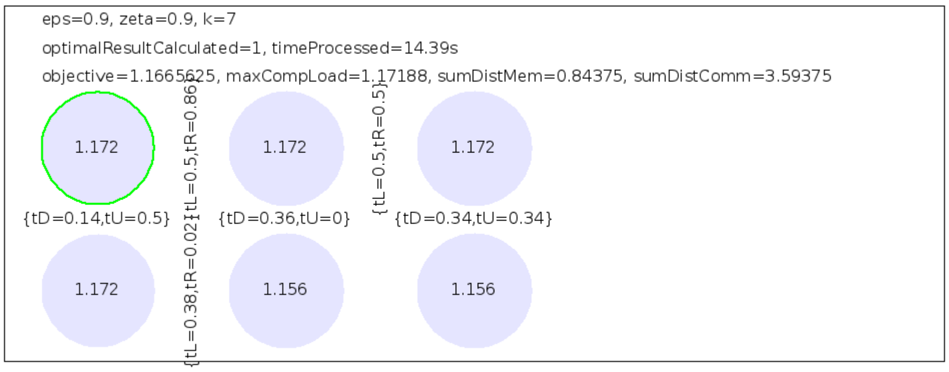


Figure 5.4.B - Close to equal task distribution.

If one now tries to find the best configuration grouped by memory controller count of all runs, the results needs to be reviewed. The outcome is summarized with the following pictures displaying the best found sub cluster. In order to get the final cluster configuration the found sub cluster configuration needs to be mirrored vertically to get a 4 rows and 3 columns sub cluster and then mirrored horizontally to get the final cluster configuration with 4 rows and 6 columns.

Best configuration for 1 MC

The Intel SCC sub cluster configuration is displayed first with the following graphic.



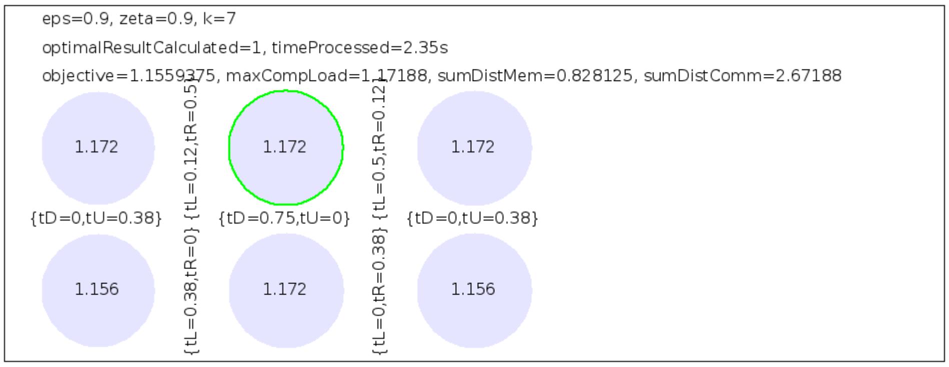
```
eps=0.9, zeta=0.9, k=7
optimalResultCalculated=1, timeProcessed=14.39s
objective=1.1665625, maxCompLoad=1.17188, sumDistMem=0.84375, sumDistComm=3.59375
```

Figure 5.4.Intel - The Intel SCC configuration

This is effectively the same configuration as the one of the full Intel SCC if just a quarter is taken into consideration. This special sub cluster can be found in the lower left quarter of the real Intel SCC.

Actually, for 1 MC there are only two possible configurations for a 2x3 sub cluster. The Intel SCC one and the best one found. This is because the other 4 are just duplicates of those two.

The next figure demonstrates the best configuration found for 1 MC. Its detailed result can be seen in the previous tables 5.4.A to 5.4.C.



```
eps=0.9, zeta=0.9, k=7
optimalResultCalculated=1, timeProcessed=2.35s
objective=1.1559375, maxCompLoad=1.17188, sumDistMem=0.828125, sumDistComm=2.67188
```

Figure 5.4.MC1win - The best configuration for one memory controller.

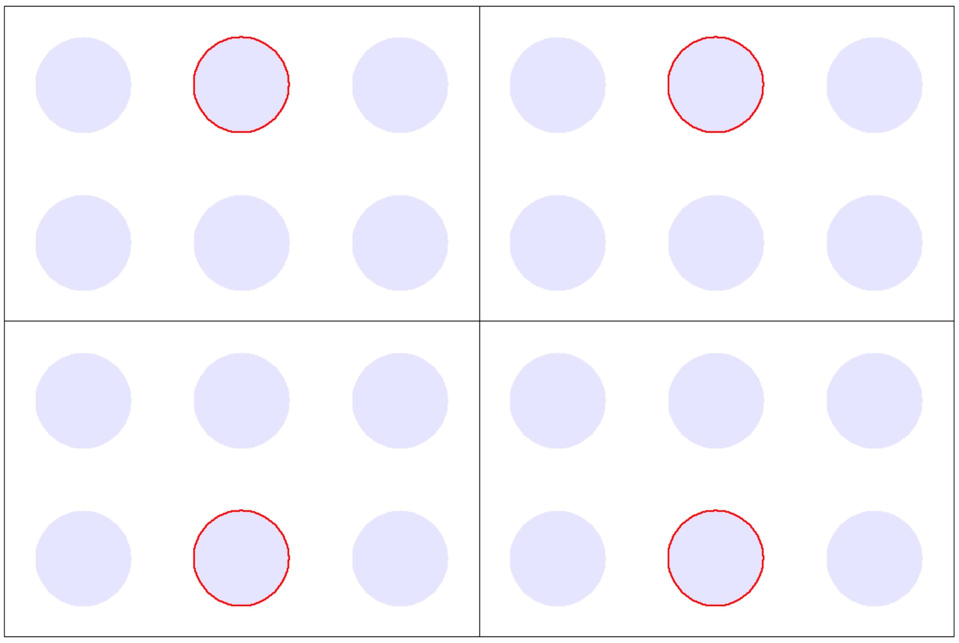


Figure 5.4.MC1winScaled - The up scaled version of the best configuration

Best configuration for 2 MC

For two memory controllers, the following table summarizes the wins of all runs for the 3 best configurations. "Win" means that the subject for the corresponding MC configuration and algorithm setup is the best. The table ignores the root co-task mapping which reduces the possible configurations down to 5.

Table 5.4.MC2win - Winning configurations for 2 MC

	Config 1	Config 2	Config 3
0.1_0.1	3	3	3
0.1_0.5	1	1	3
0.1_0.9	3	0	0
0.5_0.1	2	0	2
0.5_0.5	2	1	0
0.5_0.9	1	3	0
0.9_0.1	0	2	1
0.9_0.5	0	2	1
0.9_0.9	1	3	0
Sum	**13**	**15**	**10**

The next figures present the 3 configurations and after that the picture with the best one scaled up to 4 rows and 6 columns is shown.

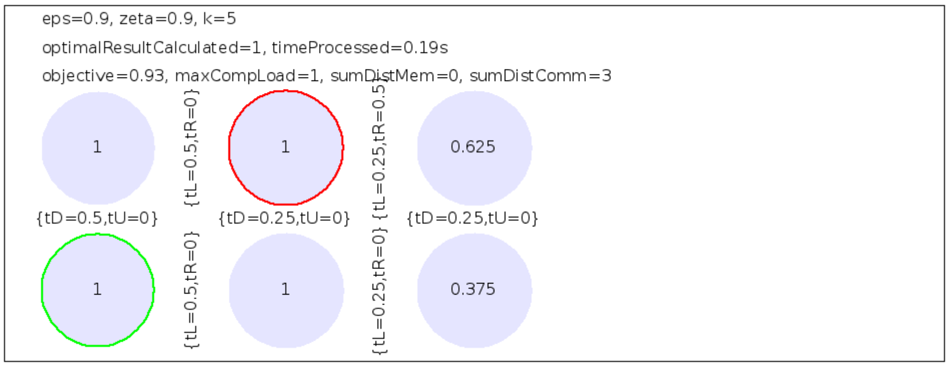

eps=0.9, zeta=0.9, k=5
optimalResultCalculated=1, timeProcessed=0.19s
objective=0.93, maxCompLoad=1, sumDistMem=0, sumDistComm=3

Figure 5.4.MC2win1 - Configuration 1

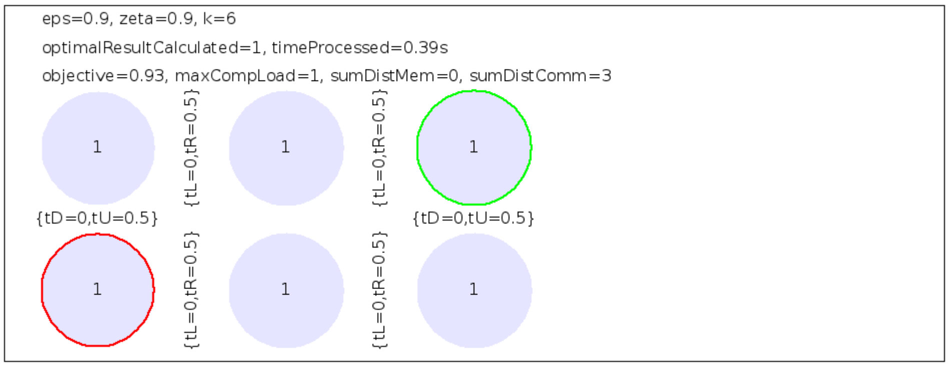

eps=0.9, zeta=0.9, k=6
optimalResultCalculated=1, timeProcessed=0.39s
objective=0.93, maxCompLoad=1, sumDistMem=0, sumDistComm=3

Figure 5.4.MC2win2 - Configuration 2

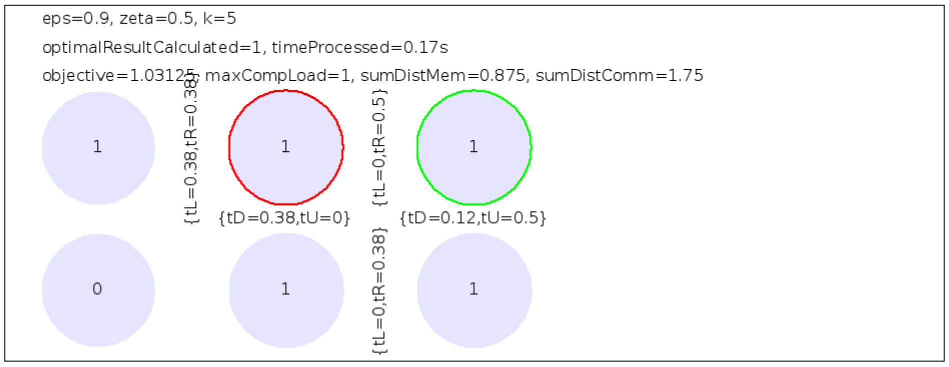

eps=0.9, zeta=0.5, k=5
optimalResultCalculated=1, timeProcessed=0.17s
objective=1.03125, maxCompLoad=1, sumDistMem=0.875, sumDistComm=1.75
1
{tL=0.38,tR=0.38}
1
{tL=0,tR=0.5}
1
{tD=0.38,tU=0}
{tD=0.12,tU=0.5}
0
1
{tL=0,tR=0.38}
1

Figure 5.4.MC2win3 - Configuration 3

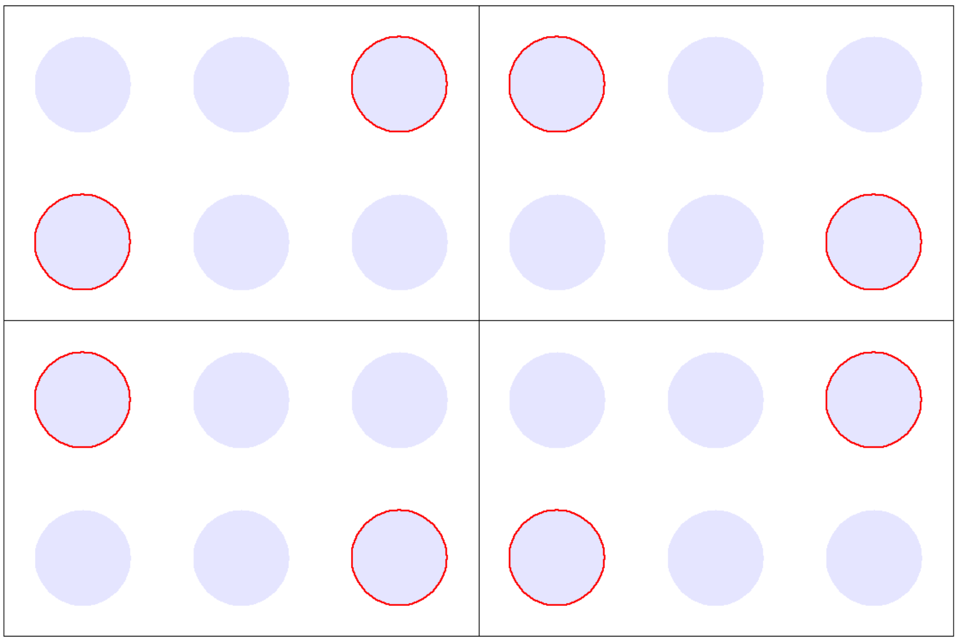


Figure 5.4.MC2win - The overall winning configuration 2 is up scaled, MC nodes are marked red.

Best configuration for 3 MC

For three memory controllers, the following table summarizes the wins of all runs for all the 4 possible configurations. The root co-task mapping is ignored here, too.

Table 5.4.MC2win - Winning configurations for 2 MC

	Config 1	**Config 2**	**Config 3**	**Config 4**
0.1_0.1	3	3	3	3
0.1_0.5	3	3	3	3
0.1_0.9	0	2	1	2
0.5_0.1	1	2	3	2
0.5_0.5	1	2	3	1
0.5_0.9	1	0	3	0
0.9_0.1	2	0	3	1
0.9_0.5	1	0	3	1
0.9_0.9	0	0	3	0
Sum	**12**	**12**	**25**	**13**

The following pictures demonstrate the layout of each sub cluster. The last one finally scales the best solution found up to the real cluster size.

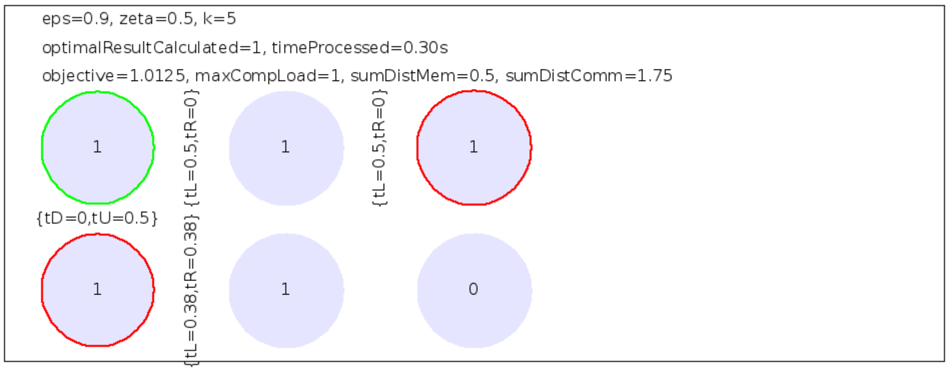

eps=0.9, zeta=0.5, k=5
optimalResultCalculated=1, timeProcessed=0.30s
objective=1.0125, maxCompLoad=1, sumDistMem=0.5, sumDistComm=1.75

Figure 5.4.MC3win1 - Configuration 1

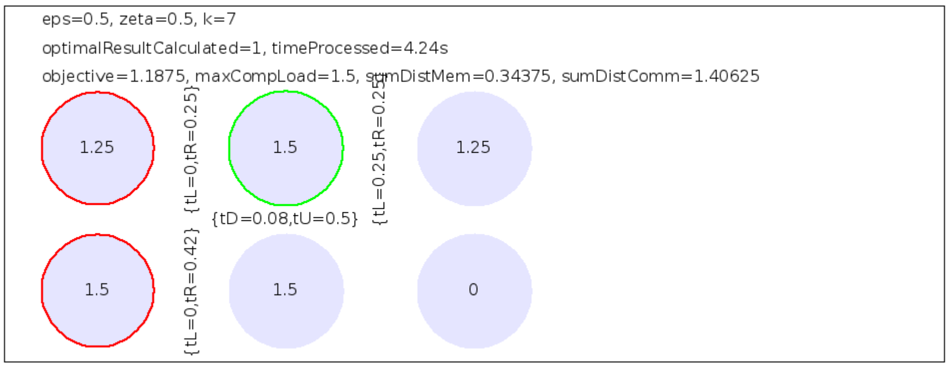

eps=0.5, zeta=0.5, k=7
optimalResultCalculated=1, timeProcessed=4.24s
objective=1.1875, maxCompLoad=1.5, sumDistMem=0.34375, sumDistComm=1.40625

Figure 5.4.MC3win2 - Configuration 2

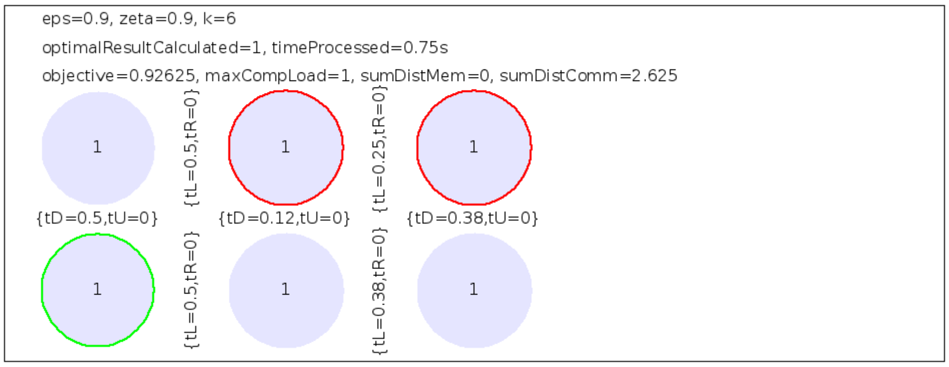


Figure 5.4.MC3win3 - Configuration 3

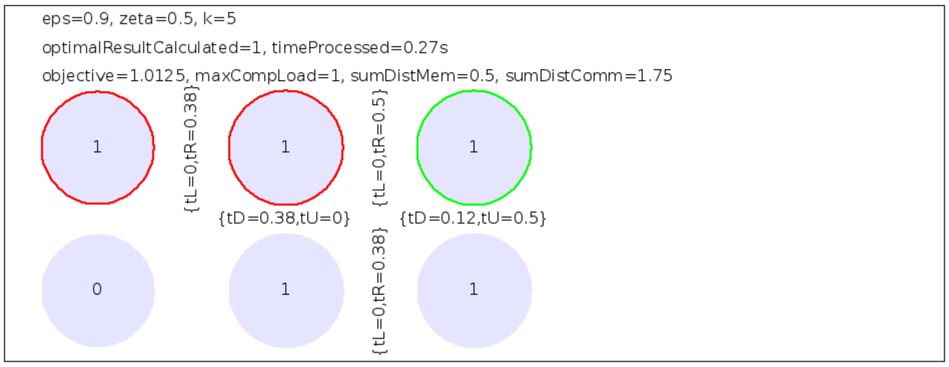

eps=0.9, zeta=0.5, k=5
optimalResultCalculated=1, timeProcessed=0.27s
objective=1.0125, maxCompLoad=1, sumDistMem=0.5, sumDistComm=1.75
1
{tL=0,tR=0.38}
1
{tL=0,tR=0.5}
1
{tD=0.38,tU=0}
{tD=0.12,tU=0.5}
0
1
{tL=0,tR=0.38}
1

Figure 5.4.MC3win4 - Configuration 4

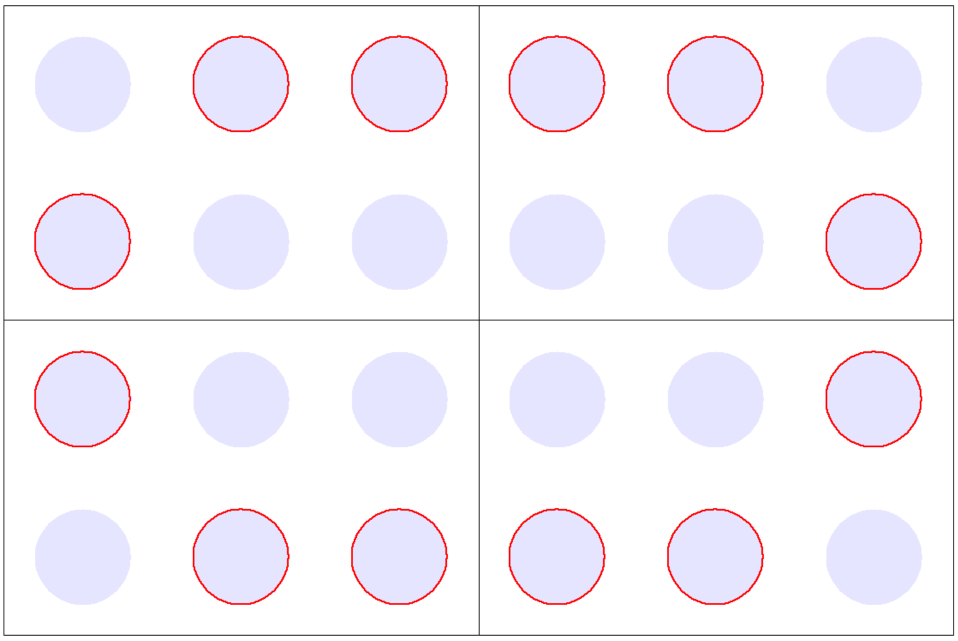


Figure 5.4.MC3win - The overall winning configuration 3 is up scaled, MC nodes are marked red.

One can see that the best solution found for 3 memory controllers is the same as for 2 memory controllers per sub cluster plus the addition of the extra MC node in the middle of the outer row.

Summary

The results found for the mirrored configuration runs are much more useful than the ones retrieved using the full model optimization. Even though they require a special adaptation of the merge sort algorithm (a separate merge step must be added), the results can be found much faster, are complete and lead for the most interesting runs where the computational cost share is higher than the communication one to a better and especially even load distribution.

5.5. Lookout into further optimizations

What is not done within this thesis is the placement of the memory controller nodes inside the cluster. This has been skipped for two reasons. First, this idea seems to be at least questionable if one wants to realize it with real hardware means. Secondly, it takes too much time as it makes only sense to optimize this for the full cluster version (4 rows, 6 columns) which requires higher k values to better reflect the reality.

Another way one could go is to change the overall structure of the cluster. One could define a single large row comparable to a BUS system, a ring of nodes, a square and so on. Also separate clusters being connected are imaginable. But as this requires too much time, it could not be realized here.

5.6. Critical view into annealing

To use annealing here was an initial idea and has not been challenged even though the execution of some runs reveal some deficits. As the neighborhood of a configuration is huge, especially with more memory controllers, it looks like the algorithm only covers a little and bounded "area" of it. This impression can be emphasized when taking a look at the spread of the results. Using a genetic approach instead or at least extending the annealing implementation by doing a far jump under certain circumstances, the area covered could be widened.

Aborting Gurobi after a given time is always risky with respect to defining the result as "best". Of course, it is not always the best. But as the results found shall just be "better" and because experiments with 4 MC and eps=0.5 and zeta=0.5 showed that after 700 seconds for most configurations the final result is given and after about 100 seconds the result is very close to it, this problem is ignorable. Chapter 6.2 introduces a different way to optimize the optimization result.

6. Investigation of further algorithms

In chapter 4 and 5 the pipelined merge sort algorithm has been used to investigate better memory controller positions. But there are a number of further communication graphs that can be mapped onto the Intel SCC that might have totally different requirements leading to completely different optimal memory controller configurations.

The following chapters take two further examples of the huge variety of parallel algorithms and investigate their perfect mapping. This is then compared against the one found for the pipelined merge sort.

6.1. Tiled-MapReduce

The Tiled-MapReduce algorithm is an extension developed by researchers of the Fudan University (China) that aims on improving the MapReduce implementation of Phoenix. Phoenix is a programming interface offering developers a framework for MapReduce applications on single hosts. The more cores the host has, the higher the performance can become. But there are limitations with respect to the scaling with higher core counts as then the memory accesses take over a high share of the processing time. To further increase the performance for such hardware, the implementation is required to be changed.

Roughly speaking the idea of the Chinese team is to minimize the data a core has to work on. This leads to better data locality and therefore more cache accesses and less external memory accesses. In details this means to subdivide the input into smaller chunks called tiles (not to mistake with the Intel SCC tiles). If one executes the process with a clever chosen tile size, the cache of the CPU is perfectly used and the speedup becomes close to optimal. The project name for a prototype implementation is Ostrich. There are more optimizations applied in Ostrich, the interested reader can find them in [6]. This is comparable to the pipelining approach where one tries to reduce the number of memory accesses to a minimum and replace them by faster means like the SCC on-chip network and the local caches.

The next figure shall summarize the high level design of the algorithm.

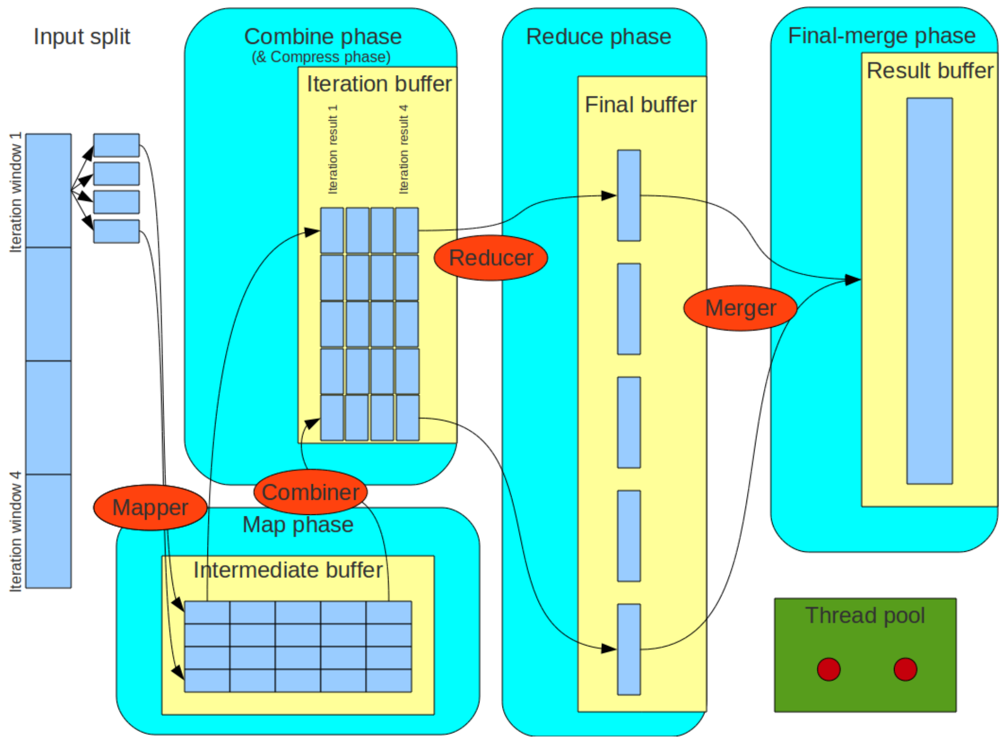


Figure 6.1 - Tiled map reduce overview - based on [6]

The mappers are processing a chunk of data (iteration window in the figure) split by a hash function in parallel. The output is placed in the intermediate buffer which is separated horizontally by the Reducer partitions and vertically by the Mapper index. Once they're done with the current iteration window, they switch their role and become Combiners. The Combiners job is it to apply the reduce function on an intermediate result partition and put the result in the iteration buffer. Having finished, the Combiners mutate back and become again mappers. Working on the same buffer with much less data decreases the likelihood of accesses to the external memory drastically. While the Map and Reduce phases are running, there are compress jobs waiting for a full iteration buffer (all slots for each horizontal partition are used up). If it is full, some threads become Compressors and compress the iteration buffer to not let the working set become too big. When the whole file has been processed successfully, the Reducers and Mergers start working and finish the MapReduce job.

6.1.1. The ILP model

The approach of avoiding the memory access as good as possible is the crucial point when trying to map an algorithm to the memory bounded Intel SCC. Like the merge sort algorithm the Tiled-MapReduce algorithm can be transformed into an ILP model describing a good mapping onto the Intel SCC. The different workers are mapped onto a core or tile of the cluster. The data is transferred by sending it over the mesh on-chip network and caching it in the local core caches instead of using a buffer in the shared main memory. This results in a communication graph. The following picture demonstrates that from a high level point of view.

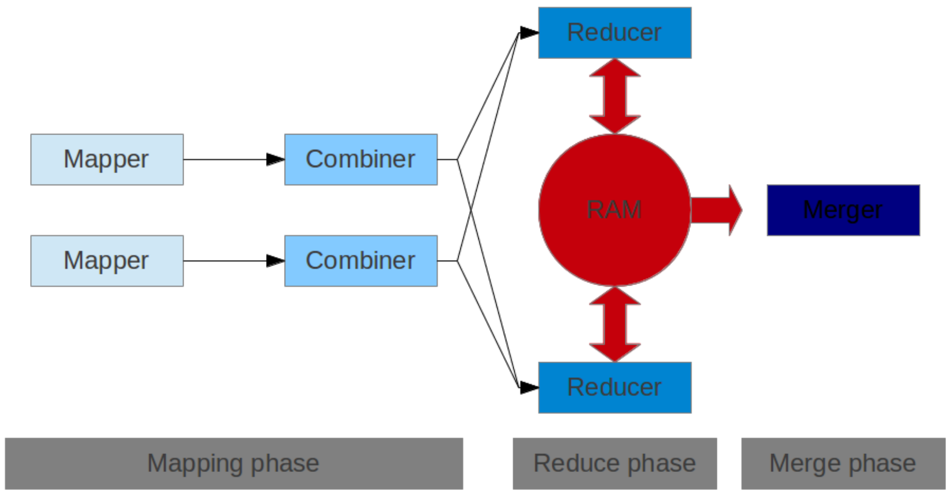


Figure 6.1.1.a - Overview of Intel SCC mapping model

The mappers are reading their data from the RAM chunk-wise. They apply a map function and send the generated data block to the next node in the graph, the combiner. The combiner stores as much data as possible locally in its caches and buffers and applies the combine method on them. Once the local buffer of the node is full, the combined data is send to the Reducer nodes partition-wise. The total data load coming into the Combiners is higher than the load send to the Reducers because the combine function has already been applied when arriving at the Reducers. The Reducer is running the reduce function again. Once the cache of the Reducer is full, it transfers the data into the RAM. When finished it pulls all data back from the external memory and forwards it to the Merger that copies the results of all Reducers together into one big data block. Of course, this requires a final reduction phase before the Merger starts working as the Reducers have processed their data block-wise. But this shall not be part of the model to keep it as simple as possible. More than that can the Merger node be skipped, too, as this node needs to wait for all data until it starts. There is no need to model this in an AMPL model. All the other nodes and their communication paths are transformable into inequalities.

The model is based on the pipelining merge sort model of the previous chapters. E.g. it uses the artifice of co-tasks for the memory controller distance calculation. Before going too much into the details of the new AMPL model, some new parameters introduced for this model shall be mentioned first:

- The Mapper produces an output data load that is higher than its incoming data load as additional information is added to each data set. This can be configured with the parameter "mapperOverhead".

- The Combiner produces less output data than it consumes because it reduces it. This is expressed with the parameter "combinerEfficiencyFactor".
- The Combiner sends an equal share (assumption) of its outgoing data to all Reducers due to a hash-based partitioning.
- The Reducers behave like the Combiner with respect to the data transformation. The factor describing the data load reduction is configured with the parameter "reducerEfficiencyFactor".

The following figures present the AMPL model which is used for the Tiled-MapReduce graph mapping onto the on-chip cluster.

```
#include "map-reduce.define"

option show_stats 1;
option omit_zero_rows 0;
option omit_zero_cols 0;
option eexit 1; // bail out on exit
option solver gurobi_ampl;
option gurobi_options GUROBI_OPTIONS;

param eps in [0, 1.0];
param zeta in [0, 1.0];

param NRows integer;
set Rows = 1..NRows;

param NCols integer;
set Cols = 1..NCols;

param memoryControllerCount integer >= 1;
set memoryControllers = 1..memoryControllerCount;
param memoryControllerMapping {group in memoryControllers, r in Rows, c in Cols};
```

Figure 6.1.1.b.1 - The AMPL Tiled-MapReduce model - part 1

The first figure is equal to the initialization steps of the chapters before and requires no further explanations.

```
param mapperCount integer;
set Mappers = 1..mapperCount;
var mapperMapping{mapper in Mappers, row in Rows, col in Cols} binary;
var mapperCoTaskMapping{mapper in Mappers, row in Rows, col in Cols} binary;

var combinerMapping{combiner in Mappers, row in Rows, col in Cols} binary;
```

Figure 6.1.1.b.2 - The AMPL Tiled-MapReduce model - part 2

As in chapter 3 and 4, the tasks must be modeled and each task must be mapped onto a tile. This is prepared here for the Mapper, its co-tasks and the Combiner tasks by defining the mapping variable sets. Like before, the tasks are numbered from 1 to n. For the Mapper tasks, n is the "mapperCount" parameter. The Mapper tasks are stored in the variable set "mapperMapping". The co-tasks are maintained here in a separate variable set called "mapperCoTaskMapping".

There are as many Combiner tasks as Mapper tasks. There is a 1 to 1 relation between those. But as the mapping is different, another variable set has been introduced to store the

Combiner mapping: "combinerMapping". As the Combiners are sending their data via the on-chip mesh network to the Reducers, there is no need for a Combiner co-task data structure.

```
param reducerCount integer;
set Reducers = 1..reducerCount;
var reducerMapping{reducer in Reducers, row in Rows, col in Cols} binary;
var reducerCoTaskMapping{reducer in Reducers, row in Rows, col in Cols} binary;
```

Figure 6.1.1.b.3 - The AMPL Tiled-MapReduce model - part 3

The Reducer tasks and their mapping is modeled here. This is as like as for the Mappers before. Like there, each Reducer has a co-task attached that models the sending of data to the external memory.

```
var  mapperToCombinerCommunicationV{mapper  in  Mappers,  r1  in  Rows,  r2  in  Rows}
binary;
var  mapperToCombinerCommunicationH{mapper  in  Mappers,  c1  in  Cols,  c2  in  Cols}
binary;

var combinerToReducerCommunicationV{combiner in Mappers, reducer in Reducers, r1 in
Rows, r2 in Rows} binary;
var combinerToReducerCommunicationH{combiner in Mappers, reducer in Reducers, c1 in
Cols, c2 in Cols} binary;

var memoryToMapperCommunicationV{mapper in Mappers, r1 in Rows, r2 in Rows} binary;
var memoryToMapperCommunicationH{mapper in Mappers, c1 in Cols, c2 in Cols} binary;

var  reducerToMemoryCommunicationV{reducer  in  Reducers,  r1  in  Rows,  r2  in  Rows}
binary;
var  reducerToMemoryCommunicationH{reducer  in  Reducers,  c1  in  Cols,  c2  in  Cols}
binary;
```

Figure 6.1.1.b.4 - The AMPL Tiled-MapReduce model - part 4

What was "yh" and "yv" before in chapter 3 and 4 becomes now "mapperToCombinerCommunicationV" and "mapperToCombinerCommunicationH" and so on. To easily map the data communication via the on-chip network and to and from the memory controllers, in total 8 variable sets have been defined that store for each sender the row and column data. The different types of senders and receivers are included in the name of those sets to make the code more readable. "V" stands as before for vertical and "H" for horizontal mesh communication.

```
param mapperLoadPerSet integer;
param combinerLoadPerSet integer;
param reducerLoadPerSet integer;

param mapperOverhead in [1, 99.0];
param combinerEfficiencyFactor in [1, 99.0];
param reducerEfficiencyFactor in [1, 99.0];
param mapperIncomingDataSize in [1, 99.0];
param combinerIncomingDataSize = mapperIncomingDataSize * mapperOverhead;
param reducerIncomingDataSize = combinerIncomingDataSize / combinerEfficiencyFactor
* mapperCount / reducerCount;
param reducerOutgoingDataSize = reducerIncomingDataSize / reducerEfficiencyFactor;
```

Figure 6.1.1.b.5 - The AMPL Tiled-MapReduce model - part 5

The already explained parameters "mapperOverhead", "combinerEfficiencyFactor" and "reducerEfficiencyFactor" are defined here. What has not been introduced yet are the following parameters:

- "mapperLoadPerSet": This defines the load per data set of the Mapper. This is used for the computational load calculations.
- "combinerLoadPerSet": This defines the load per data set of the Combiner. Same usage.
- "reducerLoadPerSet": This defines the load per data set of the Reducer. Same usage.
- "mapperIncomingDataSize": This is the relative data size the Mapper has to process. This is set to 1.

Derived from those parameters are the parameters:

- "combinerIncomingDataSize": Like the parameter "mapperIncomingDataSize" for the Mapper this describes the incoming relative data load for the Combiner. It is multiplied with "mapperOverhead" factor which expresses to what extend the data sent from the Mapper to the Combiner is enriched.
- "reducerIncomingDataSize": This parameter defines how much data is sent to each Reducer instance. The efficiency factor of the "Reducer" is taken into account. This is then multiplied by the relative share of the Mapper count (=Combiner count) to Reducer count. If there are 2 times more Reducers than Combiners, then the load sent to each Reducer from the Combiners is only half of the outgoing load of each Combiner.
- "reducerOutgoingDataSize": This expresses the outgoing relative data amount which is sent to the memory. It is based on the incoming load and divided by the Reducers efficiency.

```
var sumDistComm in interval[0.0,100000.0];
var sumDistMem  in interval[0.0,100000.0];
var maxCompLoad in interval[0.0,100000.0];

minimize obj:
    eps*maxCompLoad
    + (1-eps)*(1-zeta)*sumDistComm
    + (1-eps)*(zeta)*sumDistMem;
```

Figure 6.1.1.b.6 - The AMPL Tiled-MapReduce model - part 6

The objective function and its cost variables are the same as in the previous models.

```
subject to MapperMappingOnce {mapper in Mappers}:
    sum {row in Rows, col in Cols} mapperMapping[mapper, row, col] = 1;

subject to CombinerMappingOnce {combiner in Mappers}:
    sum {row in Rows, col in Cols} combinerMapping[combiner, row, col] = 1;

subject to ReducerMappingOnce {reducer in Reducers}:
    sum {row in Rows, col in Cols} reducerMapping[reducer, row, col] = 1;
```

Figure 6.1.1.b.7 - The AMPL Tiled-MapReduce model - part 7

The mapping of the Mapper, Combiner and Reducer tasks is also restricted. Each task must be mapped to exactly one tile.

```
subject to DefineMaxCompLoad { row in Rows, col in Cols }:
    maxCompLoad >= sum{mapper in Mappers}
                    mapperLoadPerSet    * mapperIncomingDataSize
                    * mapperMapping[mapper, row, col]
                 + sum{combiner in Mappers}
                    combinerLoadPerSet  * combinerIncomingDataSize
                    * combinerMapping[combiner, row, col]
                 + sum{reducer in Reducers}
                    reducerLoadPerSet   * reducerIncomingDataSize
                    * reducerMapping[reducer, row, col];
```

Figure 6.1.1.b.8 - The AMPL Tiled-MapReduce model - part 8

The computational load is calculated by this subject. By scaling the incoming data size of each task type with its load factor and summing this up for all task types mapped to a certain tile, the load per tile can be computed easily.

```
subject to MapperCoTaskMappingOnce {mapper in Mappers}:
    sum {row in Rows, col in Cols} mapperCoTaskMapping[mapper, row, col] = 1;
subject to ReducerCoTaskMappingOnce {reducer in Reducers}:
    sum {row in Rows, col in Cols} reducerCoTaskMapping[reducer, row, col] = 1;

subject to MapperCoTasksAreMappedToMemoryControllerNode {mapper in Mappers, row in
Rows, col in Cols}:
    mapperCoTaskMapping[mapper, row, col] <=
        sum {memGroup in memoryControllers} memoryControllerMapping[memGroup, row,
col];
subject to ReducerCoTasksAreMappedToMemoryControllerNode {reducer in Reducers, row
in Rows, col in Cols}:
    reducerCoTaskMapping[reducer, row, col] <=
        sum {memGroup in memoryControllers} memoryControllerMapping[memGroup, row,
col];
```

Figure 6.1.1.b.9 - The AMPL Tiled-MapReduce model - part 9

The co-tasks must be mapped exactly once, too. Also, a co-task is not allowed to be placed freely on the cluster. It can only be mapped to the memory controller tiles.

```
subject to VerticalCommunicationForMapperToCombiner {mapper in Mappers, r1 in Rows,
r2 in Rows}:
mapperToCombinerCommunicationV[mapper, r1, r2] >=
      sum{c1 in Cols} mapperMapping[mapper, r1, c1]
    + sum{c2 in Cols} combinerMapping[mapper, r2, c2]
    - 1;
subject to HorizontalCommunicationForMapperToCombiner {mapper in Mappers, c1 in
Cols, c2 in Cols}:
    mapperToCombinerCommunicationH[mapper, c1, c2] >=
          sum{r1 in Rows} mapperMapping[mapper, r1, c1]
        + sum{r2 in Rows} combinerMapping[mapper, r2, c2]
        - 1;

subject to VerticalCommunicationForCombinerToReducer {combiner in Mappers, reducer
in Reducers, r1 in Rows, r2 in Rows}:
    combinerToReducerCommunicationV[combiner, reducer, r1, r2] >=
        + sum{c1 in Cols} combinerMapping[combiner, r1, c1]
        + sum{c2 in Cols} reducerMapping[reducer, r2, c2]
        - 1;
subject to HorizontalCommunicationForCombinerToReducer {combiner in Mappers,
reducer in Reducers, c1 in Cols, c2 in Cols}:
    combinerToReducerCommunicationH[combiner, reducer, c1, c2] >=
        + sum{r1 in Rows} combinerMapping[combiner, r1, c1]
        + sum{r2 in Rows} reducerMapping[reducer, r2, c2]
        - 1;
```

Figure 6.1.1.b.10 - The AMPL Tiled-MapReduce model - part 10

The communication calculations are as before separated into two parts. Storing the task path data (row or column indexes) for each task as first part is done here. This is similar to the way it has been done in the chapters 3 and 4. Above one can see the node to node communication path calculations (Mapper to Combiner and Combiner to Reducer) and below one can see the same for the memory path calculations (memory to Mapper and Reducer to memory). For the latter ones it needs to be remarked that the backward direction for the memory to the Reducer node (at the end of processing all data is read and send to the Merger) has been taken into account at the end of the model.

Apart from what has already been said, another point is interesting here. For the communication between Combiner and Reducer, the communication variable set is 4 dimensional. The last 2 dimensions are as usual (contain source and destination row/column). But the first dimension is extended by the Receiver index as here we have 1 to n communication path. Each Combiner sends the data of partition A to Receiver 1, partition B to the second Receiver and so on.

```
subject to VerticalCommunicationForMemoryToMapper {mapper in Mappers, r1 in Rows,
r2 in Rows}:
    memoryToMapperCommunicationV[mapper, r1, r2] >=
          sum{c1 in Cols} mapperCoTaskMapping[mapper, r1, c1]
        + sum{c2 in Cols} mapperMapping[mapper, r2, c2]
        - 1;
subject to HorizontalCommunicationForMemoryToMapper {mapper in Mappers, c1 in Cols,
c2 in Cols}:
    memoryToMapperCommunicationH[mapper, c1, c2] >=
          sum{r1 in Rows} mapperCoTaskMapping[mapper, r1, c1]
        + sum{r2 in Rows} mapperMapping[mapper, r2, c2]
        - 1;

subject to VerticalCommunicationForReducerToMemory {reducer in Reducers, r1 in
Rows, r2 in Rows}:
    reducerToMemoryCommunicationV[reducer, r1, r2] >=
          sum{c1 in Cols} reducerCoTaskMapping[reducer, r1, c1]
        + sum{c2 in Cols} reducerMapping[reducer, r2, c2]
        - 1;
subject to HorizontalCommunicationForReducerToMemory {reducer in Reducers, c1 in
Cols, c2 in Cols}:
    reducerToMemoryCommunicationH[reducer, c1, c2] >=
          sum{r1 in Rows} reducerCoTaskMapping[reducer, r1, c1]
        + sum{r2 in Rows} reducerMapping[reducer, r2, c2]
        - 1;
```

Figure 6.1.1.b.11 - The AMPL Tiled-MapReduce model - part 11

```
subject to DefineSumDistComm:
    sumDistComm =
        sum {mapper in Mappers, r1 in Rows, r2 in Rows}
            mapperToCombinerCommunicationV[mapper, r1, r2] * abs(r2 - r1)
            * combinerIncomingDataSize
    +
        sum {mapper in Mappers, c1 in Cols, c2 in Cols}
            mapperToCombinerCommunicationH[mapper, c1, c2] * abs(c2 - c1)
            * combinerIncomingDataSize
    +
        sum {combiner in Mappers, reducer in Reducers, r1 in Rows, r2 in Rows}
            combinerToReducerCommunicationV[combiner, reducer, r1, r2]
            * abs(r2 - r1) * reducerIncomingDataSize / mapperCount
    +
        sum {combiner in Mappers, reducer in Reducers, c1 in Cols, c2 in Cols}
            combinerToReducerCommunicationH[combiner, reducer, c1, c2]
            * abs(c2 - c1) * reducerIncomingDataSize / mapperCount;
```

Figure 6.1.1.b.12 - The AMPL Tiled-MapReduce model - part 12

And as last part for the node to node distance sum calculation the contents of the communication path variable sets are used. For each link, the data size is scaled with the relative outgoing data size. This means for the Combiner to Reducer link to use the "reducerIncomingDataSize" and divide this by the "mapperCount" to get the single link weight (1 to n communication). Then this is summed over all Combiners and Reducers (for each link) to get the final weighted distance for the Combiner to Reducer part of the node to node communication. For the Mapper to Combiner data path, the derivation is not that complex as there only a simple 1 to 1 communication takes place.

```
subject to defineSumDistMem:
   sumDistMem =
      sum {mapper in Mappers, r1 in Rows, r2 in Rows}
         memoryToMapperCommunicationV[mapper,   r1,   r2]   *   abs(r2   -   r1)   *
mapperIncomingDataSize
   +
      sum {mapper in Mappers, c1 in Cols, c2 in Cols}
         memoryToMapperCommunicationH[mapper,   c1,   c2]   *   abs(c2   -   c1)   *
mapperIncomingDataSize
   +
      2 * (
         sum {reducer in Reducers, r1 in Rows, r2 in Rows}
            reducerToMemoryCommunicationV[reducer,  r1,  r2]  *  abs(r2  -  r1)  *
reducerOutgoingDataSize
         +
         sum {reducer in Reducers, c1 in Cols, c2 in Cols}
            reducerToMemoryCommunicationH[reducer,  c1,  c2]  *  abs(c2  -  c1)  *
reducerOutgoingDataSize
      );
```

Figure 6.1.1.b.13 - The AMPL Tiled-MapReduce model - part 13

This is also comparable to the chapters 3 and 4. The already stored path configuration for each path is evaluated here. For each link, the weight is taken into account ("mappingIncomingDataSize" and "reducerOutgoingDataSize"). The memory distance for the Reducers is doubled as already mentioned to express that at the end the data needs to be collected and forwarded to the Merger.

6.1.2. Results and analysis

The following pictures and tables summarize the outcome of the mirrored approach already used in the 5th chapter. Like there, one can divide the MapReduce problem into 4 sub problems and process afterwards the 4 results into one final result. To be optimized is only the sub problem. Also, the tasks of the model are mapped to tiles and not individual cores. The reason for this way of solving the optimization problem is the high complexity of a model with enough reducers, mappers and combiners to satisfy the whole cluster. This is as like as before where the high task number which is required for a sufficient cluster usage prevented realistic optimization runs.

The Tiled-MapReduce optimization runs have been done with the following algorithm setup:

- 6 mappers and combiners
- 12 reducers
- mapperOverheadFactor 1.5
- combinerEfficiencyFactor 3
- reducerEfficiencyFactor 2
- mapperLoad 1 (per data set)
- combinerLoad 3 (per data set)
- reducerLoad 4 (per data set)

This setup shall represent the fact that the Combiner and Reducer can reduce the load of the Mapper. But as reduction requires some logic the load per data set of those nodes is higher. For the Reducer it is the highest because of the additional memory operations that are required there. The actual load per node type as well as the communication load per link can be found in the following graph.

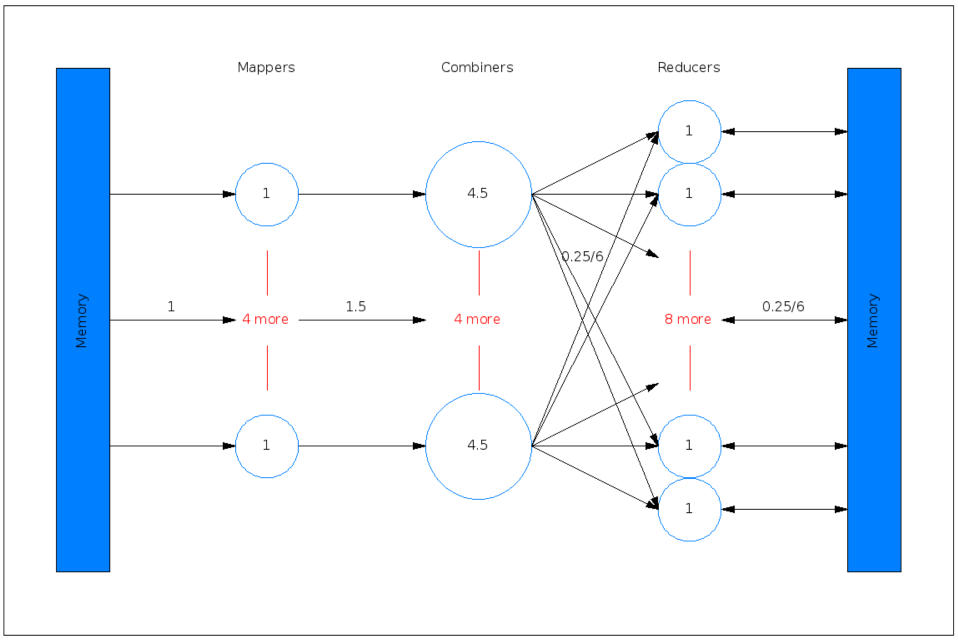


Figure 6.1.2.Task graph - The task graph that is going to be optimally mapped to the cluster

The arrows represent the data communication path, the label attached to it (or at least to one representative) gives the load being sent over that link in the given direction. If the link is bidirectional, the load of both directions in sum is given. The number in the center of each node is the load per node. How to derive those numbers can be found in the model described earlier. There are in total 24 tasks to be mapped to the tiles of the sub cluster. This makes up 4 tasks per tile of such a quadrant.

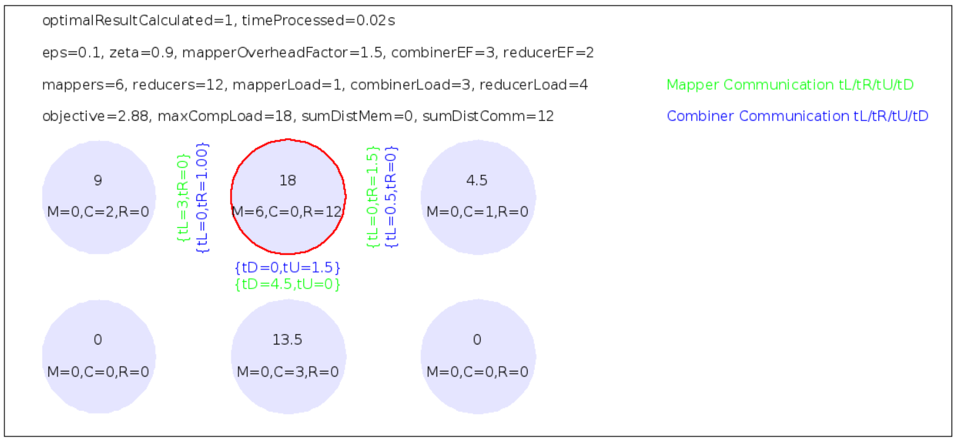
The following table summarizes the runs with different models for each eps and zeta combination. The values and row headers are to be read like introduced in chapter 5.2.

Table 6.1.2 - Results for different eps/zeta combinations for the Tiled-MapReduce model

	Intel	Best	Best with 2 MC	Best with 3 MC
0.1_0.1	3.87_22.5_1.5_4.5	3.87_22.5_1.5_4.5	3.465_22.5_1.5_0	3.465_22.5_1.5_0
	The move of the memory controller to another position has no performance advantage. Even though, the load is pretty unbalanced. Adding a second memory controller can improve the result by putting all Mappers and Reducers to the memory controller nodes hence the memory distance is 0.			
0.1_0.5	4.5_45_0_0	4.5_45_0_0	2.925_22.5_1.5_0	2.7_15_2.667_0
	Here, the load is totally unbalanced. All tasks are mapped to a single node for the 1 MC configuration as the memory distance costs share is more important. Adding memory controllers helps in reducing the unbalance. There the additional memory controllers improve the unbalanced distribution.			
0.1_0.9	2.88_18_12_0	2.88_18_12_0	2.055_9_12.833_0	1.74_15_2.667_0
	Here the load is better balanced than before with 1 MC. Due to the lower share of the node to node communication costs, the tasks are now spread to 3 respectively 4 nodes. Another interesting result is the worse balance when using 3 MC instead of 2. See also mapping figures 6.1.2.A to 6.1.2.C.			
0.5_0.1	6.3_7.5_4.167_13.5	6.15_7.5_4.167_10.5	5.925_7.5_4.167_6	5.85_7.5_4.167_4.5
	This is the first time for this model that moving the single MC to another position leads to more performance. The memory distance has been decreased. Like before, adding more MC leads to lower costs. In this case, all costs but the memory distances are equal. So here the MC position has only influence on the memory distance costs. Another interesting point being related to that: The model is perfectly balanced. See also mapping figures 6.1.2.D to 6.1.2.F.			
0.5_0.5	8.167_7.5_4.167_13.5	7.417_7.5_4.167_10.5	6.292_7.5_4.167_6	5.917_7.5_4.167_4.5
	Ignoring the total subject one can see that the results are equal to before. The subjects only differ as zeta differs.			
0.5_0.9	6.475_9_19.25_2.25	6.35_9_16.75_2.25	5.142_9_12.833_0	5.0625_9_11.25_0
	The tasks are not as balanced as before. There is one tile unused. Moving the MC to a better position results here in less node to node communication costs. Adding more MC results in a zero memory distance and in lower node to node communication costs. See also mapping figure 6.1.2.G.			

	Intel	Best	Best with 2 MC	Best with 3 MC
0.9_0.1	7.26_7.5_4.167_13.5	7.23_7.5_4.167_10.5	7.185_7.5_4.167_6	7.17_7.5_4.167_4.5
	This is as like as in the eps=0.5 and zeta in {0.1, 0.5} runs.			
0.9_0.5	7.633_7.5_4.167_13.5	7.483_7.5_4.167_10.5	7.258_7.5_4.167_6	7.183_7.5_4.167_4.5
	This is as like as before.			
0.9_0.9	7.69_7.5_13_9	7.528_7.5_10.25_7.5	7.12_7.5_10_3	7.038_7.5_8.5_2.25
	In contrast to the previous runs with eps=0.5 and zeta=0.9 this run is perfectly balanced. Compared to the other two runs with eps=0.5, the node to node distance factor is higher but the memory distance is smaller. Compared to the run with eps=0.5 and zeta=0.9, the relation is the opposite. Adding more memory controllers reduces the subject like in nearly all runs before. Here, both distance cost factors can be decreased. See also mapping figures 6.1.2.H to 6.1.2.J.			

All results are optimal as the model is small enough to be processed in less than 1000 secs. The following pictures are a well-chosen selection out of the 36 runs and its best results found.


optimalResultCalculated=1, timeProcessed=0.02s

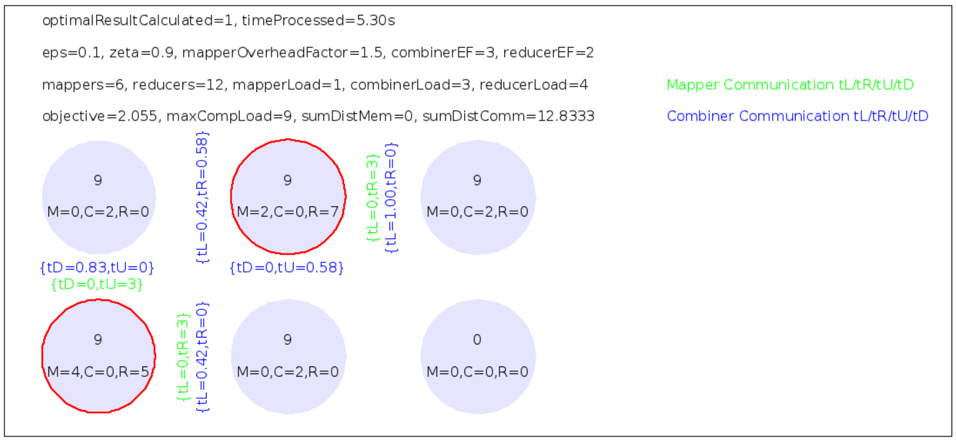
eps=0.1, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2

mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4

objective=2.88, maxCompLoad=18, sumDistMem=0, sumDistComm=12

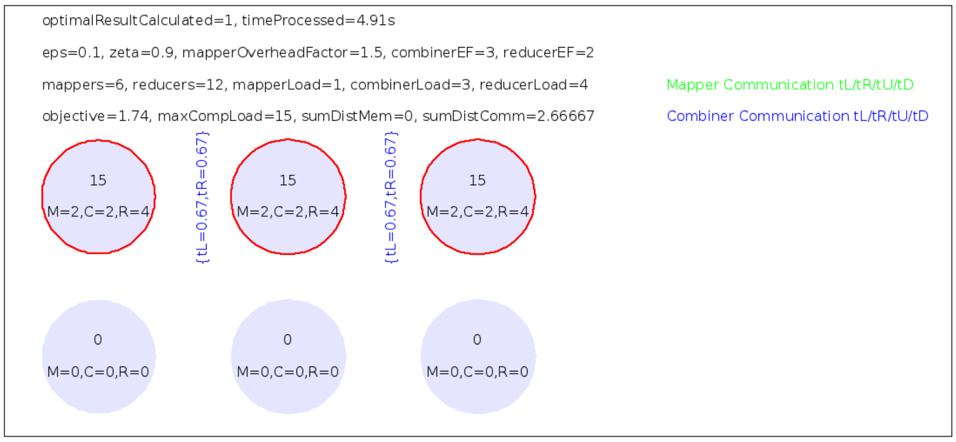
Figure 6.1.2.A - Task mapping for eps=0.1, zeta=0.9 and best configuration found

This figure presents the optimal mapping found for a given MC configuration. The number in the middle of a tile (filled circle) stands for the load of the tasks being mapped to this tile. The little text below this number summarizes the number of tasks of a certain type that are mapped to this tile. C stands for Combiners, M for Mappers and R for Reducers. The green markers between the nodes illustrate the communication between Mappers and Combiners (1 to 1). The blue markers between the nodes illustrate the communication between Combiners and Reducers (1 to n). Both colored markers do not including the memory communication costs. This is not shown. One can try to imagine the communication paths for that by following the path from the red marked MC nodes to the mappers and to/from the Reducers. The communication markers can be read as like as described in chapter 5.1.



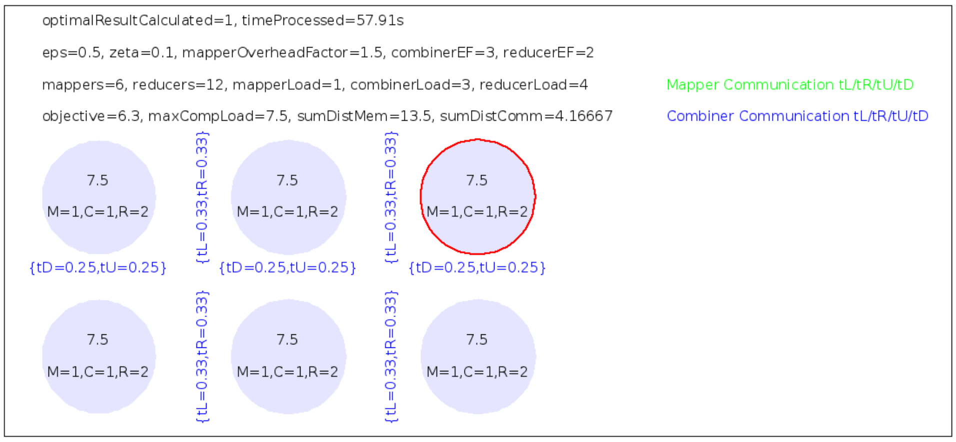
```
optimalResultCalculated=1, timeProcessed=5.30s
eps=0.1, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=2.055, maxCompLoad=9, sumDistMem=0, sumDistComm=12.8333
```

Figure 6.1.2.B - Task mapping for eps=0.1, zeta=0.9 and 2 MC



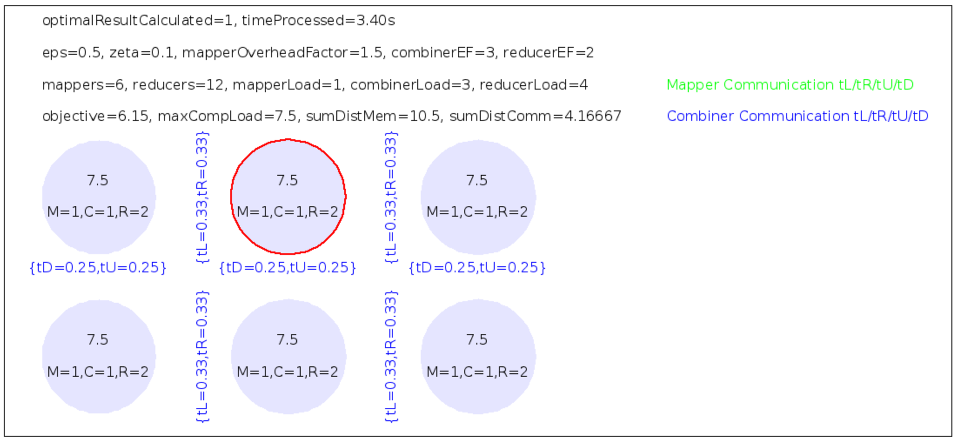
```
optimalResultCalculated=1, timeProcessed=4.91s
eps=0.1, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=1.74, maxCompLoad=15, sumDistMem=0, sumDistComm=2.66667
```

Figure 6.1.2.C - Task mapping for eps=0.1, zeta=0.9 and 3 MC



```
optimalResultCalculated=1, timeProcessed=57.91s
eps=0.5, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=6.3, maxCompLoad=7.5, sumDistMem=13.5, sumDistComm=4.16667
```

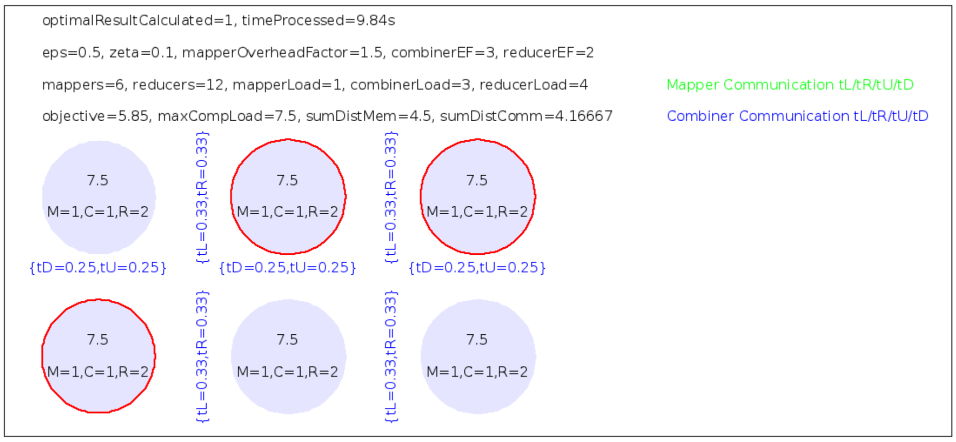
Figure 6.1.2.D - Task mapping for eps=0.5, zeta=0.1 and Intel SCC configuration



```
optimalResultCalculated=1, timeProcessed=3.40s
eps=0.5, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=6.15, maxCompLoad=7.5, sumDistMem=10.5, sumDistComm=4.16667
```

Mapper Communication tL/tR/tU/tD
Combiner Communication tL/tR/tU/tD
7.5
M=1,C=1,R=2
{tD=0.25,tU=0.25}
{tL=0.33,tR=0.33}

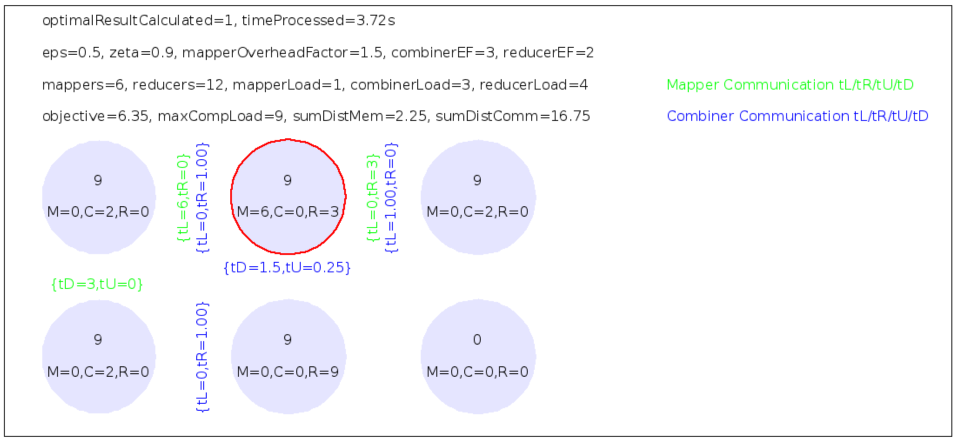
Figure 6.1.2.E - Task mapping for eps=0.5, zeta=0.1 and best configuration found



```
optimalResultCalculated=1, timeProcessed=9.84s
eps=0.5, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=5.85, maxCompLoad=7.5, sumDistMem=4.5, sumDistComm=4.16667
```

Mapper Communication tL/tR/tU/tD
Combiner Communication tL/tR/tU/tD
7.5
M=1,C=1,R=2
{tD=0.25,tU=0.25}
{tL=0.33,tR=0.33}

Figure 6.1.2.F - Task mapping for eps=0.5, zeta=0.1 and 3 MC



```
optimalResultCalculated=1, timeProcessed=3.72s
eps=0.5, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=6.35, maxCompLoad=9, sumDistMem=2.25, sumDistComm=16.75
```

Figure 6.1.2.G - Task mapping for eps=0.5, zeta=0.9 and best configuration found

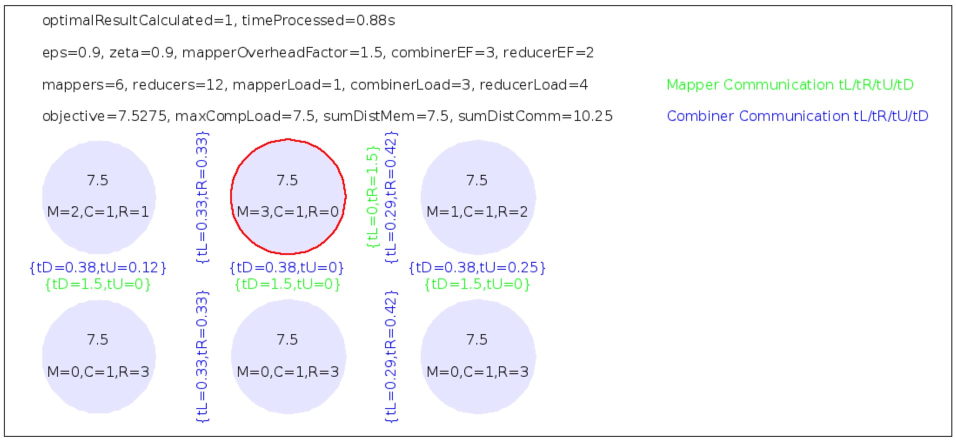

```
optimalResultCalculated=1, timeProcessed=0.88s
eps=0.9, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.5275, maxCompLoad=7.5, sumDistMem=7.5, sumDistComm=10.25
```

Figure 6.1.2.H - Task mapping for eps=0.9, zeta=0.9 and best configuration found

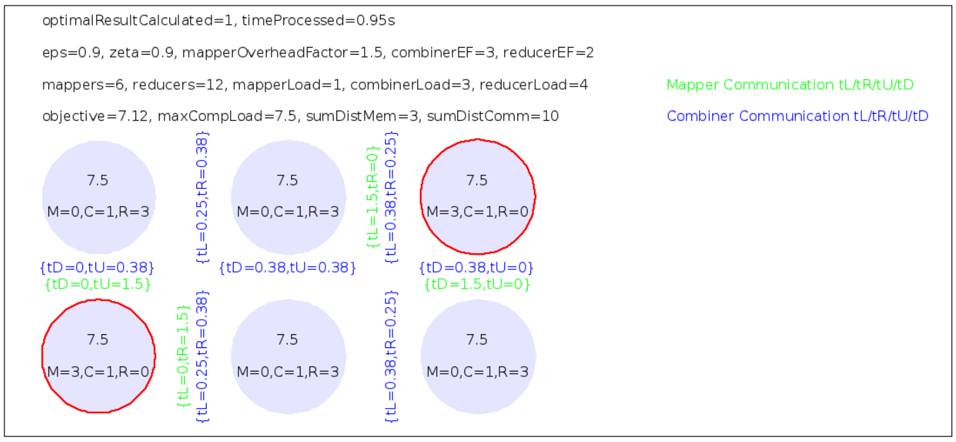

```
optimalResultCalculated=1, timeProcessed=0.95s
eps=0.9, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.12, maxCompLoad=7.5, sumDistMem=3, sumDistComm=10
```

Figure 6.1.2.I - Task mapping for eps=0.9, zeta=0.9 and 2 MC

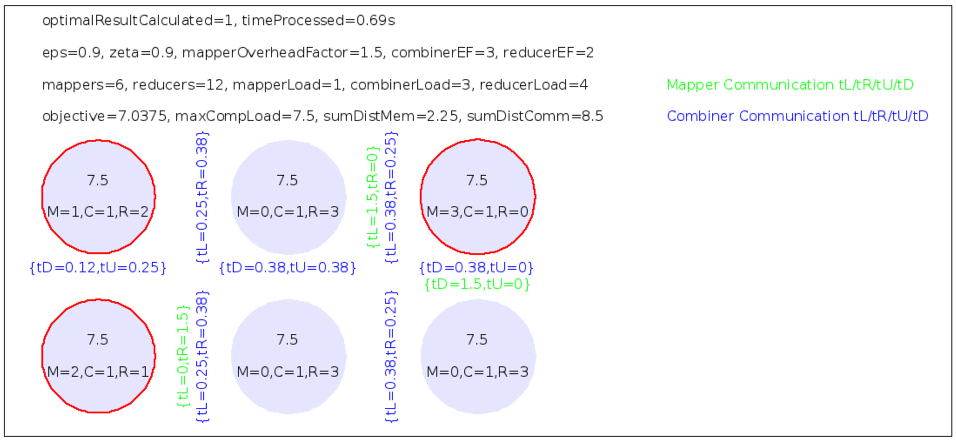

```
optimalResultCalculated=1, timeProcessed=0.69s
eps=0.9, zeta=0.9, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.0375, maxCompLoad=7.5, sumDistMem=2.25, sumDistComm=8.5
```

Figure 6.1.2.J - Task mapping for eps=0.9, zeta=0.9 and 3 MC

Best configuration for 1 MC

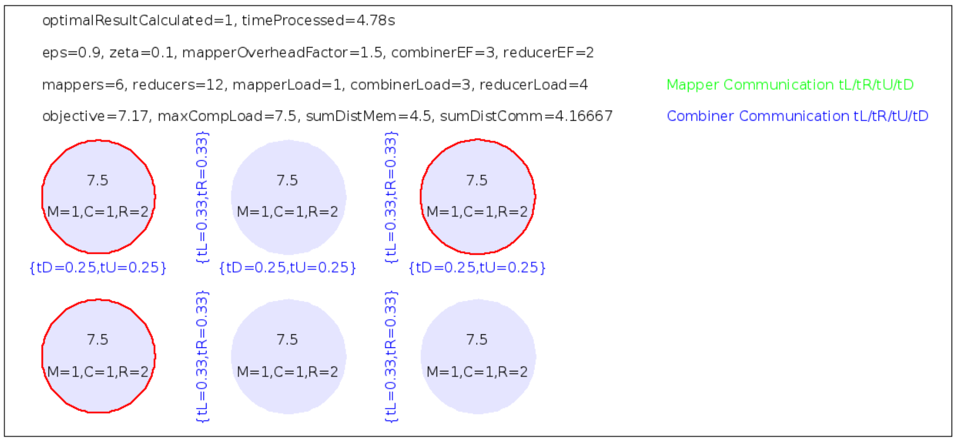
For 1 MC the superior configuration found in chapter 5.4 is also the best one here. Even though it is not always better (see runs with eps=0.1), it is at least never worse than the findings for the Intel SCC configuration.

Best configuration for 2 MC

Using the same approach of counting the "wins" as in chapter 5.4 to figure out the best MC configuration, the same configuration is found to be the winner with 5 wins over three configurations with 2 wins and one configuration without any win.

Best configuration for 3 MC

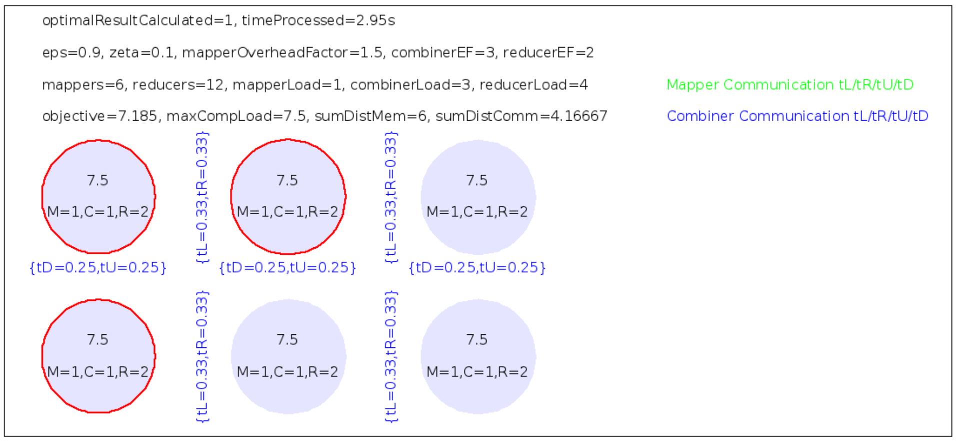
In contrast to the best configuration found for 1 and 2 MC the best configuration for 3 MC is another one than found in chapter 5.4.



```
optimalResultCalculated=1, timeProcessed=4.78s
eps=0.9, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.17, maxCompLoad=7.5, sumDistMem=4.5, sumDistComm=4.16667
```

Mapper Communication tL/tR/tU/tD
Combiner Communication tL/tR/tU/tD
7.5
M=1,C=1,R=2
{tD=0.25,tU=0.25}
{tL=0.33,tR=0.33}

Figure 6.1.2.3MC.configuration1 - This configuration had 6 wins



```
optimalResultCalculated=1, timeProcessed=2.95s
eps=0.9, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.185, maxCompLoad=7.5, sumDistMem=6, sumDistComm=4.16667
```

Mapper Communication tL/tR/tU/tD
Combiner Communication tL/tR/tU/tD
7.5
M=1,C=1,R=2
{tD=0.25,tU=0.25}
{tL=0.33,tR=0.33}

Figure 6.1.2.3MC.configuration2 - This configuration had 3 wins

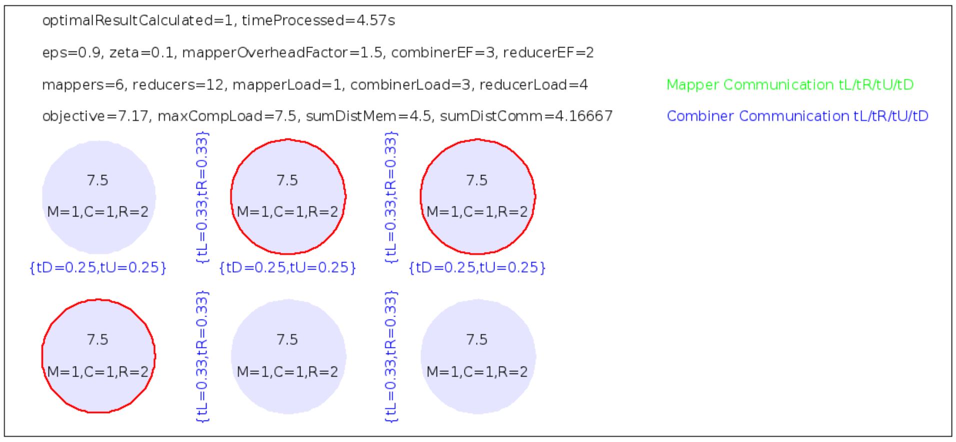

optimalResultCalculated=1, timeProcessed=4.57s
eps=0.9, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.17, maxCompLoad=7.5, sumDistMem=4.5, sumDistComm=4.16667
Mapper Communication tL/tR/tU/tD
Combiner Communication tL/tR/tU/tD
7.5
M=1,C=1,R=2
{tD=0.25,tU=0.25}
{tL=0.33,tR=0.33}

Figure 6.1.2.3MC.configuration3 - This configuration had 6 wins

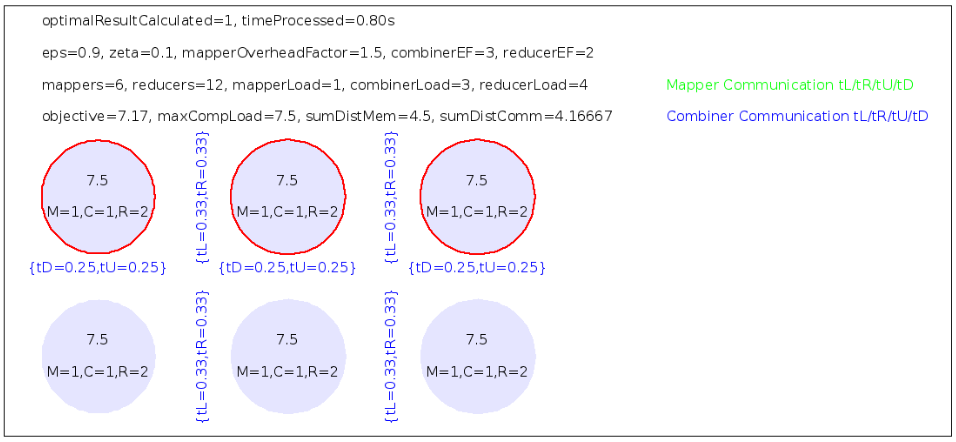

optimalResultCalculated=1, timeProcessed=0.80s
eps=0.9, zeta=0.1, mapperOverheadFactor=1.5, combinerEF=3, reducerEF=2
mappers=6, reducers=12, mapperLoad=1, combinerLoad=3, reducerLoad=4
objective=7.17, maxCompLoad=7.5, sumDistMem=4.5, sumDistComm=4.16667
Mapper Communication tL/tR/tU/tD
Combiner Communication tL/tR/tU/tD
7.5
M=1,C=1,R=2
{tD=0.25,tU=0.25}
{tL=0.33,tR=0.33}

Figure 6.1.2.3MC.configuration4 - This configuration had 7 wins

As one can see in the figure titles, configuration 4 had 7 wins and therefore is the best cluster configuration found for the given algorithm configuration. The configuration 3 was the winner of the pipelined merge sort algorithm and is the second best for Tiled-MapReduce (but this placement is shared together with configuration 1).

The following picture demonstrates the up scaled version of configuration 4.

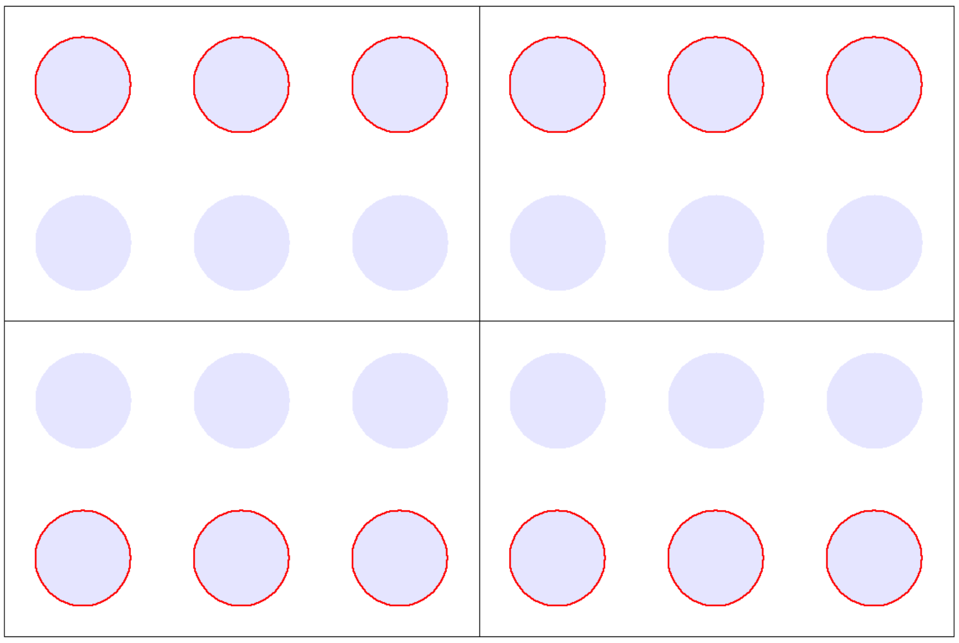


Figure 6.1.2.3MC.configuration4.upscaled - This configuration is the best for Tiled-MapReduce.

6.1.3. Non-mirrored annealing results

Unfortunately, the Tiled-MapReduce model seems to be too complex to get fully optimized for the 4x6 setup. Tests with 16 Mappers and 32 Reducers as well as 8 Mappers and 24 Reducers showed that the result becomes stable after about 4 hours. The pipelined merge sort model is stable after about 700 seconds. Even though k has been set too small to fully saturate the whole cluster, optimization runs have been done to show that an odd result is generated (see chapter 5.1 and 5.2). As this is very likely to be the same here when reducing number of Mappers and Reducers, there is no need to demonstrate it because this is less important than finding the best configuration for a realistic (meaning equally balanced) mapping.

6.2. Mesh communication

As the last graph to be mapped onto the Intel SCC die and its optimized configurations, a mesh network shall be used where all outer nodes communicate with the external memory and its neighbors and all inner nodes just with its neighbors. The load of each node within the network and also the data sent over each link including the memory link shall be equal.

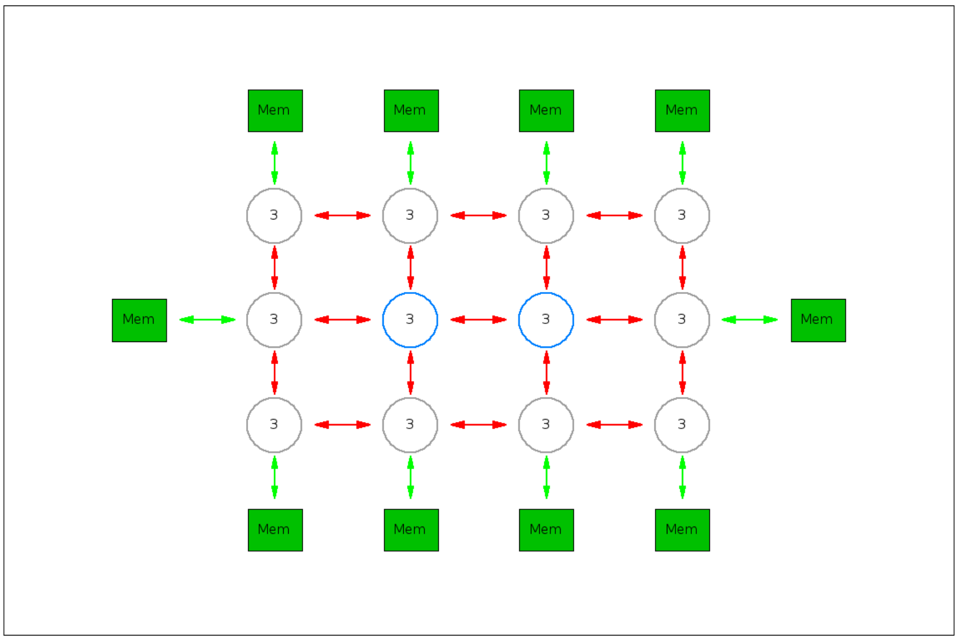


Figure 6.2.1 - Task graph for the mesh network, assuming a load of 3 per node

The Intel SCC cores are connected by a mesh network. So the natural mapping of each node of a mesh network to its corresponding counterpart of the Intel mesh network is possible if the number of rows and columns is equal. But even for this trivial case the mapping might not be optimal as the memory controller might be too far away for the majority of the nodes. E.g. putting two neighboring outer nodes of the graph onto a single core (or tile) that is close to a memory controller might be more efficient because of the lower distance to the memory controller and the omitted communication costs for data traffic between those two nodes.

In order to find the best mapping for a given setup, the following chapters describe as before the model derived from the optimization goal and the test results for it. But in contrast to before, there is no opportunity to divide the cluster into sub clusters which leads to a new optimizer optimization that is explained in the chapter 6.2.2.

6.2.1. The ILP model

The AMPL model for the mesh network is explained now in detail by separating it into several code blocks that are explained bit by bit.

```
#include "mesh.define"

option show_stats 1;
option omit_zero_rows 0;
option omit_zero_cols 0;
option eexit 1; // bail out on exit
option solver gurobi_ampl;
option gurobi_options GUROBI_OPTIONS;

param eps in [0, 1.0];
param zeta in [0, 1.0];

param NRows integer;
set Rows = 1..NRows;

param NCols integer;
set Cols = 1..NCols;

param memoryControllerCount integer >= 1;
set memoryControllers = 1..memoryControllerCount;
param memoryControllerMapping {group in memoryControllers, r in Rows, c in Cols};
```

Figure 6.2.1.1 - Code block 1 of the mesh model

This code block can also be found in the model for merge sort and Tiled-MapReduce. It initializes the cluster parameters like number of rows and columns and defines the optimization factors epsilon and zeta. Those and all following parameters are to be filled by a data file when running the model.

```
param taskRowCount integer;
param taskColCount integer;

set TaskRows = 1..taskRowCount;
set TaskCols = 1..taskColCount;
set Tasks = 0..( taskRowCount * taskColCount  -  1 );
set InnerTaskRowsPlusTop = 1..( taskRowCount - 1 );
set InnerTaskRowsPlusBottom = 2..taskRowCount;
set InnerTaskColsPlusLeft = 1..( taskColCount - 1 );
set InnerTaskColsPlusRight = 2..taskColCount;
set CoTasks = {i in Tasks:
        i mod taskColCount = 0                      // left edge
    ||  i mod taskColCount = taskColCount - 1       // right edge
    ||  int(i / taskColCount) = 0                   // upper edge
    ||  int(i / taskColCount) = taskRowCount - 1    // lower edge
};
```

Figure 6.2.1.2 - Code block 2 of the mesh model

This is the first model specific part. The mesh network to be mapped has rows and columns like the Intel SCC mesh. How many of them is hold in the parameters "taskRowCount" and "taskColCount".

Then some sets are defined to be used for later iterations. The "CoTasks" set is special as it is filled with a logic that has not been used yet. As the name implies the set contains co-tasks. The concept behind co-tasks here is equal to the merge sort co-tasks. It shall model the location of the memory controller for a certain task and is used for the memory distance calculation. As only the outer nodes of the mesh network communicate with the main memory, the conditions within the braces select only those meaning their index within in the mesh. Example: For 2 rows and 3 columns the "Tasks" set contains all the tasks from 0 to 5. The "CoTasks" set contains the same indexes as all tasks are placed at the boundary of the mesh. There is no so called inner task. If one assumes 4 rows and 3 columns then the "Tasks" array contains the numbers 0 to 11. The tasks 0 to 2 are in row 1 (left to right), the tasks 3 to 5 are in row 2 and so on. So the inner tasks are 4 and 7 as those are the only one to have a neighbor in all 4 communication directions. This means that the "CoTasks" set contains all tasks of the set "Tasks" but 4 and 7.

The sets "InnerTask*" are required for the communication distances later. E.g. "InnerTaskRowsPlusTop" contains all inner task rows plus the top one. For 4 rows this would be row 1, 2 and 3. It is later used for the communication from row 1 to 2, row 2 to 3 and row 3 to 4. Row 4 is not included as there is no fifth row where data could be sent to.

```
var taskMapping{task in Tasks, row in Rows, col in Cols} binary;
var coTaskMapping{coTask in CoTasks, row in Rows, col in Cols} binary;

subject to TaskMappingOnce {task in Tasks}:
    sum {row in Rows, col in Cols} taskMapping[task, row, col] = 1;

subject to CoTaskMappingOnce {coTask in CoTasks}:
    sum {row in Rows, col in Cols} coTaskMapping[coTask, row, col] = 1;

subject to CoTasksAreMappedToMemoryControllerNode {coTask in CoTasks, row in Rows, 
col in Cols}:
    coTaskMapping[coTask, row, col] <=
        sum {memGroup in memoryControllers} memoryControllerMapping[memGroup, row, 
col];
```

Figure 6.2.1.3 - Code block 3 of the mesh model

The "taskMapping" variable set contains for all tasks the mapping information. Example: If task 1 is mapped to tile 3,4 (row 3 and column 4 of the on-chip cluster) then the entry "taskMapping[1, 3, 4]" would be 1. All other entries for the task 1 are 0 (see also set definition "binary" - only 1 or 0 are allowed values). This and the mapping itself is enforced with the subject "TaskMappingOnce". The sum of all mappings for a task must be exactly 1.

The same constraint exists for the "coTaskMapping" variable set which contains the positions of the co-tasks. Because of the additional constraint "co-tasks must be mapped to a memory controller tile" the subject "CoTasksAreMappedToMemoryControllerNode" must be added to make sure that this is taken into account, too.

```
var sumDistComm in interval[0.0,100000.0];
var sumDistMem  in interval[0.0,100000.0];
var maxCompLoad in interval[0.0,100000.0];

minimize obj:
    eps*maxCompLoad
    + (1-eps)*(1-zeta)*sumDistComm
    + (1-eps)*(zeta)*sumDistMem;
```

Figure 6.2.1.4 - Code block 4 of AMPL model for mesh network

This is the same object function as in the previous chapters. Based on this the optimization is controlled. The goal is to find the minimum value of "obj".

```
param loadFactor in [0, 99.0];

subject to DefineMaxCompLoad { row in Rows, col in Cols }:
    maxCompLoad >=
        loadFactor * (
            sum{task in Tasks}
                taskMapping[task, row, col]  // Load is 1 per task
    );
```

Figure 6.2.1.5 - Code block 5 of the mesh model

This code block is close to equal to the previous definitions of the maximum computational load. The difference is the additional parameter "loadFactor". This parameter controls the weight of the computational load over the distance costs, too. It complements the eps parameter. This has been introduced to be able to use the same eps and zeta values as before. The load factor is set to the value resulting in a comparable load balancing of the mesh tasks on the cluster tiles for a given epsilon value. The factor 10 has been found to realize this requirement pretty well (see chapter 6.2.3).

```
var taskToTaskCommunicationVCUM{task in Tasks, r1 in Rows, r2 in Rows} binary;
var taskToTaskCommunicationVCDM{task in Tasks, r1 in Rows, r2 in Rows} binary;
var taskToTaskCommunicationVCLM{task in Tasks, r1 in Rows, r2 in Rows} binary;
var taskToTaskCommunicationVCRM{task in Tasks, r1 in Rows, r2 in Rows} binary;

subject to VerticalClusterUpCommunication {taskRow in InnerTaskRowsPlusBottom,
taskCol in TaskCols, r1 in Rows, r2 in Rows}:
    taskToTaskCommunicationVCUM[ (taskRow-1) * taskColCount + taskCol - 1 , r1,
r2] >=
          sum{c1 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
, r1, c1]
        + sum{c2 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
- taskColCount , r2, c2]
        - 1;
subject to VerticalClusterDownCommunication {taskRow in InnerTaskRowsPlusTop,
taskCol in TaskCols, r1 in Rows, r2 in Rows}:
    taskToTaskCommunicationVCDM[ (taskRow-1) * taskColCount + taskCol - 1 , r1,
r2] >=
          sum{c1 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
, r1, c1]
        + sum{c2 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
+ taskColCount , r2, c2]
        - 1;
subject to VerticalClusterLeftCommunication {taskRow in TaskRows, taskCol in
InnerTaskColsPlusRight, r1 in Rows, r2 in Rows}:
    taskToTaskCommunicationVCLM[ (taskRow-1) * taskColCount + taskCol - 1 , r1,
r2] >=
          sum{c1 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
, r1, c1]
        + sum{c2 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
- 1 , r2, c2]
        - 1;
subject to VerticalClusterRightCommunication {taskRow in TaskRows, taskCol in
InnerTaskColsPlusLeft, r1 in Rows, r2 in Rows}:
    taskToTaskCommunicationVCRM[ (taskRow-1) * taskColCount + taskCol - 1 , r1,
r2] >=
          sum{c1 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
, r1, c1]
        + sum{c2 in Cols} taskMapping[ (taskRow-1) * taskColCount + taskCol - 1
+ 1 , r2, c2]
        - 1;
```

Figure 6.2.1.6 - Code block 6 of the mesh model

The derivation of the node to node communication distance is done in two parts. First, for each task the information whether it communicates from a cluster row to another or from a cluster column to another is stored in the variable sets "taskToTaskCommunication*". Second, this data is used and aggregated. The latter one is described later in figure 6.2.1.9. The former one can be seen in the figures above and below. They are split by cluster communication directions: vertical (above code block) and horizontal (below code block). This means that up and down as well as right and left are handled together. For each mesh direction, 4 variable sets are defined, one set for each possible mesh network communication direction (up, down, left and right). Combined with the horizontal and vertical cluster directions, this makes up exactly 8 variable sets ("taskToTaskCommunication*").

Example: If there are 3 rows and 3 columns given, then row 1 contains the tasks 0 to 2, row 2 contains the tasks 3 to 5 and row 3 contains the tasks 6 to 8. Task 0 is sending data to task 3. Task 3 is sending some data back to task 0. Task 0 is also sending data to task 1 (and vice versa). For task 0 there are only those 2 out of the possible 4 mesh communication directions as it has only 2 neighbors. The used directions shall be named "Downward" and "Rightward" (from the perspective of the sender). The subjects "VerticalClusterDownCommunication" and "HorizontalClusterDownCommunication" are about saving all the required communication data (not the costs, but the flag) for the mesh direction "Downward". Depending on where the tasks 0 and 1 are mapped to, the first or second or both mentioned subjects are taken. If task 0 is mapped to tile 1,1 (top left) and task 3 is mapped to tile 2,2 then the "Downward" mesh direction is mapped to the communication variable sets by both subjects "VerticalClusterDownCommunication" and "HorizontalClusterDownCommunication". If the tasks are mapped the opposite way the subjects are still the same as the cluster communication directions upward and downward are summarized as vertical (first name part of the subjects) and leftward and rightward are expressed with horizontal.

For task 0 (source task) sending data to task 3 (destination task) the variable set filling can be explained as following:

- subject "VerticalClusterDownCommunication"
    - It iterates over all tasks. Those are the source tasks. For each such task, it checks for any combination of a source and destination row of the cluster, where the source mesh task is mapped to the source cluster row and the below (second keyword "Down") neighbor of the source mesh task is mapped to the destination cluster row. If so, the value stored in the variable set "taskToTaskCommunicationVCDM" (VCDM stands for "VerticalClusterDownMesh") must be greater or equal to 1. If the destination mesh task is not mapped to the destination cluster row or the source mesh task is not mapped to the source cluster row, the value must be greater or equal to 0. If both are not mapped to the rows, then the value must be greater or equal to -1.
    - The value 1 at "taskToTaskCommunicationVCDM[0, 1, 2]" means: Task 0 is sending data from cluster row 1 to cluster row 2. This also means that task 0 must be mapped to cluster row 1. As the "D" (meaning down) key character directly addresses the receiver (task 3) this can be extended to also show that task 3 must be mapped to cluster row 2.
    - As the data stored in the set "taskToTaskCommunicationVCDM" is of type binary and the overall model objective is to minimize the result, the case "greater or equal to 1" leads to "equal to 1", "greater or equal to 0" leads to "equal to 0" and the case "greater or equal to -1" leads to "equal to 0". Because of that an appropriate constraint/subject with an upper bound has not been added.
    - The source tasks iterated over are all calculated from the mesh row set "InnerTaskRowsPlusTop". This includes all mesh rows which can have a task directly below it. The calculation itself is trivial. As parameters to the subject,

a mesh row and column number are given from which the task number can be calculated.
    - The destination task is derived from the source task based on the mesh communication direction. In this case (down), one needs to add the column count (here 3).
    - For task 0 the outcome of taking this subject into account is: "taskToTaskCommunicationVCDM[0, 1, 2] = 1", all other row combinations are zero or ignorable. Ignorable means that in case the rows are equal, the result is 1 but this is ignored by later subjects (which is the reason why it is 1 and not the lowest binary possible value 0).
- subject "HorizontalClusterDownCommunication"
    - It iterates over all tasks. Those are the source tasks. For each such task, it checks for any combination of a source and destination column of the cluster, where the source mesh task is mapped to the source cluster column and the below (keyword "Down") neighbor of the source mesh task is mapped to the destination cluster column. If so, the value stored in the variable set "taskToTaskCommunicationHCDM" (HCDM stands for "HorizontalClusterDownMesh") must be greater or equal to 1. The rest is as for subject "VerticalClusterDownCommunication".
    - The value 1 at "taskToTaskCommunicationHCDM[0, 1, 2]" means: Task 0 is sending data from cluster column 1 to cluster column 2. This also means that task 0 must be mapped to cluster column 1. As the "D" key character directly addresses the receiver (task 3) this can be extended to also show that task 3 must be mapped to cluster column 2.
    - The other things are as above.

```
var taskToTaskCommunicationHCUM{task in Tasks, c1 in Cols, c2 in Cols} binary;
var taskToTaskCommunicationHCDM{task in Tasks, c1 in Cols, c2 in Cols} binary;
var taskToTaskCommunicationHCLM{task in Tasks, c1 in Cols, c2 in Cols} binary;
var taskToTaskCommunicationHCRM{task in Tasks, c1 in Cols, c2 in Cols} binary;

subject to HorizontalClusterUpCommunication {taskRow in InnerTaskRowsPlusBottom,
taskCol in TaskCols, c1 in Cols, c2 in Cols}:
    taskToTaskCommunicationHCUM[  (taskRow-1) * taskColCount  +  taskCol - 1  , c1,
c2] >=
          sum{r1 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
, r1, c1]
        + sum{r2 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
- taskColCount  , r2, c2]
        - 1;
subject to HorizontalClusterDownCommunication {taskRow in InnerTaskRowsPlusTop,
taskCol in TaskCols, c1 in Cols, c2 in Cols}:
    taskToTaskCommunicationHCDM[  (taskRow-1) * taskColCount  +  taskCol - 1  , c1,
c2] >=
          sum{r1 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
, r1, c1]
        + sum{r2 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
+ taskColCount  , r2, c2]
        - 1;
subject to HorizontalClusterLeftCommunication {taskRow in TaskRows, taskCol in
InnerTaskColsPlusRight, c1 in Cols, c2 in Cols}:
    taskToTaskCommunicationHCLM[  (taskRow-1) * taskColCount  +  taskCol - 1  , c1,
c2] >=
          sum{r1 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
, r1, c1]
        + sum{r2 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
- 1  , r2, c2]
        - 1;
subject to HorizontalClusterRightCommunication {taskRow in TaskRows, taskCol in
InnerTaskColsPlusLeft, c1 in Cols, c2 in Cols}:
    taskToTaskCommunicationHCRM[  (taskRow-1) * taskColCount  +  taskCol - 1  , c1,
c2] >=
          sum{r1 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
, r1, c1]
        + sum{r2 in Rows} taskMapping[  (taskRow-1) * taskColCount  +  taskCol - 1
+ 1  , r2, c2]
        - 1;
```
Figure 6.2.1.7 - Code block 7 of the mesh model

```
var  edgeTaskToMemoryCommunicationV{coTask  in  CoTasks,  r1  in  Rows,  r2  in  Rows}
binary;
var  edgeTaskToMemoryCommunicationH{coTask  in  CoTasks,  c1  in  Cols,  c2  in  Cols}
binary;

subject to VerticalCommunicationForEdgeTaskToMemory {coTask in CoTasks, r1 in Rows,
r2 in Rows}:
    edgeTaskToMemoryCommunicationV[coTask, r1, r2] >=
          sum{c1 in Cols} coTaskMapping[coTask, r1, c1]
        + sum{c2 in Cols} taskMapping[coTask, r2, c2]
        - 1;
subject  to  HorizontalCommunicationForEdgeTaskToMemory  {coTask  in  CoTasks,  c1  in
Cols, c2 in Cols}:
    edgeTaskToMemoryCommunicationH[coTask, c1, c2] >=
          sum{r1 in Rows} coTaskMapping[coTask, r1, c1]
        + sum{r2 in Rows} taskMapping[coTask, r2, c2]
        - 1;
```

Figure 6.2.1.8 - Code block 8 of the mesh model

The above figure is comparable to figures 6.2.1.6 and 6.2.1.7. This time it is about the memory distance calculation. This eases the readability as there is only one neighbor per task. A co-task can communicate only with exactly one outer task and vice versa. The latter one is not modeled here but in the last figure 6.2.1.10 by simply doubling the distance. As the "CoTask" array contains a sub set of the "Tasks" array, the entries can be used to access the "taskMapping" variable set as well.

```
subject to DefineSumDistComm:
    sumDistComm =
        sum {task in Tasks, r1 in Rows, r2 in Rows} (
              taskToTaskCommunicationVCUM[task, r1, r2] * abs(r2 - r1)
            + taskToTaskCommunicationVCDM[task, r1, r2] * abs(r2 - r1)
            + taskToTaskCommunicationVCLM[task, r1, r2] * abs(r2 - r1)
            + taskToTaskCommunicationVCRM[task, r1, r2] * abs(r2 - r1)
        )
    +
        sum {task in Tasks, c1 in Cols, c2 in Cols} (
              taskToTaskCommunicationHCUM[task, c1, c2] * abs(c2 - c1)
            + taskToTaskCommunicationHCDM[task, c1, c2] * abs(c2 - c1)
            + taskToTaskCommunicationHCLM[task, c1, c2] * abs(c2 - c1)
            + taskToTaskCommunicationHCRM[task, c1, c2] * abs(c2 - c1)
        );
```

Figure 6.2.1.9 - Code block 9 of the mesh model

This subject finally defines how to calculate the distance for the node to node communication. This is the second step after having found and encoded to what cluster row or column each mesh task is sending data to. The variable sets already explained earlier expressing this mapping are now read and interpreted. The distance calculation is based on the X-Y-routing of the cluster. This means that one can add the difference between two cluster rows and columns where a neighboring task pair is mapped to in order to get the distance between the nodes. For the above example (3x3 mesh, task 0 is sending data to task 3, task 0 is mapped to cluster tile 1,1 and task 3 to 2,2) the relevant parts of the arrays are:

- 0 to 3 (mesh downward):
    - taskToTaskCommunicationVCDM[0, 1, 2]
    - taskToTaskCommunicationHCDM[0, 1, 2]
- 3 to 1 (mesh upward):
    - taskToTaskCommunicationVCUM[3, 2, 1]
    - taskToTaskCommunicationHCUM[3, 2, 1]

Each part is multiplied with 1 (row and column distance is always 1 here) and summed up to the value 4. This means that for the communication between tasks 0 and 3 the share of the complete node to node communication load is 4.

```
subject to defineSumDistMem:
   sumDistMem = 2 * (
      sum {coTask in CoTasks, r1 in Rows, r2 in Rows}
         edgeTaskToMemoryCommunicationV[coTask, r1, r2] * abs(r2 - r1)
      + sum {coTask in CoTasks, c1 in Cols, c2 in Cols}
         edgeTaskToMemoryCommunicationH[coTask, c1, c2] * abs(c2 - c1)
   );
```

Figure 6.2.10 - Code block 10 of the mesh model

This is basically the same as for the node to node distance calculation. Here, the cluster row and column number differences for the link pair "co-task to task" are added if there is an existing link. The sum is multiplied with 2 as the communication is bidirectional. Without this the communication direction "task to co-task" would not be taken into account (see also figure 6.2.1.8).

6.2.2. Parameter tuning

Initial execution tests with the model revealed very high execution times without reaching optimality. As like as for the Tiled-MapReduce model, the result was not stable meaning it changed continuously, even after hundreds of seconds. This chapter demonstrates one possible way to faster and more stable solutions that are as close as possible to the real objective.

The following tables show the so called "incumbent" over the time. The incumbent is the currently best known objective while running the optimization with Gurobi. The "gap" reflects the uncertainty - the lower the better. It is calculated from the difference of the incumbent and the currently known best lower bound of the objective. Roughly speaking once the bound hits the incumbent, the gap becomes zero and the result optimal. The content is taken from model executions for the Intel SCC configuration with the load factors 1 and 3, eps=0.9 and zeta=0.5.

Table 6.2.2.F=1.a - Run details: factor 1, Intel SCC, 4 task rows, 6 task columns, time limit 2 hours

Time	Incumbent	Gap
17s	7.8	67.9%
18s	7.7	67.4%
19s	7.5	64.0%
22s	7.4	63.5%
40s	7.2	61.9%
588s	7.1	54.1%
3988s	7.0	44.2%

Table 6.2.2.F=3.a - Run details: factor 3, 4 task rows, 6 task columns, time limit 10 hours

Time	Incumbent	Gap
3s	15.6	72.4%
75s	12.1	58.6%
139s	12.0	57.3%
164s	11.6	55.9%
812s	10.9	51.8%
4179s	10.8	46.1%
20594s	10.7	42.6%
25501s	10.3	39.9%
25726s	9.9	37.5%
26511s	8.9	30.4%
27434s	8.5	27.0%

What one can see especially in the second table is the large time to get to a good result (meaning with a low gap value). Whether the results 7.0 (factor 1) or 8.5 (factor 3) are really optimal is actually unknown. Having a look into the task graph for the factor 3 run which maps the mesh naturally to the cluster strengthens the impression of an optimal result as the intermediate subjects map several nodes onto one tile which is at least not the expected result for eps=0.9 runs with a higher load factor.

The execution times and intermediate results have been watched for the previous models before, too. There, the circumvention is to split the graph into smaller sub graphs to generate pretty small gaps after an acceptable time (or in most cases optimality). Here, a graph split cannot be done because the graph is given without a problem behind it. So without knowing whether the problem can be split, the graph cannot be split.

To be able to get a result in acceptable time which is the precondition for annealing runs, the way forward is to use the latest Gurobi version 5.5.0. This new version introduces automatic parameter tuning to get the objective of a model even faster once the tune result is known.

The parameters have a strong influence on the linear programming algorithms used by Gurobi. So selecting them carefully might have a very positive effect on the execution time.

The automatic tuning mode works as following:

- A set of parameters is chosen and tried 2 times (default) with different seeds (start values for the randomness).
- The average gap of the trials (1 - lowest found border / incumbent) is stored.
- This is repeated for various parameter combinations until the tuning time limit is reached.
- The best parameter set is returned ["best" means the set with the lowest average gap].
- The parameters and combinations selected for testing seems to depend on:
    - intermediate results
    - time limit (time for each seed run)
    - tuning time limit (time for the complete tuning run)

For the factors 1 and 3 the tuning mode is used. As the chosen and tested parameter combinations vary especially when using different seed run times, several seed run times must be used. The best parameter configurations are summarized in the following figures and notes.

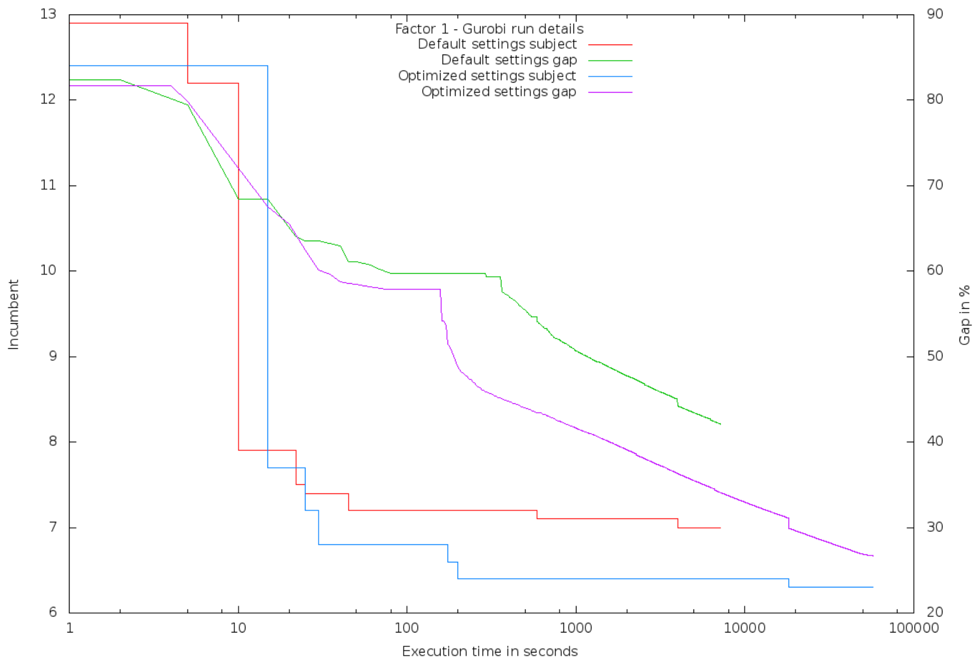


Figure 6.2.2.F=1.b - Compare optimized (16 hours run) vs. default settings, factor 1

The data shows a significant improvement for the tuned parameter set "GomoryPasses=0, Prepasses=3" over the default Gurobi setup "GomoryPasses=-1 (unlimited), Prepasses=-1 (automatic choice)". Even after 16 hours the subject is still 6.3 (found already after 18200 secs) with a gap of 26.6%. It sounds likely that this result is the optimal one as a gap of 30% or smaller seems be a good indicator for the mesh model because several executions up to now showed that. But unfortunately, the best known value is 6.2 (see later). Also with respect to the gap the improvement is remarkable. But the more interesting thing one can see for the gap value is that by logarithmically scaling the x-axis the gap "function" seems to fall linearly after about 300 secs. This means that it can take very long to bring the gap down to 0 if no further improvements of the subject are found.

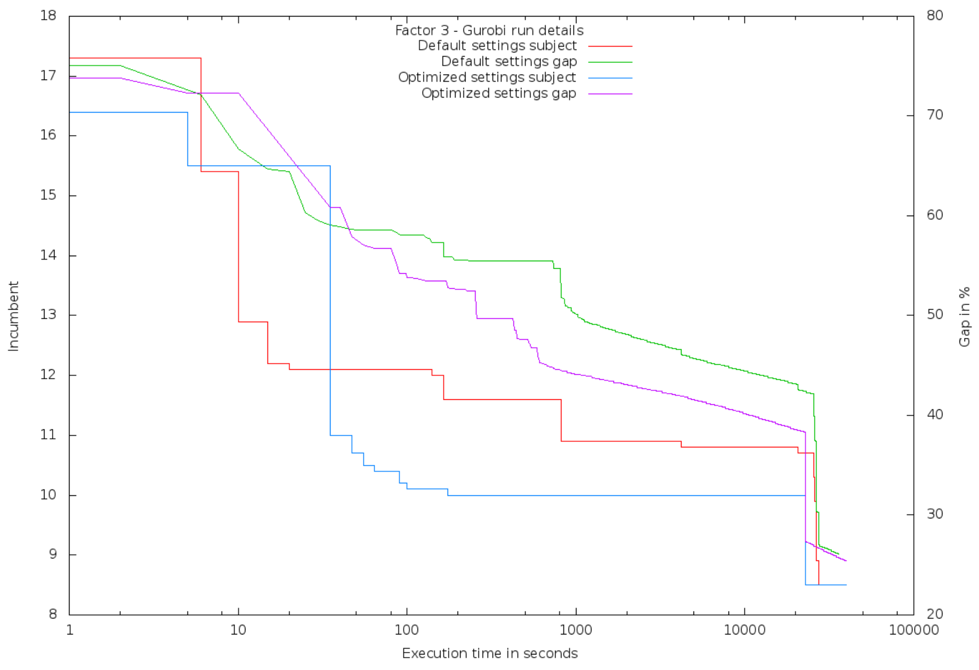


Figure 6.2.2.F=3.b - Compare optimized (11.1 hours run) vs. default settings, factor 3

The observation for the "factor=3" tuning is equal. But the performance improvement is not as good as for factor 1. The best automatically found parameter set is different: "Heuristics=0.5, GomoryPasses=1". The default value for Heuristics is 0.05 and means that MIP heuristics (see Gurobi documentation) shall be used in 5% of the processing time. The best known result 8.5 (using several optimization and normal runs) is found after about 22700 secs. This is faster than with default settings but still much too slow.

One possible next step forward is to combine the best parameters from both tuning runs and investigate the results for factor 1 and 3.

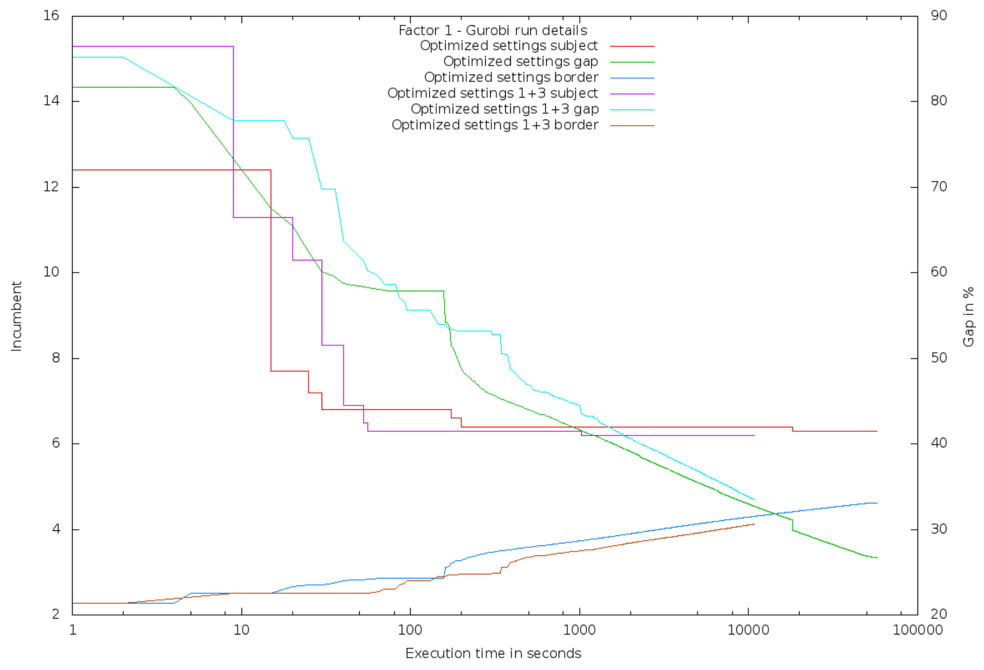


Figure 6.2.2.F=1.c - Compare optimized "1+3" (3 hours run) vs. optimized settings, factor 1

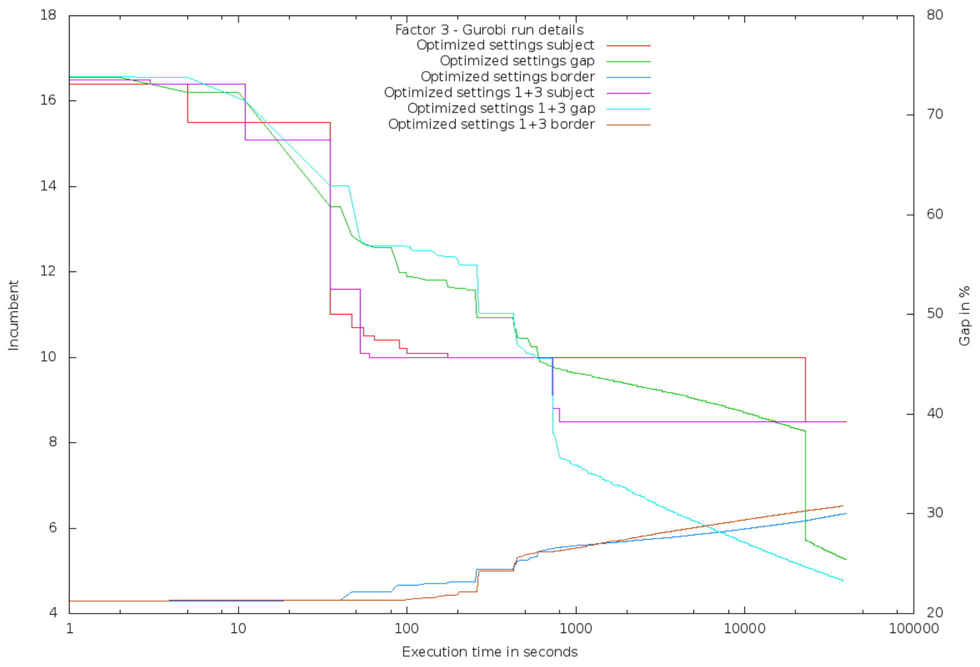


Figure 6.2.2.F=3.c - Compare optimized "1+3" (10.5 hours run) vs. optimized settings, factor 3

The "1+3" optimization parameters are: "GomoryPasses=0, PrePasses=3, Heuristics=0.5". The two figures above show that the best result known (6.2 after 1010 secs for factor 1 and 8.5 after 800 secs for factor 3) can be reached within 1100 secs. The negative side effect is that the "indicator for optimality", the gap value, cannot be used when reaching the lowest incumbent as at that time the gap is too high.
The descent of the gap value for the "1+3" optimization seems to be higher. But this is only because it is relative to the current subject. To better demonstrate this, figure 6.2.2.F=3.c also contains the lower border of the subject. One can see that after the best subject has been reached by the "factor=3" optimization, the lower border slope becomes slightly higher and therefore seems to get closer to the border value of "1+3". In the end of the run this results in a higher descent of the gap than for the "1+3" optimization.

Having a look at the gap of the run with load factor 1, the gap of the "1+3" optimization is most of the time and even at the end higher than the gap of the "factor=1" optimization. This is in contrast to the run with load factor 3 where the gap was most of the time better for the optimization "1+3". So with respect to gap improvement the "1+3" strategy seems not be to optimal for all factors.

Within the factor 3 tuning run another set has been spotted but not selected as good as the gap was too high. But the subject found by one trial (a run with a certain seed) for this parameter set was very promising. The parameter set is "MIPFocus=1". This means that the MIP solution strategy is to focus on finding feasible solutions instead of realizing a standard mix of increasing the boundary, proving optimality and finding feasible solutions. The best trial run compared to the "1+3" run with factor 3 delivered the following result:

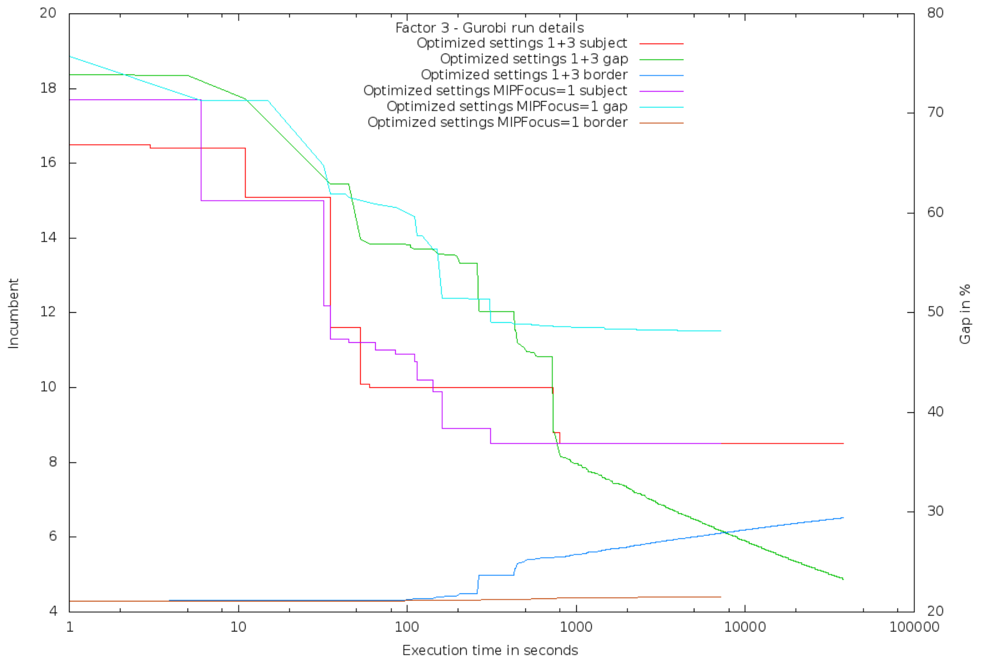


Figure 6.2.2.F=3.d - Compare optimized "MIPFocus" (2 hours run) vs. optimized "1+3" settings, factor=3

This figure shows that even though the best known result is found again faster than in the "1+3" run for load factor 3 (310 secs vs. 800 secs), the gap is worse. Of course, finding the best known subject that fast is pretty lucky which one can see from the fact that the other not shown trial only found 9.3 after 900 secs. But this parameter set can be used with several seeds to get a good impression of how good it can become at least. With a relatively high likelihood a very good result is found.

What has been stated but not justified yet is the usage of several seeds to get a good result for a certain tuned parameter set. While running the tuning mode, the importance of a good starting point for optimization algorithms that depend on random number generators became obvious. The following tables shall summarize some of the observations for 100 secs (factor 1) and 300 secs (factor 3).

Table 6.2.2.SeedExample1 - 100 secs, factor=1

Parameter set	Seed	Subject	Gap
default	default	7.2	59,7%
	1	7.6	60.5%
BranchDir=-1	default	7.2	63.9%
	1	6.6	60.6%
MIPFocus=1	default	7.3	64.4%
	1	6.9	62.3%

Table 6.2.2.SeedExample2 - 300 secs, factor=3

Parameter set	Seed	Subject	Gap
default	default	11.6	55.2%
	1	11.5	53.9%
Heuristics=0.5	default	10.6	50.9%
	1	10.6	50.9%
MIPFocus=1	default	10.7	58.9%
	1	8.5	48.2%

One can see that not for all trials a difference can be found. But for the majority of the parameter sets a more or less high difference can be spotted. Assuming that the idea of an existence of good and bad seeds also holds for longer execution times, a test with one parameter set has been done and proved its "correctness". Even though the statistic behind this test is very poor, the explicit test together with implicit tests indicates the correctness of the assumption.

One last idea of further improving the performance was not beneficial. However it is explained in the following.

The "MIPFocus=1" tuned parameter set has already been introduced. It has also been said that the good results for that are leading to higher gaps at the end. So by first improving the gap with a more stable and with respect to gap reduction better algorithm and later focusing on trying to find feasibly solutions, the result time of "1+3" could be further improved. Gurobi offers for such ideas the parameter "improvetime". This one defines the time after which the MIPFocus switches to mode 1 (finding feasibly solutions). This effectively means to set the parameter set "Heuristics=0.5 and RINS=10" (RINS defines how often to apply the equally named heuristic). Tests using a load factor value of 3 with an "improvetime" value of 30 minutes and a total duration of 1 hour reveal only for the "factor=1" parameter set a faster result (2200 secs instead of 9000 secs). For the "factor=3" parameter set this result is even worse as the total optimization time of 1 hour is not long enough.

For the "1+3" optimization the result was equally fast found which is caused by the improve time being higher than 760 secs. As in the first phase the run is not influenced by the "improvetime" parameter, the result is found as fast as before. Changing the improve time down to 500 secs leads after 800 secs again to the best known value 8.5. Setting the improve time to 300 seconds results in the best value to be found after 1030 secs. So it takes more time for "1+3" optimization to find the best value. So the "improvetime" optimization is unable to further optimize the already best tuned parameter set "1+3".

Summary

If not stated otherwise, the runs of the next chapters are all independently of the load factor and the number of mesh nodes executed in the following 2 ways:

- "MIPFocus": 4 trials of 900 secs each with "MIPFocus=1" (total time 1 hour)
- "1+3": "1+3" optimization for 3600 secs: "GomoryPasses=0, PrePasses=3, Heuristics=0.5", before that: 3 trials of 900 secs each (total time 1.75 hours)

As the time to result with 2 hours and 45 minutes for each model configuration is too high, an annealing run cannot be done. But one can try to confirm the results for the perfect memory controller configurations of the previous models.

What one also has to keep in mind: The found result is not guaranteed to be optimal and so needs to be used and interpreted with care. Both optimizations "MIPFocus" and "1+3" are developed to deliver a low subject as fast as possible and will therefore lead to poor gap values as their execution time is too short. Especially for "MIPFocus" even a longer execution time would not lead to a much better gap as it focuses on finding feasible solutions and not on improving the gaps.

6.2.3. Factor selection

A good load factor for the computational load is required for the mesh model in order to use the same eps and zeta combinations as before. Therefore, a selection run with the original cluster configuration together with the eps values 0.1, 0.5 and 0.9 is used. Zeta is set to 0.5. The mesh to be mapped is of size 4x6. The goal of the selection runs is to find a factor that fully utilizes the model for eps=0.9 and only uses a couple of tiles for eps=0.1.

Table 6.2.3 - Tiles used for different configurations (24 corresponds to 100% usage)

Tiles used	3	10	30
eps=0.1	1*	3*	3
eps=0.5	3	12	24
eps=0.9	24	24	24

The marker * refers to an optimal result. All other values are as good as possible when running tuning mode "1+3".

The run for factor 10 uses 50% of the tiles for eps=0.5. This and the fact that 3 tiles are used for eps=0.1 makes this superior over the factor 3 run. The run with factor 30 is never optimal and other than that leads to 100% usage already for eps=0.5.

6.2.4. Results and analysis

The mesh network tested now has the same dimensions like the Intel SCC cluster mesh network. Therefore a natural mapping can be reached. But the tasks are mapped to tiles instead of cores. This little drawback is due to the higher complexity of a model with 48 cores. If one assumes that each task can be parallelized (which it should as the mesh network is already parallelized) and processed by the 2 cores of a single tile so that no huge communication overhead exists, then this little limitation becomes acceptable.

The assumption of the outcome of the following tests is that the best configuration found before for 4 memory controllers is also better for the mesh network model. In case of a perfect mapping, this can be derived by summing up the distances of the nodes to the memory controller tiles. For the Intel SCC configuration, the distance is 20. For the best configuration (MC placed at tile 1, 4, 19 and 22) it should be only 16. And like before the addition of more memory controllers should result in even better performance.

As already said, for the mesh network an annealing optimization run is not reasonable. Instead of that the following memory controller configurations are tested:

- Best 4MC: 1 4 19 22 (referred to as "best configuration")
  - The winning configuration of all previous chapters.
- Best 8MC: 2 3 6 11 12 17 20 21
  - The winning configuration for 8 MC of all previous chapters.
- Best 12MCA: 1 2 3 4 6 11 12 17 19 20 21 22
  - The winning configuration for 12 MC of chapter 5 (Pipelined merge sort).
- Best 12MCB: 0 1 2 3 4 5 18 19 20 21 22 23
  - The winning configuration for 12 MC of chapter 6.1 (Tiled-MapReduce).

The following table summarizes the results of the model runs for each combination of eps and zeta. The result values and the row headers are encoded as already described in chapter 5.2.

Table 6.2.4.1 - Results for 4x6 mesh model

eps_zeta	Intel	Best 4MC	Best 8MC	Best 12MCA	Best 12MCB
0.1_0.1	19.92_120_8_16	19.92_120_8_16	18.48_120_8_0	18.48_120_8_0	18.48_120_8_0

All runs are optimal when using the "1+3" optimization.

The Intel configuration is as good as the best one found for 4 MC.

In case of 4 MC, only 2 tiles are used. This means 12 nodes per tile. One of the tiles is a memory controller tile and the other not. This results in memory controller distance costs. Once more memory controllers are added, the result should not become better but it does. This is because there the at least 2 memory controller tiles are next to each other. So the memory distance becomes 0. This can be seen in the figure 6.2.4.1.

eps_zeta	Intel	Best 4MC	Best 8MC	Best 12MCA	Best 12MCB
0.1_0.5	18.8_80_16_8	22.4_80_24_8	15.6_120_8_0	15.2_80_16_0	15.2_80_16_0

All runs but the one with 12 memory controllers are optimal when using the "1+3" optimization.

The computational load is improved for 4 MC and 12 MC. There 3 to 4 nodes are used instead of 2 for zeta=0.1.

This time, the best configuration is worse than the original Intel configuration. This is caused by the placement of the MC which are closer to each other in the Intel case. This can be seen in the mapping in figures 6.2.4.2 and 6.2.4.3.

Adding more memory controllers results in higher performance due to decreasing communication costs. In the case of 8 MC the node mapping does not change to before (see figure 6.2.4.1) which means the maximum computational load is higher than for all other cases. As before, there is no difference visible in the optimal mapping for the 2 configurations with 12 MC. The figure 6.2.4.4 shows this mapping.

eps_zeta	Intel	Best 4MC	Best 8MC	Best 12MCA	Best 12MCB
0.1_0.9	10.48_40_72_0	10.12_40_68_0	8.68_40_52_0	7.76_20_64_0	7.6_40_40_0

All runs with 4 MC are optimal when using the "1+3" optimization. For 8 MC and for "Best 12MCA" there is a discrepancy between the "MIPFocus" and the "1+3" optimization result. The former one is better and listed here.

By putting more importance into the memory controller distance and therefore less into the node to node communication costs, the load distribution is increased further compared to zeta=0.5.

This time the best configuration is better than the Intel one. The difference can be seen in the figures 6.2.4.5 (Intel) and 6.2.4.6 and 6.2.4.7 (two possible optimal mappings can be found).

The only difference in the results found for the two configuration versions with 12 MC can be seen here. The configuration which is best for Tiled-MapReduce is slightly better than the best one found for Pipelined merge sort.

The figures 6.2.4.8 to 6.2.4.10 demonstrate the mapping for 8 and 12 MC. There one can see that the subjects compared against the results for 4 MC are better as the distance between the memory controller tiles is smaller which leads to higher node to node communication costs for 4 MC.

eps_zeta	Intel	Best 4MC	Best 8MC	Best 12MCA	Best 12MCB
0.5_0.1	33.8_40_28_24	34.4_40_28_36	33.6_40_28_20	33.4_40_28_16	33.4_40_28_16
	From now on, no result is optimal any more. For the Intel configuration there is a discrepancy between the "MIPFocus" and the "1+3" optimization result. The former one is better and listed here. Here, like for eps=0.1 and zeta=0.5, the best configuration is worse than the Intel configuration which has the same reason like there. For all configurations the differences to eps=0.1 and zeta=0.9 are the higher memory distance and lower node to node communication costs. The mapping for Intel, Best 8MC and Best 12MCA is shown below in the figures 6.2.4.11 to 6.2.4.13. There one can see an equal mapping that only differs in the distance to memory property which is caused by the different memory controller locations.				
0.5_0.5	32_20_64_24	29_20_60_16	27.5_20_56_14	25_20_52_8	25_20_52_8
	For the Intel and the "Best 8MC" configuration there is a discrepancy between the "MIPFocus" and the "1+3" optimization result. The latter one is better and listed here. For the "Best 12MCA" this difference also exists but is the other way around. The change to zeta=0.1 is the better spread of the nodes over the cluster. Even though the node to node communication costs are higher, the memory controller distance can be decreased in all cases but Intel. The more MC are added, the better the result becomes. From now on, the Intel result is always worse than the best configuration. Figures 6.2.4.14 to 6.2.4.18 illustrate the changed mappings.				
0.5_0.9	21_20_76_16	20.2_20_60_16	13.8_20_76_0	12.4_10_76_8	12.4_10_76_8
	For the Intel and "Best 12MCA" configuration there is a discrepancy between the "MIPFocus" and the "1+3" optimization result. The former one is better and listed here. The mapping of the Intel and "Best 8MC" configuration is changed to before. The nodes are more spread around the cluster. Therefore, the memory distance is smaller and the node to node distance higher. This can be seen in the figures 6.2.4.19 and 6.2.4.20. If 12 MC are used, the natural mapping is the optimal one from now on. This can be seen in the figure 6.2.4.21 for the "Best 12MCB" configuration.				
0.9_0.1	16.24_10_76_40	16.16_10_76_32	16_10_76_16	15.92_10_76_8	15.92_10_76_8
	This is the first eps and zeta configuration for which each memory controller configuration has a natural mapping as the optimal result. One can see that the expectation for the natural mapping and 4 MC is met. The Intel configuration is not as good as the best one. Also adding more memory controllers results in better subjects.				
0.9_0.5	14.8_10_76_40	14.4_10_76_32	13.6_10_76_16	13.2_10_76_8	13.2_10_76_8
	There is no difference to before.				

eps_zeta	Intel	Best 4MC	Best 8MC	Best 12MCA	Best 12MCB
0.9_0.9	12.76_10_88_32	12.08_10_92_24	11.2_10_76_16	10.48_10_76_8	10.48_10_76_8

The discrepancy between "MIPFocus" and "1+3" can be healed for the Intel configuration by running the "1+3" optimization for 2 hours instead of only 1 hour.

The optimization results for the configurations Intel, "Best 4MC" and "Best 12MCB" are as good as optimal (<5% gap).

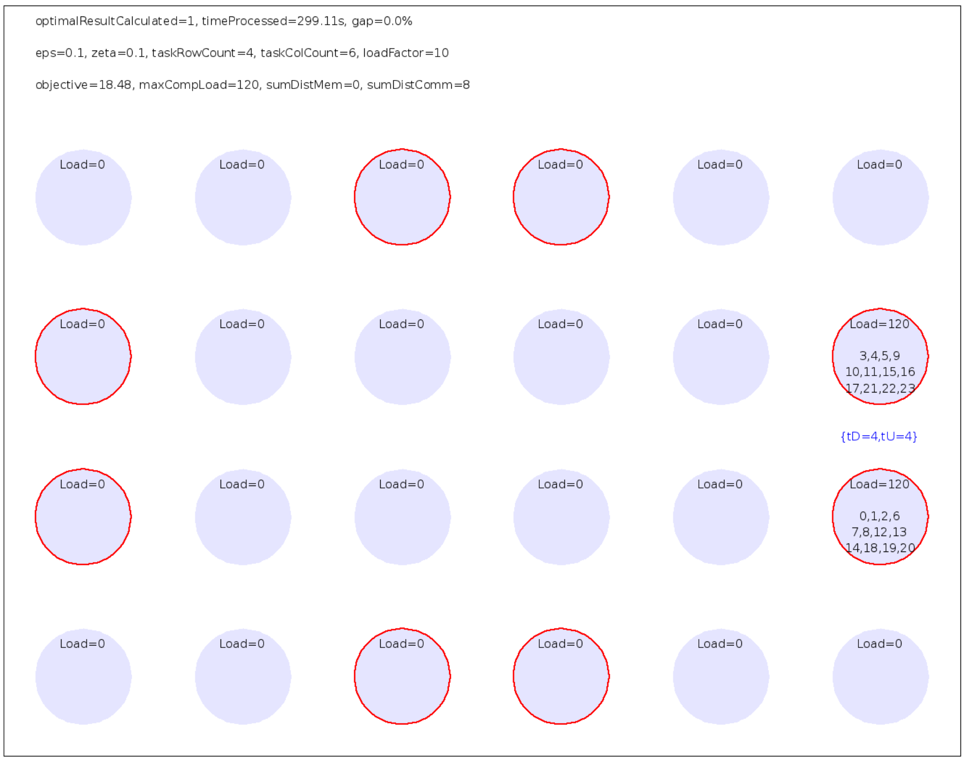
For the configurations with 8 and 12 MC there is no difference to before. But looking into the mapping with 4 MC one can see something surprising. The natural mapping is not the perfect one anymore. The cluster is still perfectly used, but some consecutive mesh rows (Intel) or columns (Best 4MC) are switched. This can be seen in the figures 6.2.4.22 and 6.2.4.23.

In general for 4 memory controllers the best configuration for Tiled-MapReduce and Pipelined merge sort is still the best one that can be used. Even though it cannot play out its advantages for cases where not all cluster tiles are used, it is better in 6 out of 9 cases.

The configuration with 8 memory controllers is still better than all 4 memory controller configurations.

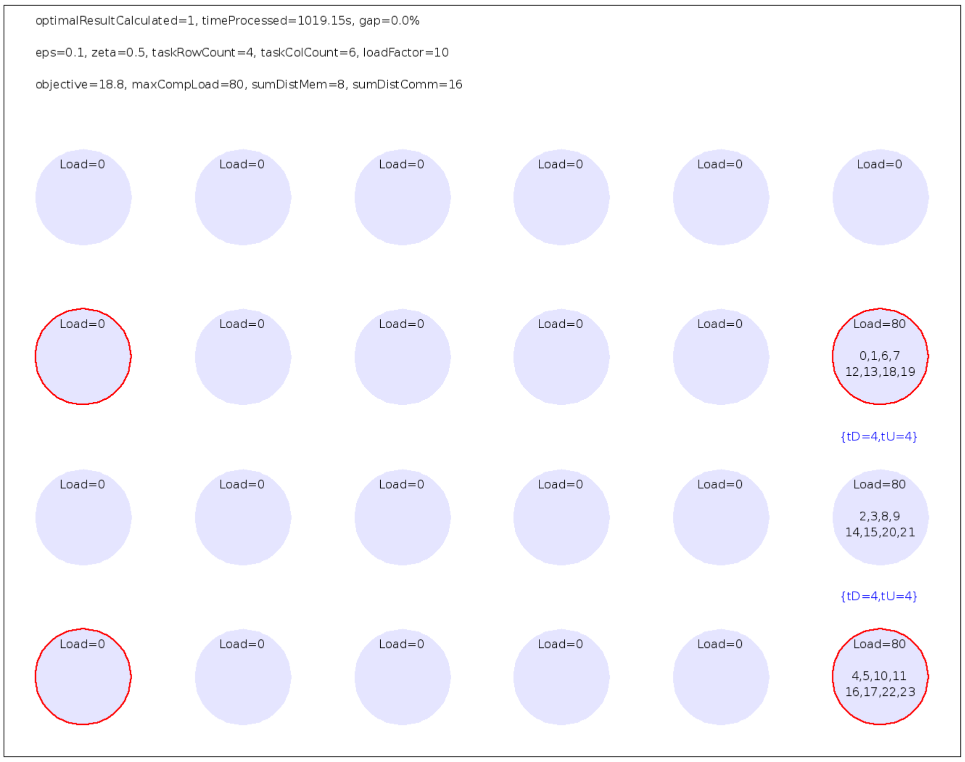
Taking a look into the results with 12 memory controllers, one can see nearly no difference. But with a very small gap the Tiled-MapReduce configuration is winning the comparison.

The following figures are a selected set of the results mentioned above. They visualize the mapping as already introduced in the previous chapters. But this time the mapping is more detailed. One can see the node indexes in the tile it is mapped to. The mapping of mesh index to the mesh node is working as for the Intel SCC tiles (see chapter 4.3). Also the gap value is displayed now. With respect to communication one can see the node to node communication load (between the tiles) but the memory communication load is excluded.



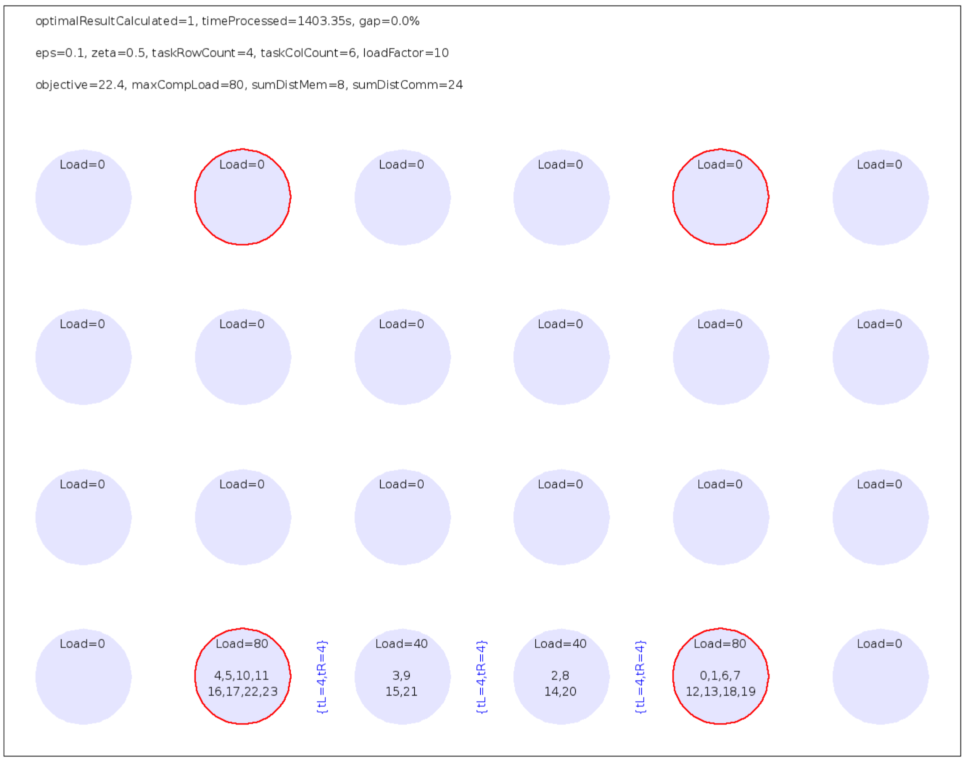
```
optimalResultCalculated=1, timeProcessed=299.11s, gap=0.0%

eps=0.1, zeta=0.1, taskRowCount=4, taskColCount=6, loadFactor=10

objective=18.48, maxCompLoad=120, sumDistMem=0, sumDistComm=8
```

Figure 6.2.4.1 - Best 8MC, eps=0.1, zeta=0.1



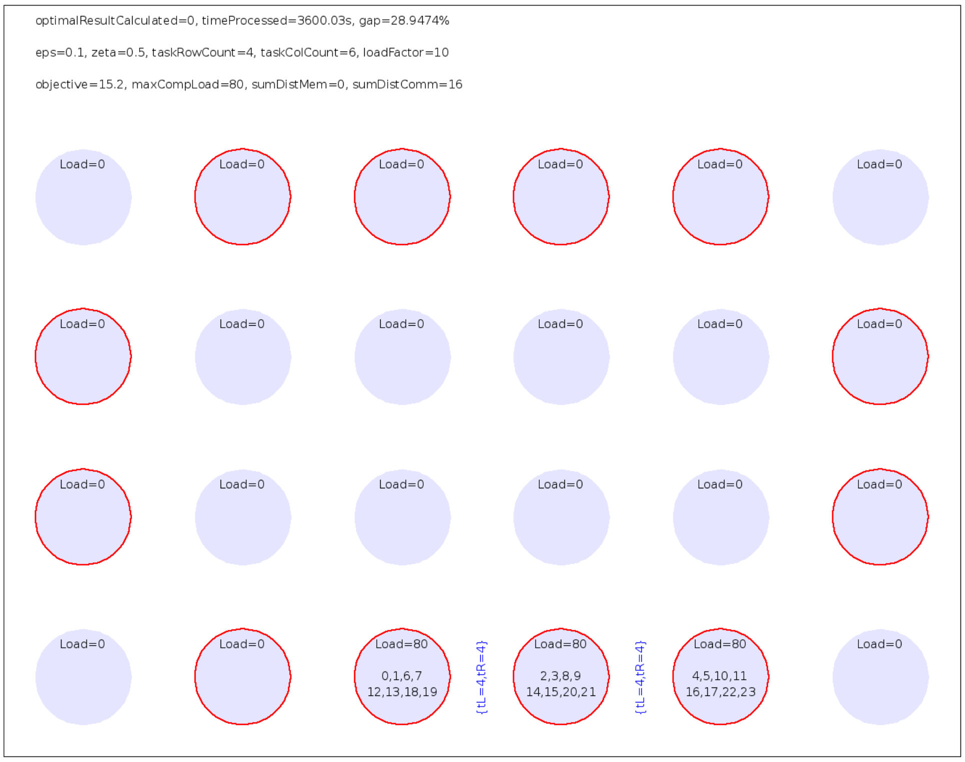
```
optimalResultCalculated=1, timeProcessed=1019.15s, gap=0.0%
eps=0.1, zeta=0.5, taskRowCount=4, taskColCount=6, loadFactor=10
objective=18.8, maxCompLoad=80, sumDistMem=8, sumDistComm=16
```

Figure 6.2.4.2 - Intel, eps=0.1, zeta=0.5



```
optimalResultCalculated=1, timeProcessed=1403.35s, gap=0.0%
eps=0.1, zeta=0.5, taskRowCount=4, taskColCount=6, loadFactor=10
objective=22.4, maxCompLoad=80, sumDistMem=8, sumDistComm=24
```

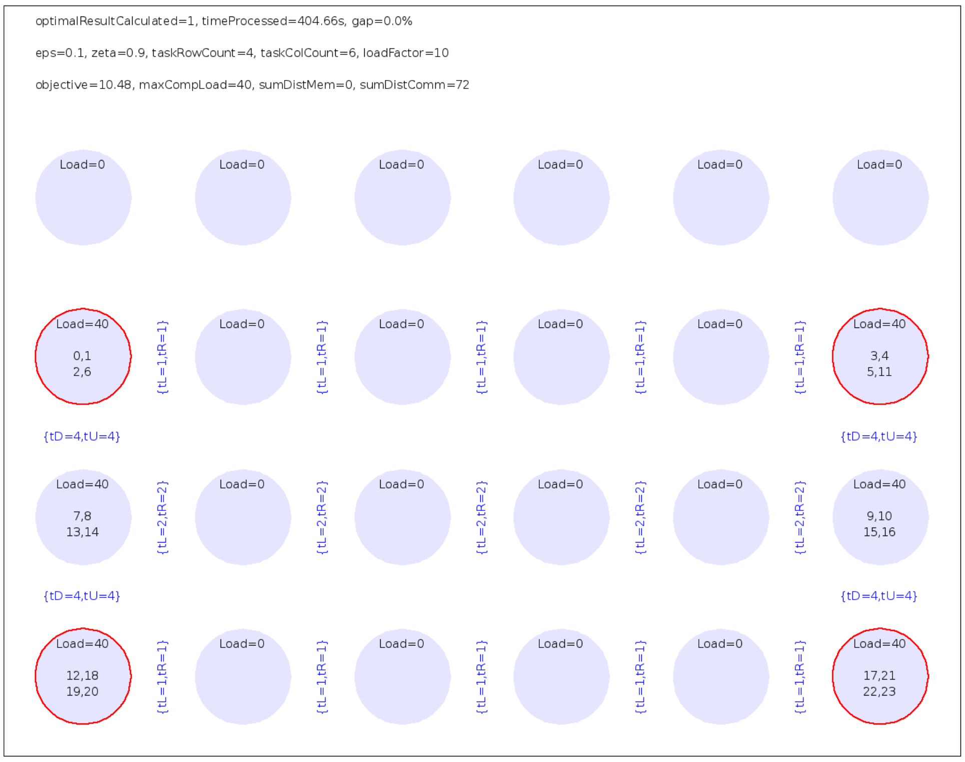
Figure 6.2.4.3 - Best 4MC, eps=0.1, zeta=0.5


optimalResultCalculated=0, timeProcessed=3600.03s, gap=28.9474%

eps=0.1, zeta=0.5, taskRowCount=4, taskColCount=6, loadFactor=10

objective=15.2, maxCompLoad=80, sumDistMem=0, sumDistComm=16

Figure 6.2.4.4 - Best 12MCA, eps=0.1, zeta=0.5


optimalResultCalculated=1, timeProcessed=404.66s, gap=0.0%

eps=0.1, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=10.48, maxCompLoad=40, sumDistMem=0, sumDistComm=72

Figure 6.2.4.5 - Intel, eps=0.1, zeta=0.9

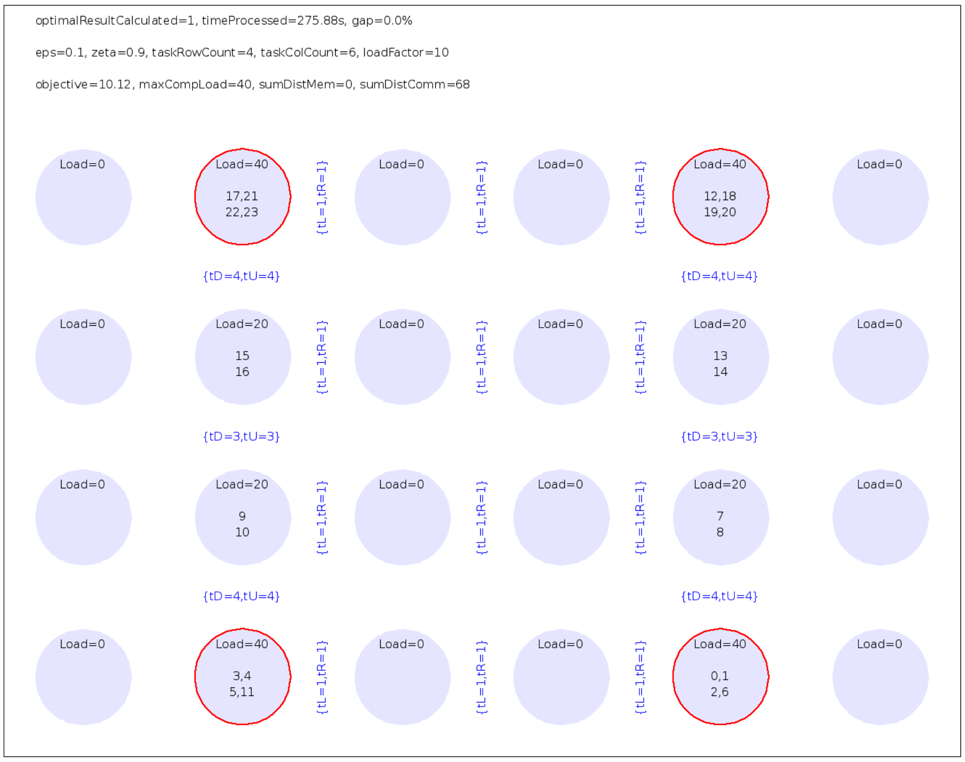


Figure 6.2.4.6 - Best 4MC, eps=0.1, zeta=0.9, mapping 1

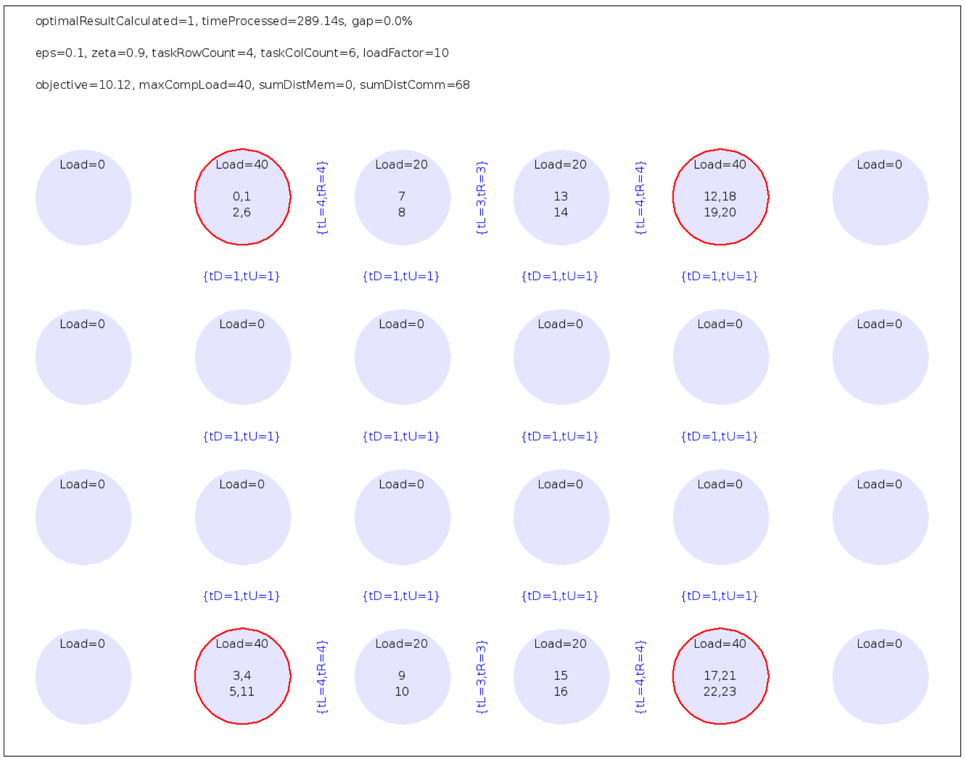
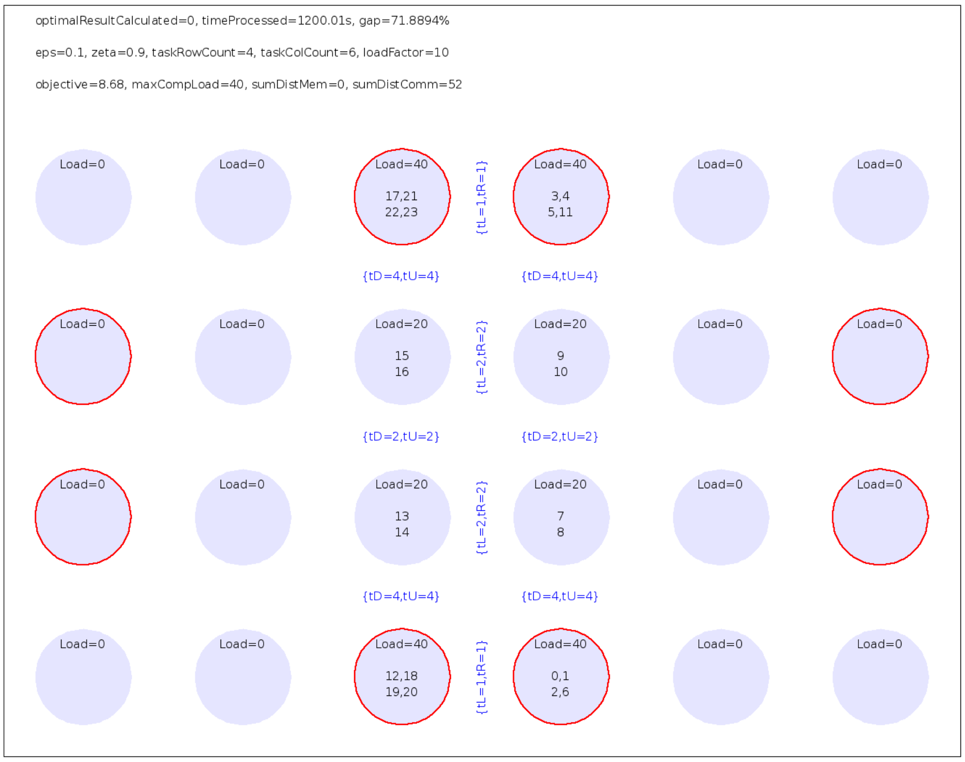
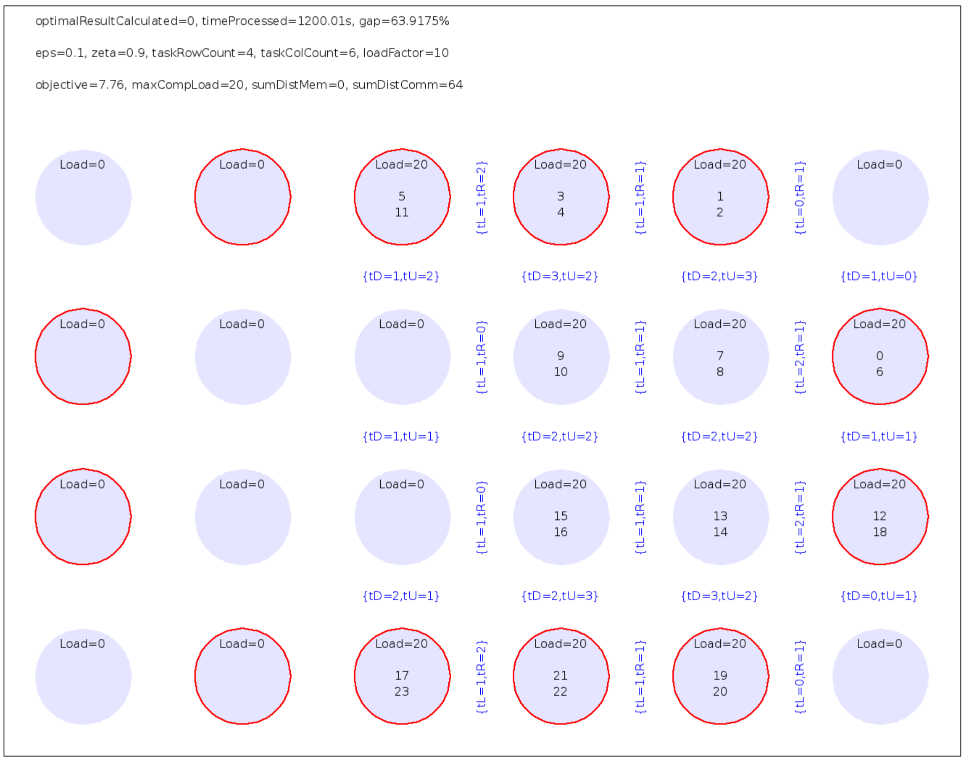


Figure 6.2.4.7 - Best 4MC, eps=0.1, zeta=0.9, mapping 2



```
optimalResultCalculated=0, timeProcessed=1200.01s, gap=71.8894%

eps=0.1, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=8.68, maxCompLoad=40, sumDistMem=0, sumDistComm=52
```

Figure 6.2.4.8 - Best 8MC, eps=0.1, zeta=0.9



```
optimalResultCalculated=0, timeProcessed=1200.01s, gap=63.9175%

eps=0.1, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=7.76, maxCompLoad=20, sumDistMem=0, sumDistComm=64
```

Figure 6.2.4.9 - Best 12MCA, eps=0.1, zeta=0.9

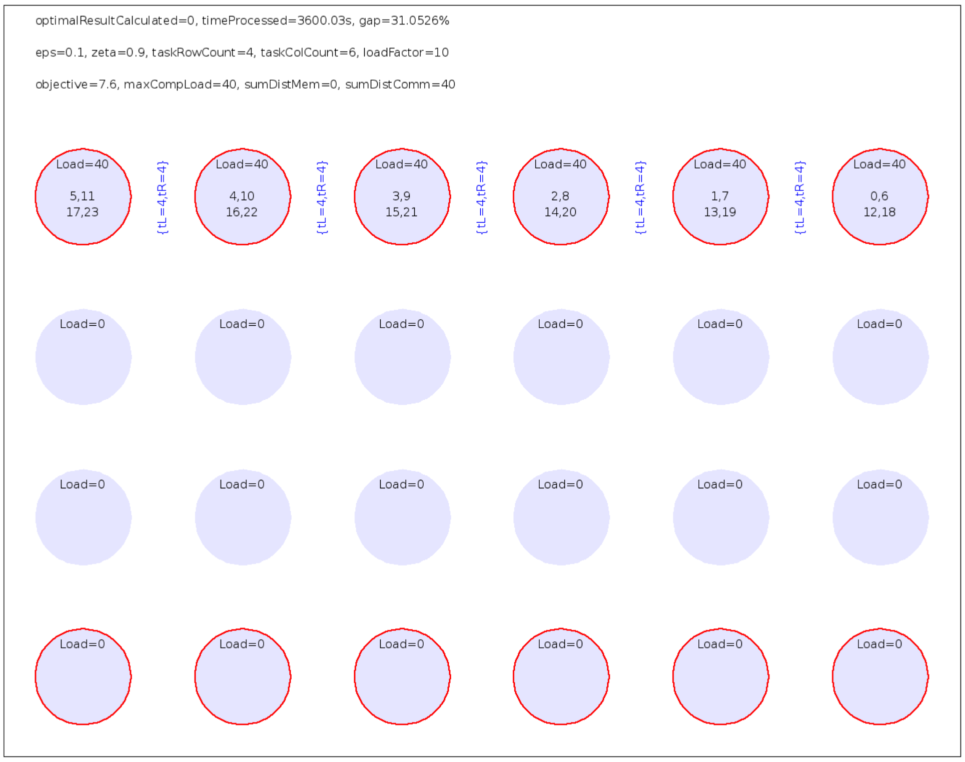

optimalResultCalculated=0, timeProcessed=3600.03s, gap=31.0526%
eps=0.1, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10
objective=7.6, maxCompLoad=40, sumDistMem=0, sumDistComm=40
Load=40 5,11 17,23
{tL=4,tR=4}
Load=40 4,10 16,22
{tL=4,tR=4}
Load=40 3,9 15,21
{tL=4,tR=4}
Load=40 2,8 14,20
{tL=4,tR=4}
Load=40 1,7 13,19
{tL=4,tR=4}
Load=40 0,6 12,18
Load=0

Figure 6.2.4.10 - Best 12MCB, eps=0.1, zeta=0.9

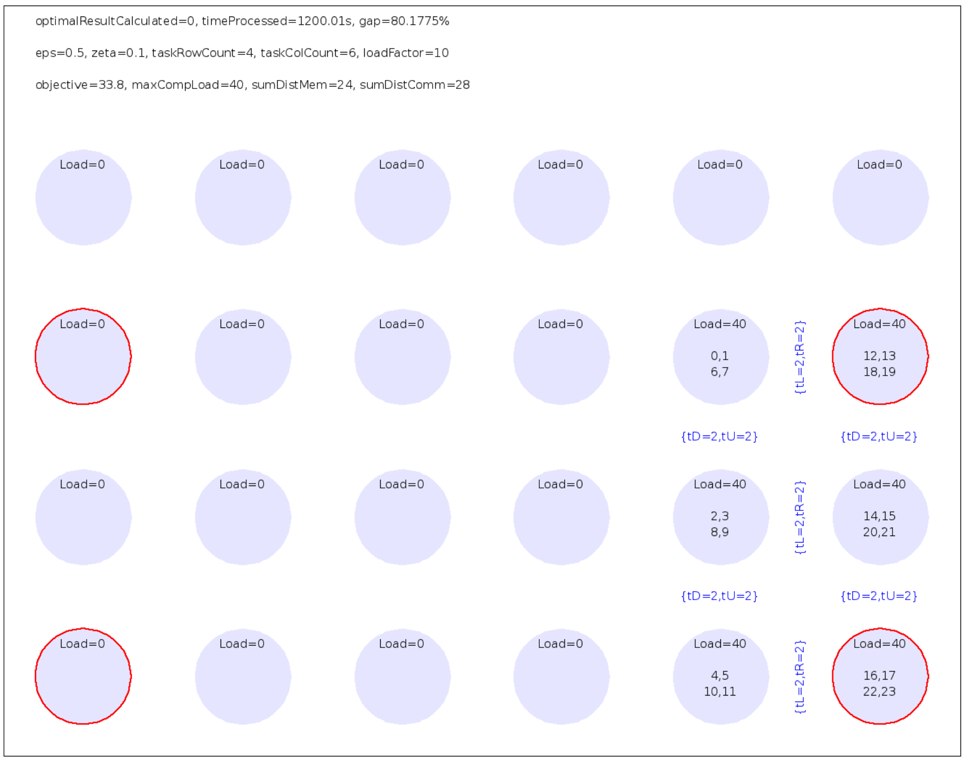
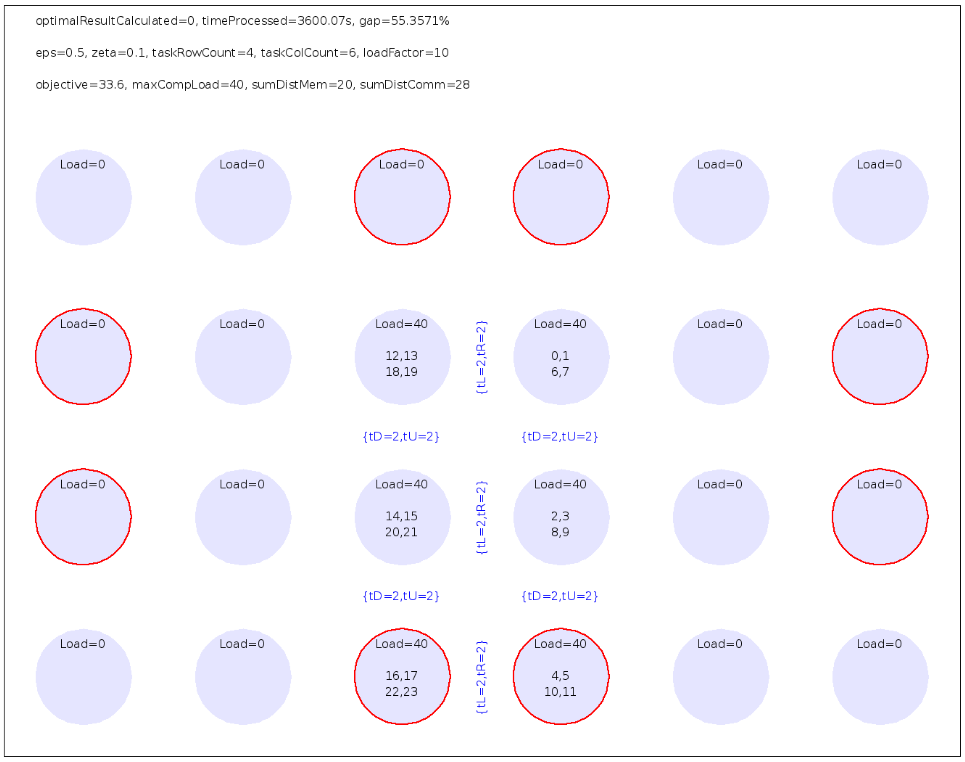

optimalResultCalculated=0, timeProcessed=1200.01s, gap=80.1775%
eps=0.5, zeta=0.1, taskRowCount=4, taskColCount=6, loadFactor=10
objective=33.8, maxCompLoad=40, sumDistMem=24, sumDistComm=28
Load=0
Load=40 0,1 6,7
{tL=2,tR=2}
Load=40 12,13 18,19
{tD=2,tU=2}
{tD=2,tU=2}
Load=40 2,3 8,9
{tL=2,tR=2}
Load=40 14,15 20,21
{tD=2,tU=2}
{tD=2,tU=2}
Load=40 4,5 10,11
{tL=2,tR=2}
Load=40 16,17 22,23

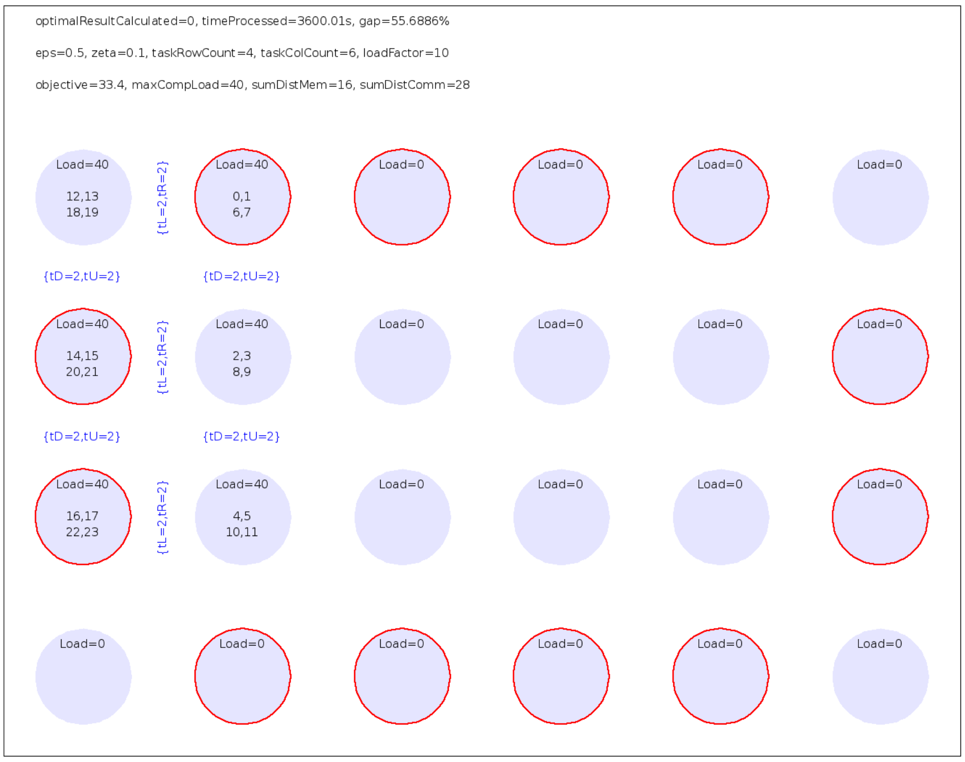
Figure 6.2.4.11 - Intel, eps=0.5, zeta=0.1


optimalResultCalculated=0, timeProcessed=3600.07s, gap=55.3571%

eps=0.5, zeta=0.1, taskRowCount=4, taskColCount=6, loadFactor=10

objective=33.6, maxCompLoad=40, sumDistMem=20, sumDistComm=28

Figure 6.2.4.12 - Best 8MC, eps=0.5, zeta=0.1


optimalResultCalculated=0, timeProcessed=3600.01s, gap=55.6886%

eps=0.5, zeta=0.1, taskRowCount=4, taskColCount=6, loadFactor=10

objective=33.4, maxCompLoad=40, sumDistMem=16, sumDistComm=28

Figure 6.2.4.13 - Best 12MCA, eps=0.5, zeta=0.1

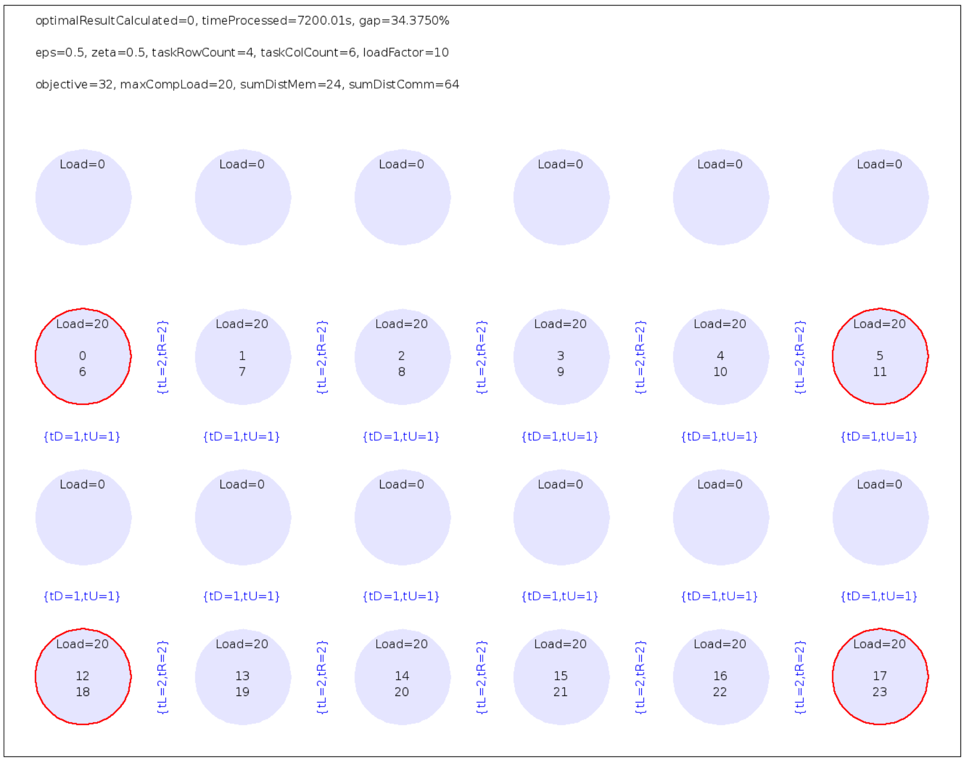

optimalResultCalculated=0, timeProcessed=7200.01s, gap=34.3750%
eps=0.5, zeta=0.5, taskRowCount=4, taskColCount=6, loadFactor=10
objective=32, maxCompLoad=20, sumDistMem=24, sumDistComm=64
Load=0
Load=20
0
6
1
7
2
8
3
9
4
10
5
11
{tL=2,tR=2}
{tD=1,tU=1}
12
18
13
19
14
20
15
21
16
22
17
23

Figure 6.2.4.14 - Intel, eps=0.5, zeta=0.5

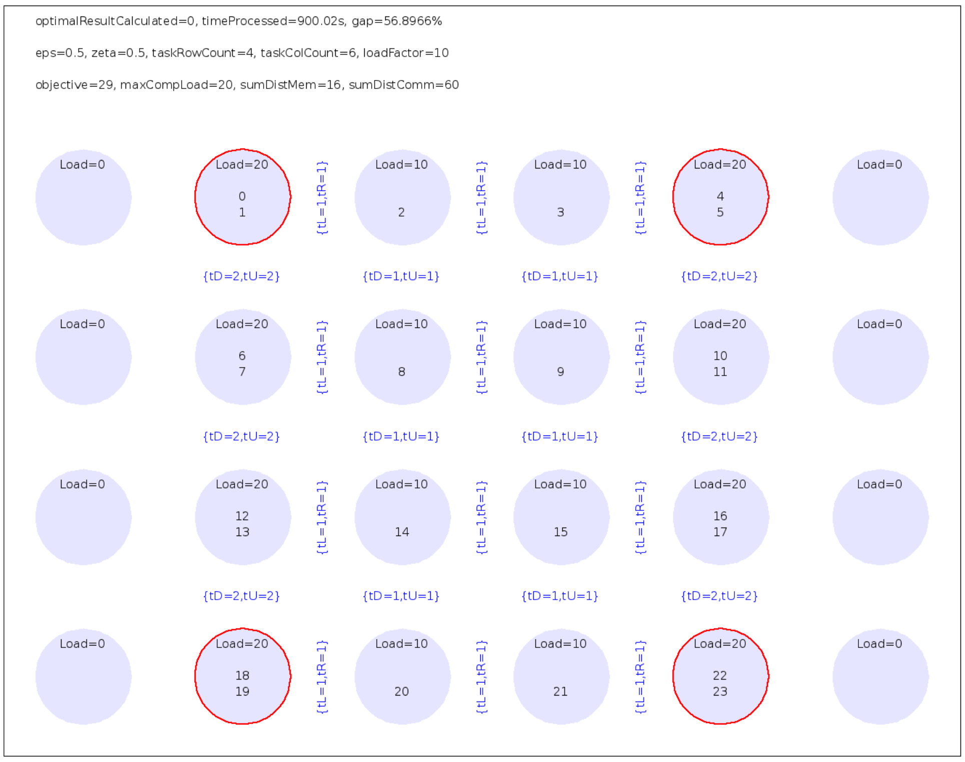

optimalResultCalculated=0, timeProcessed=900.02s, gap=56.8966%
eps=0.5, zeta=0.5, taskRowCount=4, taskColCount=6, loadFactor=10
objective=29, maxCompLoad=20, sumDistMem=16, sumDistComm=60
Load=0
Load=20
Load=10
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
{tL=1,tR=1}
{tD=2,tU=2}
{tD=1,tU=1}

Figure 6.2.4.15 - Best 4MC, eps=0.5, zeta=0.5, mapping 1

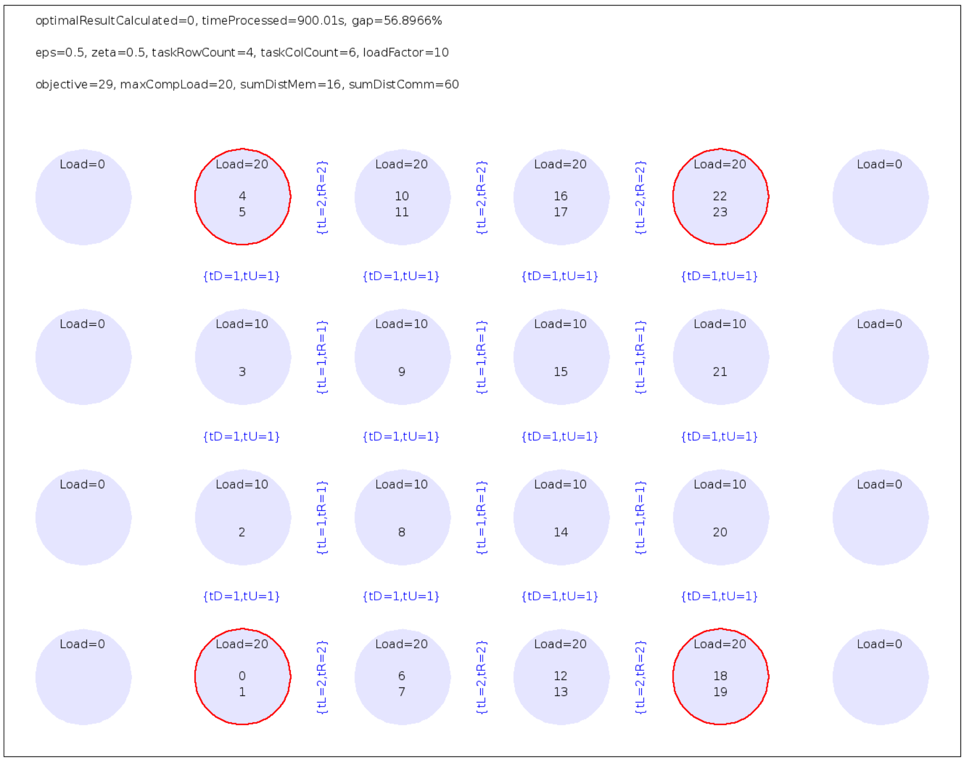


Figure 6.2.4.16 - Best 4MC, eps=0.5, zeta=0.5, mapping 2

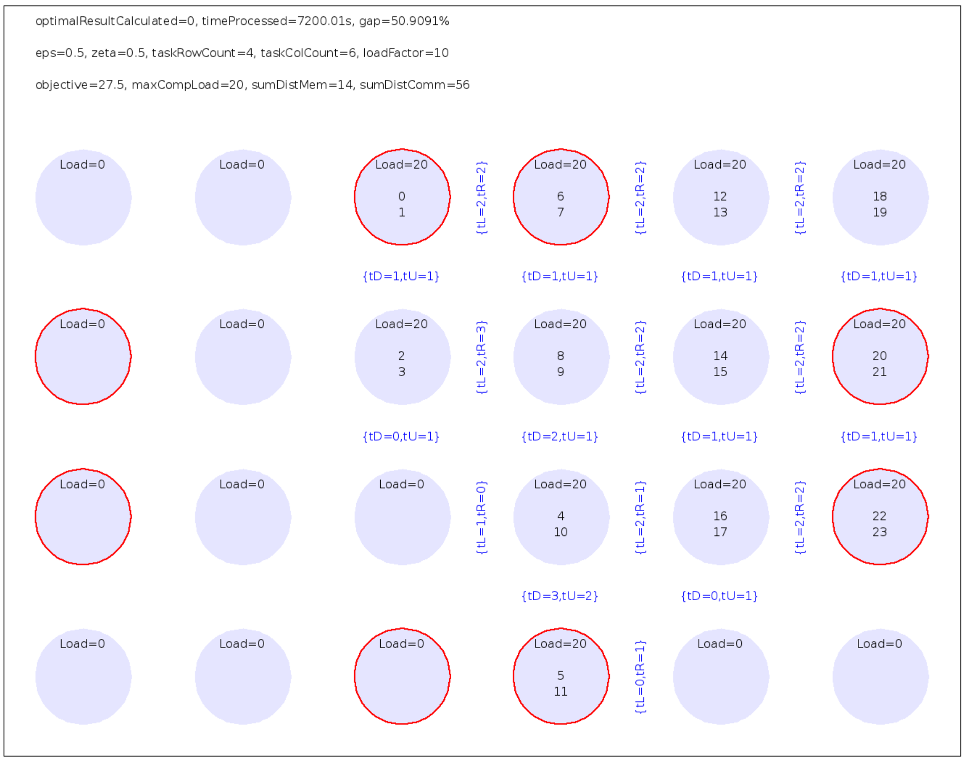
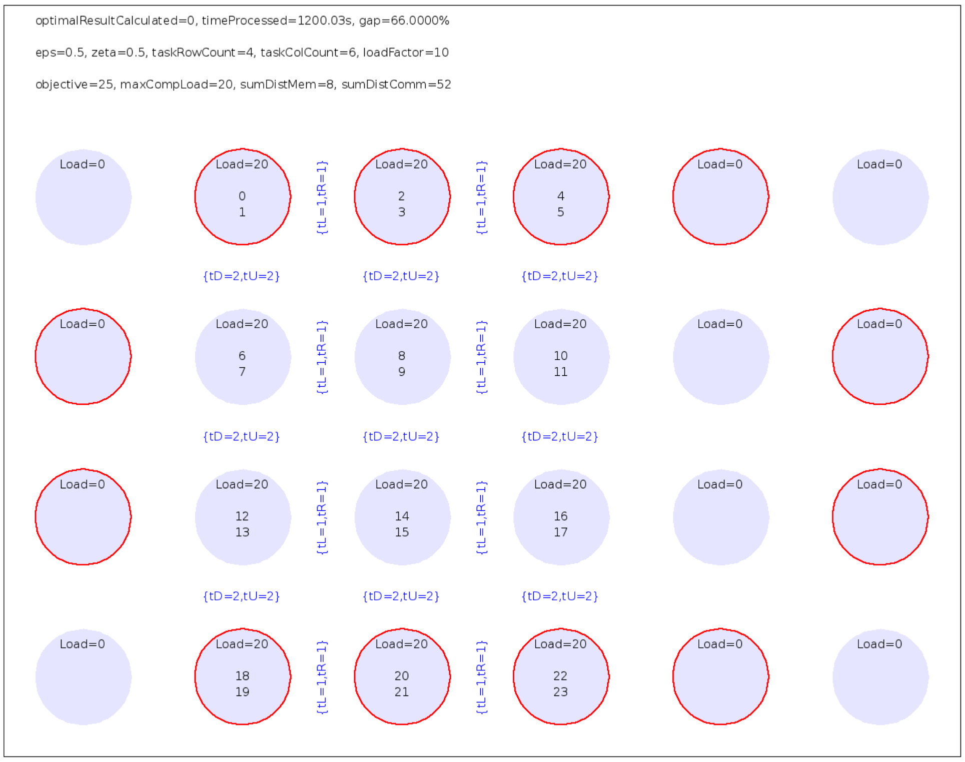
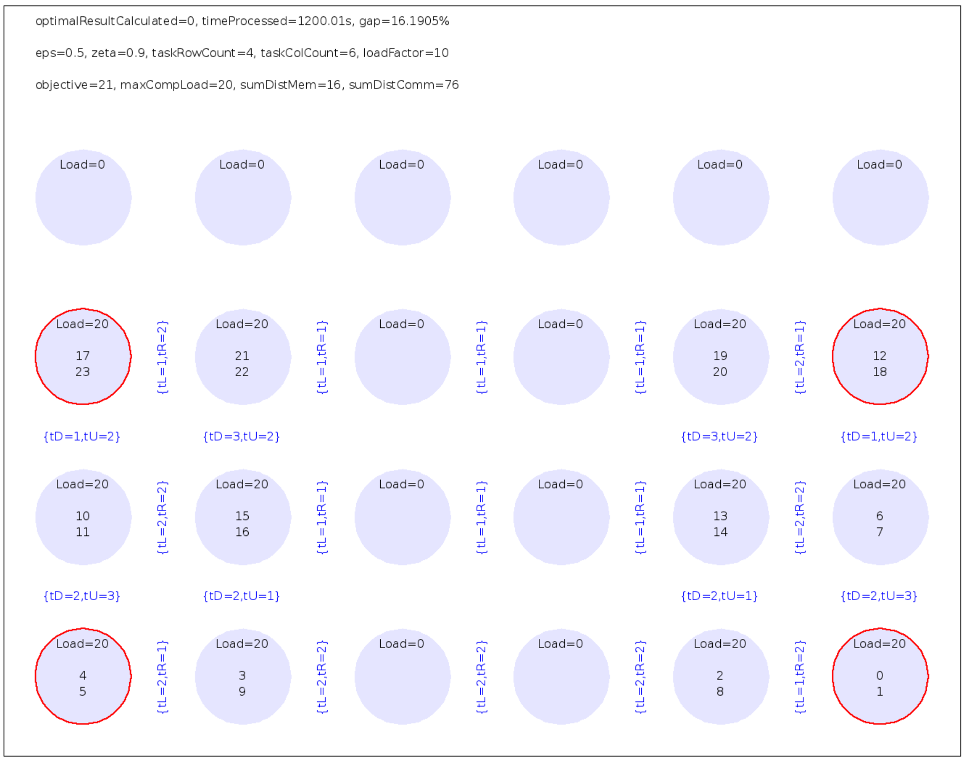


Figure 6.2.4.17 - Best 8MC, eps=0.5, zeta=0.5



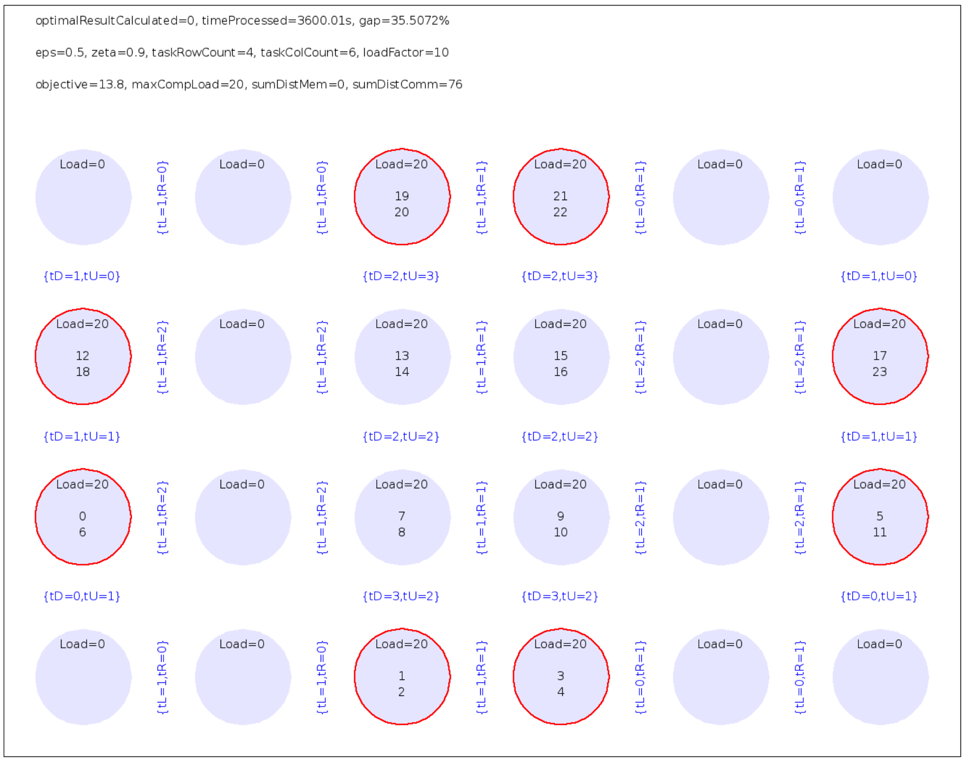
```
optimalResultCalculated=0, timeProcessed=1200.03s, gap=66.0000%
eps=0.5, zeta=0.5, taskRowCount=4, taskColCount=6, loadFactor=10
objective=25, maxCompLoad=20, sumDistMem=8, sumDistComm=52
```

Figure 6.2.4.18 - Best 12MCA, eps=0.5, zeta=0.5



```
optimalResultCalculated=0, timeProcessed=1200.01s, gap=16.1905%
eps=0.5, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10
objective=21, maxCompLoad=20, sumDistMem=16, sumDistComm=76
```

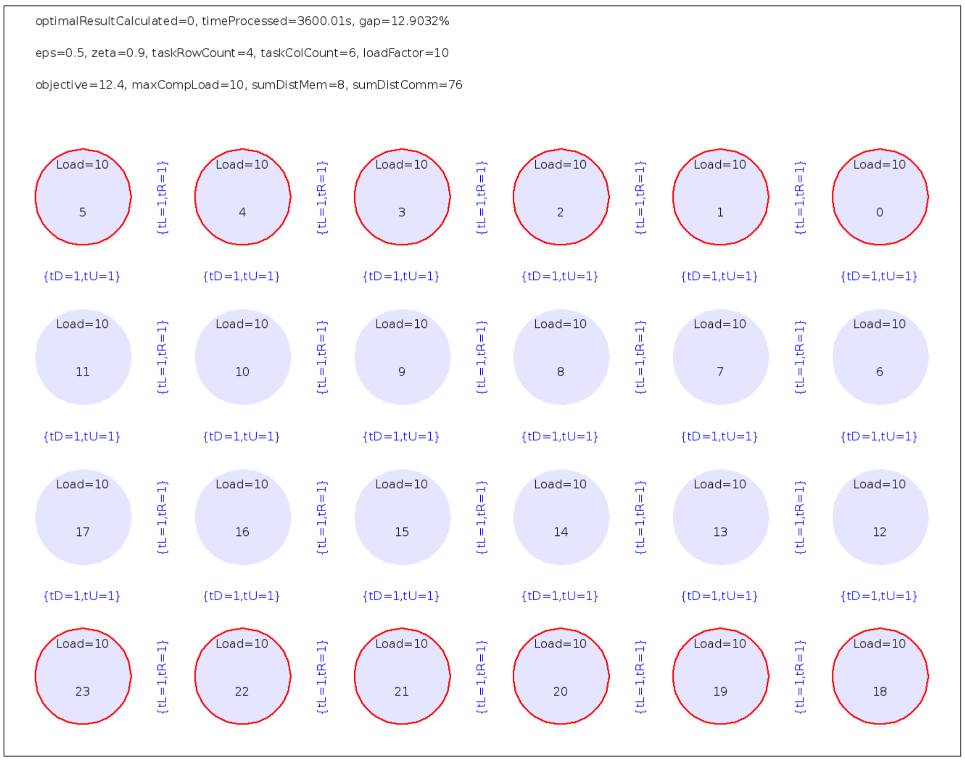
Figure 6.2.4.19 - Intel, eps=0.5, zeta=0.9


optimalResultCalculated=0, timeProcessed=3600.01s, gap=35.5072%

eps=0.5, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=13.8, maxCompLoad=20, sumDistMem=0, sumDistComm=76

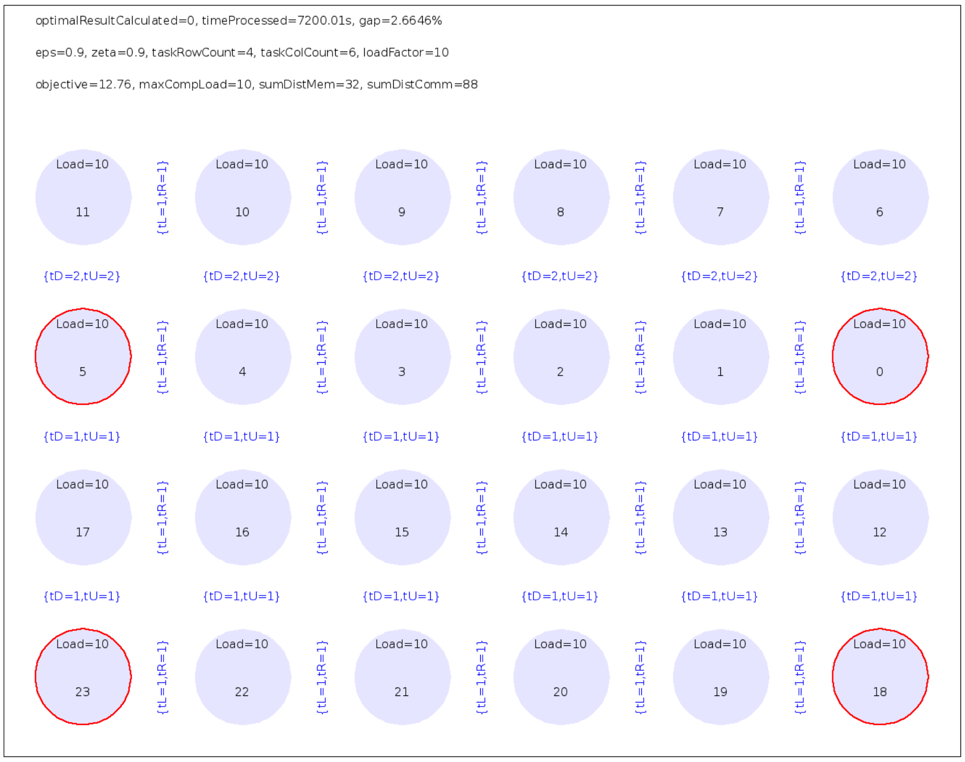
Figure 6.2.4.20 - Best 8MC, eps=0.5, zeta=0.9


optimalResultCalculated=0, timeProcessed=3600.01s, gap=12.9032%

eps=0.5, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=12.4, maxCompLoad=10, sumDistMem=8, sumDistComm=76

Figure 6.2.4.21 - Best 12MCB, eps=0.5, zeta=0.9, natural mapping

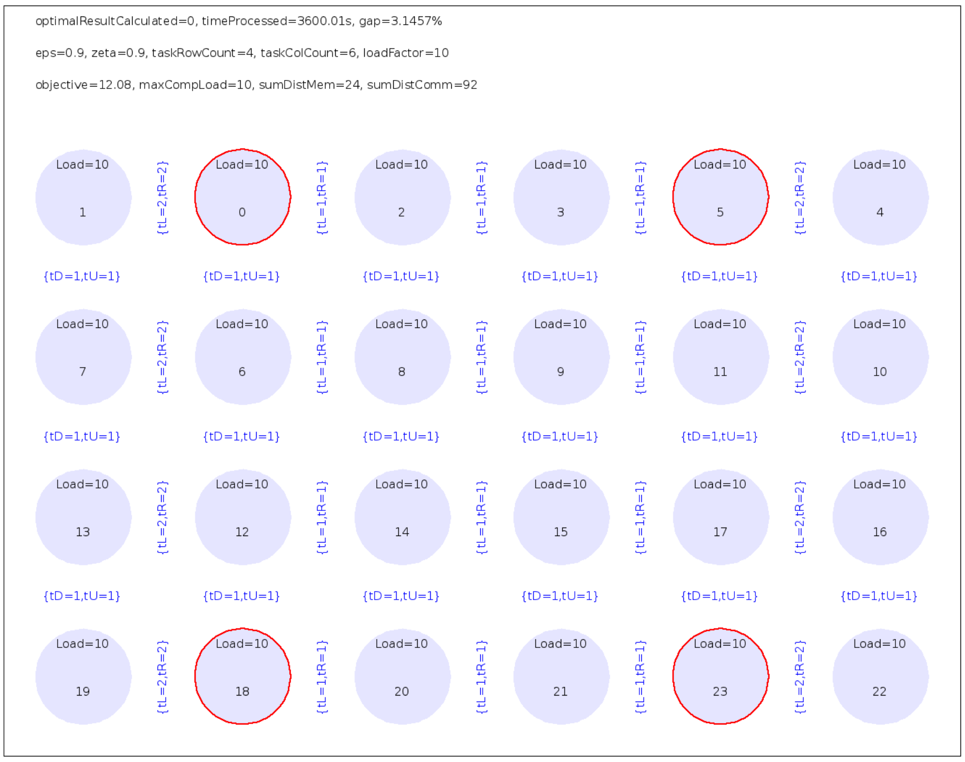


optimalResultCalculated=0, timeProcessed=7200.01s, gap=2.6646%

eps=0.9, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=12.76, maxCompLoad=10, sumDistMem=32, sumDistComm=88

Figure 6.2.4.22 - Intel, eps=0.9, zeta=0.9, no natural mapping any more



optimalResultCalculated=0, timeProcessed=3600.01s, gap=3.1457%

eps=0.9, zeta=0.9, taskRowCount=4, taskColCount=6, loadFactor=10

objective=12.08, maxCompLoad=10, sumDistMem=24, sumDistComm=92

Figure 6.2.4.23 - Best 4MC, eps=0.9, zeta=0.9, no natural mapping any more

7. Conclusion

As first part of this work the steadily growing importance of the many-core research has been outlined. Based on the significance of this topic, the idea of this work is to improve the processing performance of a modern many-core CPU by finding a perfect memory controller placement. Therefore, the Intel SCC was chosen as a famous example for such processors. It has been introduced in more detail and explained why the position of a memory controller can lead to performance bottlenecks. This CPU is the base for all modeling decisions and optimizations found in this thesis.

The main part was about the pipelined merge sort algorithm. The application of this algorithm on the Intel SCC and its optimization were explained in detail. In addition to that two other commonly used algorithms - a special form of a MapReduce algorithm and a mesh communication based algorithm - were used to demonstrate the applicability of the introduced optimization means to different examples.

This thesis has proven that for the Intel SCC a placement of memory controllers can be found that is better than the original one. The positive outcome of adding more memory controllers was shown as well.

The methods described here are still improvable with respect to the annealing implementation and the Gurobi parameter tuning. However, these improvements were not part of this thesis and could be done in future work. Once this is done other more sophisticated parallel algorithms can be further investigated to strengthen the findings of this work. However, there is still a chance for other important algorithms to benefit from the Intel memory controller design or even different ones than found here.

But until this is done, the improved memory controller configuration is the best known design optimization of the Intel SCC. As already stated in the introduction, the decision of Intel to place the memory controllers there where they are now is questionable. This work revealed that for certain algorithms the decision is not perfect. This work has also shown that more memory controllers are helping in further improving the performance. But more memory controllers lead to higher costs of the die and possibly higher temperatures. So it is up to the market and engineers to let this be realized or not.

An interesting aspect that was not part of this work either is the impact of different tile structures. This is also regarding the chapter 6.2 where only a 4x6 mesh that can be mapped perfectly to the Intel SCC was tested. One such structure that is likely to be seen in future is a local network of several many-core units in a single consumer system. They might be connected by a slower (compared against the on-chip network) but still fast bus or hierarchical routed network. The impact of such computing structures on typical applications of the real world would be interesting to see in advance.

References

[1] Intel labs, "Intel SCC platform overview", http://communities.intel.com/docs/DOC-5512, 2010

[2] Wikipedia foundation, "Moores law", http://en.wikipedia.org/wiki/Moore%27s_law

[3] Nicolas Melot, Christoph Kessler, Kenan Avdic, Patrick Cichowski and Jörg Keller, "Engineering parallel sorting for the Intel SCC, http://www.fernuni-hagen.de/imperia/md/content/fakultaetfuermathematikundinformatik/pv/wepa-2012-amkk.pdf, 2012

[4] Kenan Avdic, Nicolas Melot, Jörg Keller and Christoph Kessler, "Parallel sorting on Intel Single-Chip Cloud computer", http://www.fernuni-hagen.de/imperia/md/images/fakultaetmathematikundinformatik/pv/paper-11-04-avdic_isca_2011.pdf, 11/2011

[5] Nicolas Melot, Kenan Avdic, Christoph Kessler and Jörg Keller, "Investigation of Main Memory Bandwidth on Intel Single-Chip Cloud Computer", http://www.fernuni-hagen.de/imperia/md/images/fakultaetmathematikundinformatik/pv/paper-11-07-marc3-avdic-melot-kessler-keller.pdf, 07/2011

[6] Rong Chen, Haibo Chen and Binyu Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-parallel Applications on Multicore with Tiling", http://ipads.se.sjtu.edu.cn/lib/exe/fetch.php?media=publications:ostrich-pact10.pdf

[7] Gurobi documentation, http://www.gurobi.com

[8] AMPL documentation, http://www.ampl.com

[9] Ron Wilson, "From Multicore to Many-Core: Architectures and Lessons", http://www.altera.com/technology/system-design/articles/2012/multicore-many-core.html, 2012

[10] Forum thread: "Questions about the BMC of SCC", http://communities.intel.com/thread/24488, 09/2011

[11] Ryan Shrout, "Intel Shows 48-core x86 Processor as Single-chip Cloud Computer", http://www.pcper.com/reviews/Processors/Intel-Shows-48-core-x86-Processor-Single-chip-Cloud-Computer, 12/2009

Abbreviations

ILP - Integer linear programming, chapter 3.3

Intel SCC - Intel Single-chip cloud computer, chapter 2

MC - memory controller, chapter 2

MIP - Mixed integer programming, chapter 6.2.2

NUMA - Non-uniform memory access, chapter 1

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich meine Abschlussarbeit „Optimization of cluster on-chip architectures“ selbstständig und ohne fremde Hilfe angefertigt habe, und dass ich alle von anderen Autoren wörtlich übernommenen Stellen wie auch die sich an die Gedankengänge anderer Autoren eng anlegenden Ausführungen meiner Arbeit besonders gekennzeichnet und die Quellen zitiert habe.

Ort, Datum: Unterschrift: