# Hierarchical Image Computation with Dynamic Conjunction Scheduling

Christoph Meinel
FB Informatik, University of Trier
meinel@uni-trier.de

Christian Stangier
FB Informatik, University of Trier
stangier@uni-trier.de

## Abstract

*Image computation is the core operation for optimization and formal verification of sequential systems like controllers or protocols. State exploration techniques based on OBDDs use a partitioned representation of the transition relation to keep the OBDD-sizes manageable. This paper presents algorithms for building a hierarchically partitioned transition relation and conjunction scheduling based on this partitioning. The conjunction scheduling algorithm allows to dynamically reorder partitions and is targeted to improve the AndExist operation. Model checking experiments prove the effectiveness of the new algorithms.*

## 1. Introduction

The computation of the reachable states (RS) of a finite state machine (FSM) is an important task for synthesis, logic optimization and formal verification. The increasing complexity of sequential systems like controllers or protocols requires efficient RS computation methods. If the RS are computed by using Ordered Binary Decision Diagrams (OBDDs) [2], the system under consideration is represented in terms of a transition relation (TR). Since the monolithic representation of the circuit's TR usually leads to unmanageable large OBDD-sizes, the TR has to be partitioned [3, 6]. The quality of the partitioning is crucial for the efficiency of the RS computation. The computation of transitions will be unnecessarily time consuming, if the TR is divided into too many parts. On the other hand a number of partitions that is too small will lead to a blow-up of OBDD-size and hence, memory consumption.

The standard method is to sort the latches according to a benefit heuristic [7, 13] and then, apply a clustering algorithm. This clustering algorithm follows a greedy scheme [5] that is guided only by OBDD-size, i.e if the OBDD-size of a partition is exceeding a certain threshold a new partition has to be created.

Recently, new approaches for partitioning of the transition relation have been published: [9] presents a heuristic that minimizes *active lifetime* of the variables to gain a good conjunction schedule computed from a dependency matrix. Additionally the authors give a blocking strategy for the clustering. But, this method is restricted to forward image computation. In [11] and [12] heuristics are presented that focus on grouping related variables to increase the quality of the partitioning, clustering takes place only within the given groups. The groups are determined from RTL descriptions resp. from a dependency matrix.

In this paper we extend the method of [11] to produce a real hierarchical partitioning of the transition relation. The hierarchical image computation is completed by an algorithm that performs the AndExist operation on the tree-like partitions that result from the hierarchical partitioning algorithm. The main impact comes from the heuristic that solves the problem of ordering the clusters for conjunction. It emerged that the AndExist algorithm works very well with a hierarchical partitioning and that this heuristic optimizes the performance of the AndExist. Additionally, the scheduling heuristic allows a true dynamic rescheduling of the partitions.

## 2. Preliminaries

Modern complex designs require a structured hierarchical description to be feasible. Often they are written in a hardware description language (HDL) at register transfer level (RTL). The term RTL is used for an HDL description style that utilizes a combination of *data flow* and *behavioral constructs*. Logic synthesis tools take the RTL HDL description to produce an optimized gate level netlist and high level synthesis tools at the behavioral level output RTL HDL descriptions. Verilog [15] and VHDL [8] are the most popular HDLs used for describing the functionality at RTL.

The design methodology in Verilog is a top down hierarchical modeling concept based on modules, which are the basic building block. Our experimental work is based on designs written in this language, but this approach can be easily extended to any hierarchical finite state machine representation as it is e.g. provided by state space decomposition algorithms (see. e.g. [10]).

### 2.1 Partitioned Transition Relations

The computation of the RS is a core task for optimization and verification of sequential systems. The essential part of OBDD-based traversal techniques is the transition relation TR:

$$\mathrm{TR}(x, y, e) = \prod_i \delta_i(x_i, e) \equiv y_i,$$

which is the conjunction of the transition relations of all latches ($\delta_i$ denotes the transition function of the $i$th latch).

This *monolithic* TR is represented as a single OBDD and usually is much too large to allow an efficient computation of the RS. Sometimes a monolithic TR is even too large for a representation with OBDDs. Therefore, more sophisticated RS computation methods make use of a *partitioned* TR [3], i.e. a cluster of OBDDs each of them representing the TR of a group of latches. A transition relation partitioned over sets of latches $P_1, \ldots, P_j$ can be described as follows:

$$\text{TR}(x, y, e) = \prod_j \prod_{i \in P_j} \delta_i(x_i, e) \equiv y_i.$$

## 2.2 Image Computation using AndExist

The RS computation consists of repeated image computations $Img(\text{TR}, R)$ of a set of already reached states $R$:

$$Img(\text{TR}, R) = \exists_{x,e}(\text{TR}(x, y, e) \cdot R)$$

With the use of a partitioned TR the image computation can be iterated over $P_j$ and the $\exists$ operation can be applied during the product computation *(early quantification)*. The so called *AndExist* [3] or *AndAbstract* operation performs the AND operation on two functions (here partitions) while simultaneously applying existential quantification ($\exists_x f = f_{x=1} \vee f_{x=0}$) on a given set of variables, i.e the variables that are not in the support of the remaining partitions. Unlike the conventional AND operation the AndExist operation only has a exponential upper bound for the size of the resulting OBDD, but for many practical applications it prevents a blow-up of OBDD-size during the image computation.

Since the number of quantified variables depends on the order in which the partitions are processed, finding an optimal order of the partitions for the AndExist operation is an important problem. We refer to this problem as the *conjunction scheduling problem*. Geist and Beer [7] presented a heuristic for scheduling of partitions each representing a single state variable. More sophisticated heuristics for partitions with several variables are given by [13, 9].

# 3. Hierarchical Partitioning of Transition Relations

In [11] a partitioning heuristic that utilizes hierarchical information – i.e. RTL modules of a Verilog description – was presented. The main idea of this work was to have few groups consisting of the main modules of a design (e.g. sender and receiver or two CPUs and a cache) and to put the latches of the FSM in the according groups. This keeps closely related variables in one group. Also, the groups are separated, this means clustering takes place only within the groups.

The positive effect of this heuristic on the partitioning is best described by a *cluster dependency matrix* (CDM). Entry $(i, j)$ denotes the number of variables that cluster $i$ and cluster $j$ share. By using the RTL method the CDM of the design becomes much sparser and the entries are smaller compared to the standard method [13]. Sparseness in a CDM means easier to perform AndExist operations and smaller entries in the CDM generally result in smaller OBDDs, as fewer variables are involved in the AndExist operation.

Although the RTL method utilizes hierarchical information it produces a kind of *flat* clustering as only the top level of the hierarchy is taken under consideration. The intention for this was not to produce a partitioning that consists of too many very small clusters that might have a bad performance.

The heuristic that we describe in the following extends the RTL method to use the whole given hierarchical structure.

As in the RTL method the hierarchical information is observed from the module structure given in the RT level description of Verilog designs. The heuristic is not restricted to RTL, but any method that detects hierarchical modules or FSMs in a design is suitable. RTL Verilog has just been chosen for ease of understanding and portability.

The main idea of the hierarchical partitioning is to take a complete tree of FSMs and subFMSs (see. Figure 1) and produce a partitioning based on this tree. The partitioning algorithm is recursive and consists of two steps:

1. The modules own latches are clustered, following the conventional scheme, i.e add latches to a cluster until a given threshold (*cluster-threshold*) for the OBDD size of the cluster is exceeded
2. Call the procedure recursively for all submodules of the module.

The result of this partitioning is outlined in Figure 1.

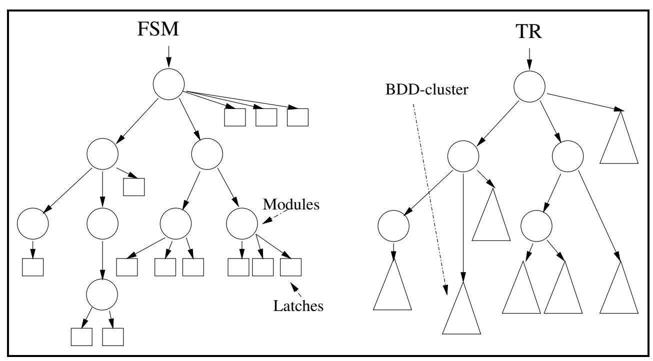



Figure 1: Hierarchical FSM and Transition Relation.

The effect of this strategy on the partitioning is the following: Smaller and less complex submodules that have a small TR will result in a small OBDD, nevertheless this OBDD is isolated from the other submodules and does not interfere with other parts of the partitioning. Larger and more complex submodules will have in addition to their own submodules a cluster of OBDDs representing the more complex TR. We can see this strategy as a more *natural* partitioning that reflects the intention of the designer.

One of the major benefits of this heuristic is that we are able to reduce the influence of the cluster-threshold resulting in a more robust partitioning. For comparison, when using the IWLS95 method we face a *butterfly effect* i.e. small changes in the cluster-size result in a large influence in the performance of the method (positive as well as negative).

The influence of the cluster-threshold has now been reduced to the clustering of a single (sub)module. But, we can reduce the influence even further: We introduced a *preclustering* step, where latches representing a multivalued register are clustered separately. Each multivalued register results in one or more clusters that are passed to the standard clustering routine described above. The impact of this preclustering was so evident that we increased the cluster-threshold for this step of the computation by a factor of two to allow more latches of a multivalued register to stay in one cluster. For comparison: increasing the standard cluster-threshold size of the IWLS method leads to a much poorer performance. See Figure 2 for a sketch of the clustering algorithm.

```
HierarchicalCluster(module,threshold){
  /* First, cluster the modules own latches */
  mv_relations =
    preclusterMVlatches(module→latches,threshold*2 );
  latch_cluster =
    CreateClusters(mv_relations,threshold);

  append(cluster_array,latch_cluster);
  /* Then, cluster the children of the module */
  ForEachItem(module→children, child){
    child_cluster =
      HierarchicalCluster(child,threshold);
    append(cluster_array,child_cluster);
  }
  return cluster_array;
}
```

Figure 2: Algorithm in pseudocode for hierarchical partitioning.

The benefits of the hierarchical partitioning heuristic are:

- We gain a less arbitrary and more structured partitioned transition relation.
- The partitioning method is more robust, i.e. the cluster-threshold can be widely extended to increase performance for larger designs.
- The heuristic performs excellently for structured design.
- The heuristic is applicable to forward and backward image computation.

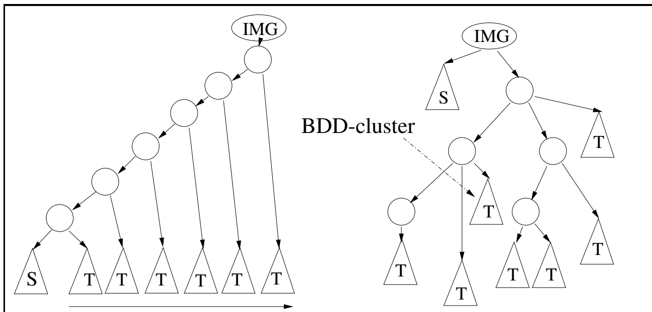
But one problem remains: The heuristic is not able to produce a schedule for conjunction of clusters during the AndExist operation. Also, it seems unlikely that conjunction scheduling heuristics like [13, 9] improve the performance of this heuristics since their ordering strategies conflict with the grouping paradigm of this method.

# 4. Hierarchical Image Computation

In the following we will present algorithms to complete our framework for hierarchical image computation. The result of the algorithm *HierarchicalCluster* is a (linear) list of clusters that are not ordered (see. Figure 3a). This type of linearly arranged clusters is the same that we get from other partitioning algorithms (e.g. [13]). The image of a certain state set (represented by the OBDD $S$) is obtained by consecutively applying AndExist (◯) to the OBDDs ($T$) representing the transition relation. This algorithm is called “LinearAndSmooth”.

On the other hand, from *HierarchicalCluster* we obtain a basic ordering of clusters that are local to a certain (sub)module, and this is not adequately represented by a linear list.

Up to now no order for the processing of the submodules of a module has been computed. It is reasonable to think of an ordering for these submodules, since there is no way to detect an efficient schedule for processing from hierarchical description.



a) Linear Image Computation b) Hierarchical Image Computation

Figure 3: Linear and Hierarchical Image Computation.

## 4.1 General Algorithm

The algorithm outlined in Figure 4 describes the general way to compute an image hierarchically. To allow hierarchical image computation *HierarchicalCluster* has to be modified. The clusters are no longer put in a list, but stored in their according module (see. Figure 3b).

*HierarchicalAndSmooth* computes the image recursively in *preorder* style, i.e. first the module’s local clusters are conjuncted in the temporary product, then the computation continues with the submodules.

The preorder computation introduces the modules’ control variables first, resulting in an increase in the number of variables during the AndExist. On the other hand after finishing a submodule all variables that control only this module are quantified out, resulting in a decrease in OBDD-size.

```
HierarchicalAndSmooth(fromSet,module){
    product = fromSet;
    ForEachItem(img→cluster,cluster){
        smoothVars = ComputeSmoothVars(module,cluster);
        if (smoothVars)
            tmpProduct = bdd_and_smooth(product,cluster,
                    smoothVars);
        else
            tmpProduct = bdd_and(product, cluster);
        product = tmpProduct;
    }
    childrenreamining = module→children;
    while(childrenremaining){
        child = ChooseBestSubmodule(childrenremaining);
        tmpProduct =
            HierarchicalAndSmooth(product,child);
        product = tmpProduct;
        remove_from(childrenremaining,child);
    }
    return product;
}
```

Figure 4: Algorithm in Pseudocode for Hierarchical Image Computation.

## 4.2 Dynamic Reordering of the Conjunction Schedule

The conjunction schedule for the image computation is determined in the *HierarchicalAndSmooth* by *ChooseBestSubmodule*, which can be computed statically or dynamically (the simplest solution would be the list order). Ordering heuristics like [13, 9] may be applied as well, but they are not useful for dynamic rescheduling as they only take structural information of the transition relation into account and will always result in the same schedule. Nevertheless, adjusting the conjunction schedule to changing state sets, OBDD-sizes, or variable orders might be very profitable.

We describe a strategy to improve the performance of the AndExist operation twofold: The AndExist operation generally profits from a hierarchical partitioning. And, we can use the hierarchy structure to improve the conjunction schedule dynamically.

The AndExist operation profits from a *compact* cube of smooth variables. The cube of smooth variables describes the set of variables that are quantified out during the AndExist operation. We call this cube compact, if the variables that appear in the cube are adjacent and not spread over the variable order of the OBDD. During a step of the AndExist recursion the following three cases are possible:

1. The current variable is contained in the smooth variable set: Then the recursion continues and the two resulting OBDDS are combined by an OR operation.
2. The current variable is not contained in the smooth variable set: The result is a new node labeled with the current index and whose successors are the results of the two recursions.
3. The cube has reached the sink node: The recursion reduces to an AND operation.

If the smooth variable cube is compact the third case appears earlier, improving the efficiency of the operation. And, if the clusters are separated, i.e. do not share many variables, then the third case may reduce to an identity function, because the cube and the cluster reach the sink node simultaneously.

This leads us to the following strategy for *ChooseBestSubmodule*:

1. Compute the maximum level (maxlevel) in the OBDD of a variable to be quantified out in all clusters and submodules of a given submodule.
2. Choose the submodules of the current module in increasing order of their maxlevels.

This strategy gives us a good schedule as we expect from the hierarchical partitioning that the clusters of the modules have highly separated variable sets resulting in compact cubes. Also, the schedule is changed dynamically as the variable order changes during the computation as a result of increasing state sets etc.

## 5. Experiments

We implemented our algorithms in the VIS-package [5] (version 1.3) using the underlying CUDD-package [14] (version 2.3.0). VIS is a popular verification and synthesis package in academic research. It inherits state of the art techniques for OBDD manipulation, image and reachable states computation as well as formal verification techniques. Together with the vl2mv translator VIS provides a Verilog front-end.

### 5.1 Benchmarks

For our experiments we used Verilog designs from the Texas97 benchmark suite [1]. This publicly available benchmark suite contains real life designs from industry and academics including: MSI Cache Coherence Protocol, PCI Local BUS, PI BUS Protocol, MESI Cache Coherence Protocol, MPEG System Decoder, DLX and PowerPC 60x Bus Interface. The benchmark suite also contains properties given in CTL formulae for verification.

We chose those designs that represent RTL (i.e. including more than one module) rather than gate level descriptions. Only those designs were considered that could be read in and whose transition relation could be build respecting our system limitations. Table 1 shows 32 different benchmarks for which one or two sets of properties have been checked (resulting in 54 experiments). The runtime heavily depends on the chosen set of properties to be checked and is not proportional to the number of image computations. Therefore it is reasonable to check more than one set of properties. Some very small examples (CPU time $< 10$s) are not shown.

## 5.2 Experimental Setup

We left all parameters of VIS and CUDD unchanged. (Partition cluster size = 5000, partition method for MDDs = inout, OBDD variable reordering method = sifting, first reordering threshold = 4004 nodes) The model checking was preceeded by a forced variable reordering. The CPU time was limited to 6 CPU hours and memory usage was limited to 200MB. All experiments were performed on Linux PentiumIII 500Mhz workstations.

## 5.3 Results

We compare our method (Hierarchy) to the standard method (IWLS95). For results on runtime and space requirements see Table 1. Icmp is the sum of forward and backward image computations performed during the analysis. Parts gives the number of partitions of the transition relation. The OBDD-size of the transition relation cluster and the peak number of live nodes is given by TRn resp. Peakn. The CPU time is measured in seconds and given as Time. The columns denoted with % describe the improvement in percent[1].

At the bottom of Table 1 you can find the sum of all numbers of partitions, BDD-sizes and CPU-times. Also, the *total improvement* is given.

The experiments show significant improvements in time and space: The overall CPU time could be reduced to 1/4 of the original CPU time (11h instead of 45h). The hierarchical method outperforms the standard method in 51 of the 54 benchmarks. The decrease in computation time ranges up to 97%. The OBDD peak sizes could be lowered by 59% overall (20 million nodes instead of 50 million) Interestingly, the average OBDD size of a cluster reduced from 2464 nodes to 1761 nodes, although the threshold was doubled for multivalued registers. The overall number of clusters remains unchanged.

The effort for variable reordering during symbolic model checking is usually quite high, using the hierarchy method we were able to reduce, beyond all time improvements, the time fraction spent for variable reordering from 58% to 54% (overall).

## Conclusion

We have presented algorithms for partitioning of transition relations and conjunction scheduling. The partitioning algorithm uses hierarchical information to produce a tree-like clustered transition relation. We used RTL information given in Verilog, but any other algorithm that detects submodules of a FSM would work as well. The main impact is due to the algorithm that performs image computation based on this tree-like partitioning. This algorithm allows a dynamic rescheduling of the clusters, allowing to fine-tune the AndExist operation for a hierarchical partitioning. These algorithms resulted in significant reductions in CPU-time and space.

The presented strategy for rescheduling only stands exemplarily for a wide variety of possible heuristics that may be implemented on the basis of the hierarchical partitioning, e.g. a history function that detects "expensive" AndExist operations and schedules them to a more suitable position.

**Acknowledgement**

The authors would like to thank Jim Kukula for fruitful discussions and valuable ideas.

[1] $0 <$ improvement $< 100$; $-100 <$ impairment $< 0$.

# References

[1] A. Aziz et. al., *Texas-97 benchmarks*,http://www-cad.EECS.Berkeley.EDU/Respep/Research/Vis/texas-97.

[2] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, C-35, 1986.

[3] J. R. Burch, E. M. Clarke and D. E. Long, *Symbolic Model Checking with partitioned transition relations*, Proc. of Int. Conf. on VLSI, 1991.

[4] J. R. Burch, E. M. Clarke, D. L. Dill, L. J. Hwang and K. L. McMillan, *Symbolic model checking:* $10^{20}$ *states and beyond*, Proc. of Logic in Computer Science (LICS'90), 1990.

[5] R. K. Brayton, G. D. Hachtel et.al., *VIS: A System for Verification and Synthesis*, Proc. of Computer Aided Verification (CAV'96), 1996.

[6] O. Coudert, C. Berthet and J. C. Madre, *Verification of Synchronous Machines using Symbolic Execution*, Proc. of Workshop on Automatic Verification Methods for Finite State Machines, LNCS 407, Springer, 1989.

[7] D. Geist and I. Beer, *Efficient Model Checking by Automated Ordering of Transition Relation Partitions*, Proc. of Computer Aided Verification (CAV'94), 1994.

[8] R. D. M. Hunter and T. T. Johnson, *Introduction to VHDL*, Chapman & Hall, 1996.

[9] I. Moon, G. D. Hachtel and F. Somenzi , *Border-Block Triangular Form and Conjunction Schedule in Image Computation*, Proc. of Formal Methods in CAD (FMCAD'00), LNCS 1954, 2000.

[10] I. Moon, J. Jang, G. D. Hachtel, F. Somenzi, J. Yuan and C. Pixley, *Approximate Reachability Don't cares for CTL Model Checking*, Proc. of International Conference on CAD (ICCAD'98), 1998.

[11] Ch. Meinel and C. Stangier, *Speeding Up Image Computation by using RTL Information*, Proc. of Formal Methods in CAD (FMCAD'00), LNCS 1954, 2000.

[12] Ch. Meinel and C. Stangier, *A New Partitioning Scheme for Improvement of Image Compuation*, Proc. of ASP Design Automation Conference (ASPDAC'01), 2001.

[13] R. K. Ranjan, A. Aziz, R. K. Brayton, C. Pixley and B. Plessier, *Efficient BDD Algorithms for Synthesizing and Verifying Finite State Machines*, Proc. of Int. Workshop on Logic Synthesis (IWLS'95), 1995.

[14] F. Somenzi, *CUDD: CU Decision Diagram Package*, ftp://vlsi.colorado.edu/pub/ .

[15] D. E. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, 1991.

| | | IWLS95 | | | | Hierarchy | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Icmp | Peakn | Parts | TRn | Time | Peakn | % | Parts | % | TRn | % | Time | % |
| ONE.pixley_cpu | 113 | 24045 | 3 | 4456 | 12 | 29798 | -19 | 4 | -25 | 3220 | 27 | 13 | -7 |
| PCIabnorm.PCI | 304 | 176276 | 14 | 28613 | 253 | 116345 | 33 | 10 | 28 | 18061 | 36 | 152 | 39 |
| PCInorm.PCI | 206 | 81123 | 15 | 35124 | 56 | 69291 | 14 | 11 | 26 | 16993 | 51 | 47 | 16 |
| TWO.PPCliveness | 74 | 263756 | 8 | 13118 | 303 | 2083768 | -87 | 10 | -19 | 13881 | -5 | 3436 | -91 |
| TWO.contention | 37 | 97622 | 7 | 11865 | 47 | 215686 | -54 | 10 | -30 | 10508 | 11 | 150 | -68 |
| multi_main.multim | 45 | 38694 | 5 | 14700 | 34 | 33423 | 13 | 6 | -16 | 1578 | 89 | 18 | 47 |
| p62_LS_LS_V01.ccp | 64 | 166074 | 23 | 49952 | 200 | 124430 | 25 | 23 | 0 | 41246 | 17 | 84 | 58 |
| p62_LS_LS_V01.p6l | 99 | 452267 | 23 | 49952 | 831 | 158021 | 65 | 23 | 0 | 41246 | 17 | 173 | 79 |
| p62_LS_LS_V02.ccp | 54 | 146494 | 22 | 59487 | 105 | 117628 | 19 | 23 | -4 | 42463 | 28 | 70 | 33 |
| p62_LS_LS_V02.p6l | 97 | 167454 | 22 | 59487 | 174 | 117628 | 29 | 23 | -4 | 42463 | 28 | 79 | 54 |
| p62_LS_L_V01.cc | 64 | 176540 | 23 | 49684 | 210 | 132128 | 25 | 22 | 4 | 41091 | 17 | 103 | 50 |
| p62_LS_L_V01.p6li | 99 | 1617162 | 23 | 49684 | 3511 | 183674 | 88 | 22 | 4 | 41091 | 17 | 252 | 92 |
| p62_LS_L_V02.ccp | 54 | 148560 | 23 | 62140 | 106 | 91257 | 38 | 23 | 0 | 37319 | 39 | 74 | 30 |
| p62_LS_L_V02.p6li | 89 | 183811 | 23 | 62140 | 193 | 91257 | 50 | 23 | 0 | 37319 | 39 | 76 | 60 |
| p62_LS_S_V01.ccp | 64 | 176540 | 23 | 49684 | 210 | 132128 | 25 | 22 | 4 | 41091 | 17 | 103 | 50 |
| p62_LS_S_V01.p6li | 99 | 1614473 | 23 | 49684 | 3601 | 183674 | 88 | 22 | 4 | 41091 | 17 | 260 | 92 |
| p62_LS_S_V02.ccp | 54 | 148560 | 23 | 62140 | 106 | 91257 | 38 | 23 | 0 | 37319 | 39 | 74 | 30 |
| p62_LS_S_V02.p6li | 89 | 183811 | 23 | 62140 | 193 | 91257 | 50 | 23 | 0 | 37319 | 39 | 74 | 61 |
| p62_L_L_V01.ccp | 52 | 164244 | 23 | 48961 | 189 | 117269 | 28 | 23 | 0 | 41165 | 15 | 78 | 58 |
| p62_L_L_V01.p6liv | 89 | 477543 | 23 | 48961 | 934 | 189172 | 60 | 23 | 0 | 41165 | 15 | 159 | 82 |
| p62_L_L_V02.ccp | 53 | 144504 | 23 | 48971 | 172 | 119677 | 17 | 23 | 0 | 41992 | 14 | 71 | 58 |
| p62_L_L_V02.p6liv | 96 | 242452 | 23 | 48971 | 377 | 119677 | 50 | 23 | 0 | 41992 | 14 | 90 | 76 |
| p62_L_S_V01.ccp | 75 | 168782 | 22 | 62479 | 121 | 123551 | 26 | 23 | -4 | 42294 | 32 | 93 | 23 |
| p62_L_S_V01.p6liv | 118 | 192410 | 22 | 62479 | 231 | 137714 | 28 | 23 | -4 | 42294 | 32 | 166 | 28 |
| p62_L_S_V02.ccp | 55 | 140767 | 22 | 57365 | 104 | 98699 | 29 | 22 | 0 | 40454 | 29 | 73 | 29 |
| p62_L_S_V02.p6liv | 96 | 140767 | 22 | 57365 | 106 | 98699 | 29 | 22 | 0 | 40454 | 29 | 74 | 30 |
| p62_ND_LS_V01.ccp | 83 | 396642 | 24 | 63506 | 830 | 299289 | 24 | 24 | 0 | 46550 | 26 | 559 | 32 |
| p62_ND_LS_V01.p6l | 128 | 5583160 | 24 | 63506 | 21039 | 1747044 | 68 | 24 | 0 | 46550 | 26 | 4544 | 78 |
| p62_ND_LS_V02.ccp | 103 | 191386 | 22 | 63321 | 331 | 156783 | 18 | 23 | -4 | 44262 | 30 | 175 | 47 |
| p62_ND_LS_V02.p6l | 192 | 1564426 | 22 | 63321 | 3611 | 445241 | 71 | 23 | -4 | 44262 | 30 | 669 | 81 |
| p62_ND_L_V01.ccp | 75 | 356794 | 25 | 65964 | 781 | 380121 | -6 | 24 | 4 | 45076 | 31 | 577 | 26 |
| p62_ND_L_V02.ccp | 161 | 5614430 | 23 | 60383 | timeout | 1352990 | 75 | 23 | 0 | 47981 | 20 | 3050 | 85 |
| p62_ND_L_V02.p6li | 200 | 5573568 | 23 | 60383 | timeout | 3009524 | 46 | 23 | 0 | 47981 | 20 | 4965 | 77 |
| p62_ND_S_V02.ccp | 84 | 150630 | 23 | 46744 | 188 | 133586 | 11 | 24 | -4 | 41048 | 12 | 142 | 24 |
| p62_ND_S_V02.p6li | 177 | 645917 | 23 | 46744 | 1231 | 164486 | 74 | 24 | -4 | 41048 | 12 | 200 | 83 |
| p62_S_S_V01.ccp | 43 | 147063 | 23 | 62209 | 101 | 97360 | 33 | 23 | 0 | 39901 | 35 | 62 | 38 |
| p62_S_S_V01.p6liv | 80 | 153012 | 23 | 62209 | 106 | 97360 | 36 | 23 | 0 | 39901 | 35 | 67 | 36 |
| p62_S_S_V02.ccp | 37 | 129492 | 23 | 54800 | 94 | 90710 | 29 | 23 | 0 | 39819 | 27 | 60 | 36 |
| p62_S_S_V02.p6liv | 74 | 129492 | 23 | 54800 | 95 | 90710 | 29 | 23 | 0 | 39819 | 27 | 61 | 35 |
| p62_V_LS_V01.ccp | 108 | 283494 | 24 | 58415 | 587 | 210147 | 25 | 23 | 4 | 45928 | 21 | 362 | 38 |
| p62_V_LS_V01.p6li | 153 | 4483034 | 24 | 58415 | timeout | 2126524 | 52 | 23 | 4 | 45928 | 21 | 6236 | 71 |
| p62_V_LS_V02.ccp | 90 | 165200 | 23 | 52073 | 221 | 128016 | 22 | 22 | 4 | 41985 | 19 | 148 | 33 |
| p62_V_LS_V02.p6li | 178 | 1059895 | 23 | 52073 | 1864 | 224255 | 78 | 22 | 4 | 41985 | 19 | 286 | 84 |
| p62_V_S_V01.ccp | 82 | 213245 | 23 | 61795 | 245 | 142930 | 32 | 23 | 0 | 43513 | 29 | 173 | 29 |
| p62_V_S_V01.p6liv | 127 | 964988 | 23 | 61795 | 2168 | 442596 | 54 | 23 | 0 | 43513 | 29 | 890 | 58 |
| p62_V_S_V02.ccp | 84 | 163439 | 22 | 54807 | 189 | 126542 | 22 | 23 | -4 | 41969 | 23 | 120 | 36 |
| p62_V_S_V02.p6liv | 177 | 351553 | 22 | 54807 | 475 | 228621 | 34 | 23 | -4 | 41969 | 23 | 241 | 49 |
| packet.packet | 65326 | 53790 | 3 | 9704 | 5122 | 68473 | -21 | 4 | -25 | 4742 | 51 | 5068 | 1 |
| single_main | 108 | 14936 | 2 | 6352 | 13 | 9360 | 37 | 4 | -50 | 884 | 86 | 8 | 38 |
| single_main.1 | 52 | 14936 | 2 | 6352 | 13 | 9360 | 37 | 4 | -50 | 884 | 86 | 7 | 46 |
| three_processor.p | 244 | 4621235 | 9 | 19750 | timeout | 3062696 | 33 | 7 | 22 | 4838 | 75 | 2959 | 86 |
| three_processor_bin. | 140 | 8779857 | 7 | 20387 | timeout | 560970 | 93 | 7 | 0 | 5140 | 74 | 522 | 97 |
| two_processor.pro | 264 | 903917 | 4 | 12311 | 676 | 88215 | 90 | 5 | -19 | 1810 | 85 | 72 | 89 |
| two_processor_bin | 141 | 252974 | 4 | 11610 | 150 | 64924 | 74 | 5 | -19 | 2623 | 77 | 42 | 72 |
| Sum: | 70748 | 50497236 | 1022 | 2518138 | 160514 | 20625941 | | 1027 | | 1809018 | | 38307 | |
| Total Improvement: | | | | | | 59% | | 0% | | 28% | | 76% | |

Table 1: Comparison of IWLS95 Method and Hierarchy Heuristic