List of Figures

[16] Yang, T. and Gerasoulis, A. Dominant Sequence Clustering Heuristic Algorithm for Scheduling DAGs on Multiprocessor, Report, Dept. of Computer Science, Rutgers Univ., 1991.

[17] Wu, M. Y. and Gajski, D. A Programming Aid for Hypercube Architectures, *J. of Supercomputing*, 2, (1988), pp. 349-372.

[3] El-Rewini, H. and Lewis, T.G. Scheduling Parallel Program Tasks onto Arbitrary Target Machines, *J. of Parallel and Distributed Computing,* 9, (1990), pp. 138-153.

[4] Gerasoulis, A. and Nelken, I. Static Scheduling for Linear Algebra DAGs, *Proc. of Fourth Conference on Hypercubes*, Monterey, Vol. 1, 1989, pp. 671-674.

[5] Gerasoulis, A., Venugopal, S., and Yang, T. Clustering Task Graphs for Message Passing Architectures, *Proc. of 4th ACM International Conference on Supercomputing*, 1990, pp. 447-456.

[6] Gerasoulis, A. and Yang, T. On the Granularity and Clustering of Directed Acyclic Task Graphs, TR-153, Dept. of Computer Science, Rutgers Univ., 1990.

[7] Girkar, M. and Polychronopoulos, C. Partitioning Programs for Parallel Execution, *Proc. of ACM International Conference on Supercomputing*, St. Malo, France, July 4-8, 1988.

[8] Kasahara, H. and Narita, S. Practical Multi-processor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Trans. on Computers,* C-33, (1984), pp. 1023-1029.

[9] Kim, S. J. A General Approach to Multiprocessor Scheduling, TR-88-04, DCS, Univ. of Texas at Austin, 1988.

[10] Kim, S. J. and Browne, J. C. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures, *International Conference on Parallel Processing*, vol 3, 1988, pp. 1-8.

[11] Kung, S. Y. *VLSI Array Processors*, Prentice Hall, 1988.

[12] Ortega, J. M. *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum, 1988.

[13] Papadimitriou, C. and Yannakakis, M. Towards on an Architecture-Independent Analysis of Parallel Algorithms, *SIAM J. Comput.,* 19, (1990), pp. 322-328.

[14] Sarkar, V. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*, The MIT Press, 1989.

[15] Yang, T. and Gerasoulis, A. A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors, *Proc. of Supercomputing '91*, IEEE, 1991, pp. 633-642.

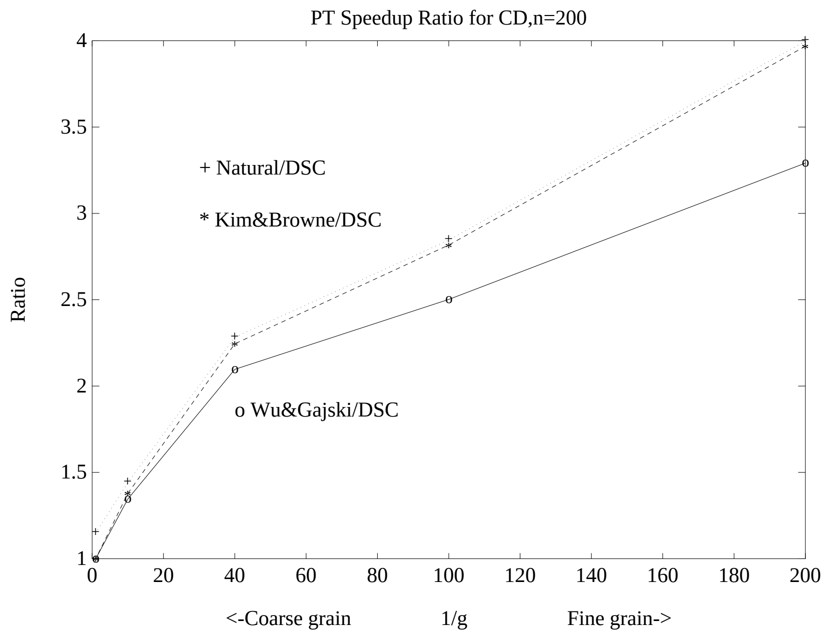
PT Speedup Ratio for CD,n=200
Ratio
+ Natural/DSC
* Kim&Browne/DSC
o Wu&Gajski/DSC
<-Coarse grain
1/g
Fine grain->

Figure 15: DSC vs. other clustering algorithms for the CD DAG.

algorithm that has better performance than DSC with the same computational complexity? How does the performance of DSC compares to higher complexity clustering algorithms that satisfy $T1$ and $T2$ and are optimal for the primitive structures?

9 Acknowledgments

We thank Sesh Venugopal for several useful discussions on this work and Ajay Bakre and referees for reading the manuscript and suggesting improvements in the presentation. This work was in part supported by a Grant No. DMS-8706122 from NSF.

References

[1] Chretienne, Ph. Task Scheduling over Distributed Memory Machines, *Proc. of the International Workshop on Parallel and Distributed Algorithms*, North Holland, 1989.

[2] Chretienne, Ph. Complexity of Tree Scheduling with Interprocessor Communication Delays, Report, M.A.S.I. 90.5, Universite Pierre et Marie Curie, 1990.

task T_k^j represents a vector modification while T_k^k is the pivot operation. The weights are

$$\tau_k^k = (n-k+2)w, \quad \tau_k^j = (2(n-j)+1)w, \quad k = 1:n-1, \quad j = k+1:n$$

$$c_k^j = (n-k+2)\beta, \quad c_k^{k+1} = (n-j+1)\beta, \quad k = 1:n-1, \quad j = k+1:n$$

where τ_k^j are the computation weighs, c_k^j the communication cost between tasks T_k^k and T_k^j, c_k^{k+1} the communication cost between tasks T_k^j and T_{k+1}^j.

In Figure 15, we fix $n = 200$ and vary the granularity of the DAG by increasing the communication constant factors as follows, $\beta = w$, $2w$, $5w$, $40w$, $100w$, $200w$. The number of tasks is about 2000 and edges 4000. We observe that the DSC is superior to all methods for this example particularly when the DAG becomes fine grain. For coarse grain DAGs the DSC algorithm is better at most by a factor of 2. This result is expected, since we have shown in [6] that any linear clustering is within factor of 2 of the optimum. Both Natural and KB/L are linear clusterings, and the clusterings produced by DSC and MCP are also linear.

Figure 15 also verifies our theoretical analysis of the previous section and the importance of zeroing edges in the dominant sequence. MCP and DSC are similar with one major exception that DSC zeroes the edges in DS while MCP does not. Notice that the MCP parallel time is 3 times longer than DSC parallel time, while both natural and KB/L parallel times are within a factor of 4 with DSC when $g = 1/200$. With regards to the computational complexity, DSC is much faster than all others heuristics. Similar results have been derived for other task graphs.

8 Conclusions

We have provided a general framework for comparing clustering algorithms. Guided by this framework we were able to present, evaluate and compare several existing algorithms in a systematic manner. We have demonstrated the importance of having performance features $T1$ and $T2$ in each clustering step. We have shown that every algorithm satisfies $T1$ for any DAG but only DSC satisfies additional performance properties and has a lower complexity. This is why DSC is superior to other algorithms in practice which has been demonstrated in the experimental results.

Several interesting questions remain open for future investigation. Is it possible to develop a clustering

7 Experimental results and comparisons

DSC vs. Sarkar's. In Yang and Gerasoulis [15] we compared the DSC algorithm and Sarkar's algorithm on 100 randomly generated DAGs and weights. These graphs are produced by randomly generating the number of tasks and edges and assigning random numbers as weights. The size of the graphs varies from a minimum average of 70 nodes with 311 edges to a maximum average of 329 nodes with 3430 edges. The ratio of computation and communication varies from 0.8 to 8.7. On the average we found that $PT_{dsc} = .83 PT_{sarkar}$ for these graphs. As far as the algorithm execution speed is concerned, DSC is one order faster than Sarkar's.

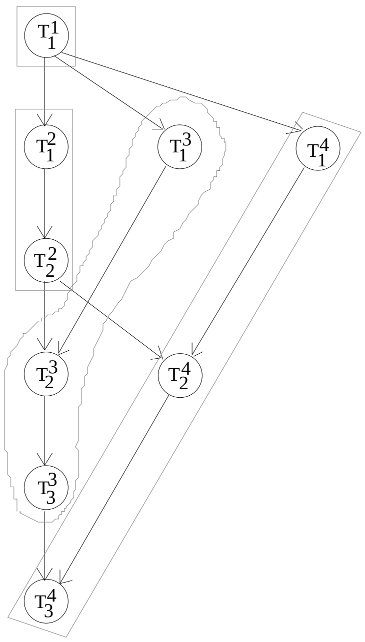

Figure 14: The Cholesky Decomposition task graph and natural linear clustering for n=4.

DSC vs. other clustering heuristics. We have chosen for experimentation the well known Cholesky Decomposition (CD) DAG. There are several reasons for choosing this DAG for comparison. One is that we can compute the clustering of KB/L analytically rather than computationally which will be impossible because of the high complexity. Another reason is that we would like to compare DSC with the *natural clustering* which is a widely used clustering for this DAG [4, 12]. However, we did not include Sarkar's heuristic because the graph is too large to be handled.

In Figure 14, we give the DAG for the special case of $n = 4$, where n is the dimension of the matrix. Each

and $ptrw$ set does not change. If this is not true, assume that (n_1, n_2) is an edge in GCP but not in $ptrw$ which is zeroed in step k such that $i < k < j$. Now since (n_s, n_t) is also in the GCP, the edge (n_1, n_2) must lie either on the root side of (n_s, n_t) or on the leaf side of it. If (n_1, n_2) lies on the leaf side of (n_s, n_t), all the edges in the CP of the subtree including (n_s, n_t) will be zeroed in step k which is a contradiction. If on the other hand, (n_1, n_2) lies on the root side of (n_s, n_t) then it should also be in $ptrw(op_i)$ which is again a contradiction. ■

Proposition 6.1.8 (Sarkar's Algorithm). *Sarkar's algorithm does not satisfy $T2$ and is not optimal for any tree.*

Proof: The simplest counter example is a fork (or join) where Sarkar's does not satisfy $T2$ and is not optimal in general. ■

6.2 A general summary

In Table VI we summarize the characterization of the four algorithms.

Table VI: A comparison of four clustering algorithms.

	Sarkar	**MCP**	**KB/L**	**DSC**
Goals	G1	G1/G2	G4/G1	G1/G2
Transformed goals	Reduce CV	Minimize starting time	Compress CP in subgraphs	Compress DS
Multiplicity of zeroing	One	One	All in CP	Some incoming edges of a free node
Constraint imposed	$T1$	Non increase in ST	Linearity of clustering	$CT1$, or $CT1$& $CT2$
$T1$	Yes	Yes	Yes	Yes
$T2$	No	No	No	No for fine grain Yes for coarse grain
Complexity	$e(v+e)$	$v^2 \log v$	$v(v+e)$	$(e+v)\log v$

Notice that DSC satisfies $T2$ for any coarse grain DAG. The proof is similar to the one given for the out-tree case. The coarse grain constraint assures that only one incoming edge would be zeroed if it would be zeroed.

that it determines the optimum for coarse grain in-trees. ■

Proposition 6.1.7 (KB/L Algorithm). *KB/L is not optimal for any tree and in general does not satisfy $T2$. It only satisfies $T2$ for coarse-grain in/out-trees.*

Proof: To show that KB/L is not optimal we present a counter example in Figure 13. The optimal result derived by DSC is $M_1 = \{n_1, n_3\}$, $M_2 = \{n_2\}$, $M_3 = \{n_4, n_5\}$ with $PT = 13.5$. The result derived by Kim's algorithm is $M_1 = \{n_1, n_3, n_5\}$, $M_2 = \{n_2\}$, $M_3 = \{n_4\}$ with $PT = 14$.

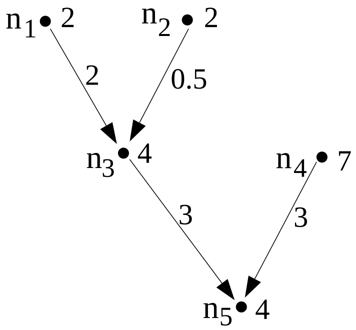

Figure 13: A counter example of an in tree.

We prove that KB/L satisfies $T2$ for a coarse grain in-tree. The proof for an out-tree is similar. We notice that $DS_i = CP_i$ since KB/L is a linear clustering algorithm. Now assume that $(n_s, n_t) \in ptrw(op_i)$. This means that the parallel time is reducible by zeroing this edge. We first show that no incoming edge of n_t has been examined or zeroed at any step less or equal to i. Assume the contrary, which means that one incoming edge has been zeroed and the others, including (n_s, n_t), have been examined. Then zeroing (n_s, n_t) is impossible to reduce the parallel time because the tree is coarse grain. That is a contradiction. Secondly, since KB/L zeroes all edges in the CP of each subgraph at each step and the subgraph is a subtree, we can easily see that all edges in the subtree rooted with n_t are not examined. As a result, when the global CP of the whole tree (called GCP) goes through any unexamined subtree, the part of GCP in this subtree is also the CP of this subtree. And when KB/L is working on the subtree where (n_s, n_t) resides, the edges in its CP including (n_s, n_t) will be zeroed.

Next, we show that one such edge (n_s, n_t) is still in *ptrw* set when that edge is zeroed. If KB/L does not zero any edge in $ptrw(op_i)$ set after step i until step j, then no edge in GCP is zeroed during those steps

Proposition 6.1.5 (DSC Algorithm). *DSC satisfies $T2$ and gives the optimal solution for coarse grain in-trees but not for fine grain in-trees. For both fine and coarse grain out-trees, DSC satisfies $T2$ but it is not optimal.*

Proof: The optimum solution can be derived by DSC for coarse grain in-trees. It is proven by induction on the depth of the tree in Yang and Gerasoulis [15, 16] by showing that DSC produces a schedule where every node has the minimum starting time.

Without inverting an out-tree, DSC cannot get the optimum but it still satisfies the $T2$ property. We prove it in two steps.

In the first step we show that if $ptrw(op_i) \neq \emptyset$, then the edges in $ptrw(op_i)$ must be unexamined. If not, suppose $(n_s, n_t) \in ptrw(op_i)$ and it has been examined and n_t has been scheduled. Zeroing (n_s, n_t) will strictly reduce the parallel time implying that the starting time n_t can be reduced. Since n_t has only one incoming edge (n_s, n_t), if the assumption was true, (n_s, n_t) would have been zeroed when n_t was scheduled, a contradiction.

Next we must prove that (n_s, n_t) will be zeroed at some future step. Assume that the top-most unexamined edge in the current DS, and which also belongs to $ptrw(op_i)$, is (n_s, n_t). From step $i + 1$ to the step when n_t becomes free, the parallel time cannot be reduced and (n_s, n_t) is always in the $ptrw$ set. At the step at which n_t becomes free, it must have the highest priority and then (n_s, n_t) must be zeroed to reduce the starting time of n_t. ■

Proposition 6.1.6 (MCP Algorithm). *MCP satisfies $T2$ and gives the optimal solution for coarse grain in-trees but not for general in-trees. For out-trees, MCP does not satisfy $T2$ and it is not optimal.*

Proof: Since the problem itself is NP-complete, for a general in-trees and out-trees the MCP cannot determine the optimum. Also since for the special case of fork/join MCP does not satisfy $T2$, it does not satisfy $T2$ in general.

The proof that for a coarse grain in-tree, MCP is optimal and satisfies $T2$ is similar to DSC. When the tree has height 1 and is a coarse grain join then MCP determines the optimum. Inductively, we can prove

For a fork set, we present a counter example shown Figure 12. The optimal clustering for this fork, derived by DSC, has parallel time equal to 8 and is given by: $M_1 = \{n_1, n_2, n_3, n_5\}$, $M_2 = \{n_4\}$. MCP derives the following clusters with PT=10: $M_1 = \{n_1, n_2, n_3\}$, $M_2 = \{n_4\}$, $M_4 = \{n_5\}$.

The MCP examines the free nodes in the order of n_3, n_2, n_4, n_5. Moreover, $ptrw(op_0) = \{(n_1, n_5)\}$, but this edge is never zeroed. Thus MCP does not satisfy the $T2$ property. ■

Proposition 6.1.3 (Sarkar's Algorithm). *Sarkar's algorithm does not satisfy $T2$ and is not optimal for any join or fork.*

Proof: We only present a counter example for a fork in Figure 12. The join case is similar. Sarkar's clustering with $PT = 9$ is given below and this is not optimum: $M_1 = \{n_1, n_2, n_4, n_5\}$, $M_2 = \{n_3\}$. Regarding the $T2$ property, we look at the zeroing sequence. At step 1, (n_1, n_5) is zeroed. At step 2, (n_1, n_2) is zeroed. Then $PT_2 = 9$ and $ptrw(op_2) = \{(n_1, n_3)\}$. But this edge is never zeroed. ■

Proposition 6.1.4 (KB/L Algorithm.) *KB/L does not satisfy $T2$ and it is not optimal for an arbitrary join or fork. In the special case of a coarse grain join or fork, KB/L satisfies $T2$ and is optimal.*

Proof: For a join with the root n_x and $\beta_i + \tau_i \geq \beta_{i+1} + \tau_{i+1}$, the CP is (n_x, n_1) and after it is zeroed $PT_1 = \tau_x + max(\tau_1, \beta_2 + \tau_2)$. This is optimum for coarse grain graphs. However, if the graph is not coarse grain, then nonlinear clustering is necessary and KB/L is not optimum. Also since $ptrw(op_1) = (n_x, n_2)$ and this edge is never zeroed it does not satisfy $T2$. For Figure 12 Kim's algorithm gives $PT = 9$ and $M_1 = \{n_2\}$, $M_2 = \{n_3\}$, $M_4 = \{n_4\}$, $M_5 = \{n_1, n_5\}$. The results are similar for a join. ■

6.1.2 Performance on in/out trees

Finding an optimal solution for a tree is still NP-complete as shown by Chretienne [2]. However, when this tree is coarse grain, the optimal solution is computable in polynomial time. This can be shown by using the fact that an optimum schedule can be derived by linear clustering for coarse grain DAGs and then use the special tree structure to determine the optimum linear clustering in polynomial time. As a matter of fact, the DSC algorithm will find the optimum linear clustering for trees.

Table V: Performance of clustering algorithms on primitive structures.

Coarse grain								
	Join		Fork		In-tree		Out-tree	
	$T2$	Opt.	$T2$	Opt.	$T2$	Opt.	$T2$	Opt.
KB/L	Yes	Yes	Yes	Yes	Yes	No	Yes	No
Sarkar	No	No	No	No	No	No	No	No
DSC	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
MCP	Yes	Yes	No	No	Yes	Yes	No	No

Fine grain								
	Join		Fork		In-tree		Out-tree	
	$T2$	opt.	$T2$	opt.	$T2$	opt.	$T2$	opt.
KB/L	No	No	No	No	No	No	No	No
Sarkar	No	No	No	No	No	No	No	No
DSC	Yes	Yes	Yes	Yes	Yes	No	Yes	No
MCP	No	No	No	No	No	No	No	No

timum clustering and also satisfies property $T2$.

Proof: For both join and fork, the DSC algorithm derives the same edge zeroing sequence as the optimal algorithm in section 3.1 which satisfies $T1$ and $T2$, as shown in section 4. ■

Proposition 6.1.2 (MCP Algorithm). *The MCP algorithm does not derive the optimum for arbitrary fork and join graphs and also does not satisfy $T2$. Only for a coarse grain join the MCP determines the optimum and satisfies $T2$.*

Proof: Consider a join set with the root n_x, invert the fork shown in Figure 3 (a), with $\beta_i + \tau_i \geq \beta_{i+1} + \tau_{i+1}$. The MCP only zeroes one incoming edge (n_1, n_x) and the parallel time is $PT_{m+1} = \tau_x + max(\tau_1, \beta_2 + \tau_2)$. If the graph is coarse grain, $\tau_1 \geq \beta_2 + \tau_2$ and MCP finds the optimal solution. However, if the graph is fine grain and zeroing (n_2, n_x) strictly reduces PT_{m+1}, then (n_2, n_x) is in $ptrw(op_{m+1})$. Since MCP does not zero this edge at any step it does not satisfy the $T2$ property and also it does not derive the optimum.

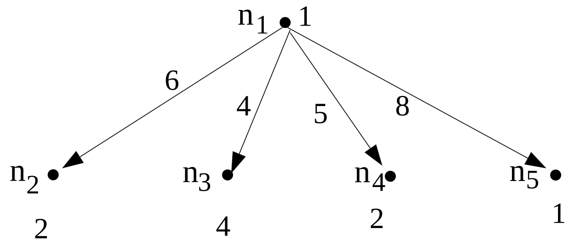

Figure 12: A counter example of a fork.

We first need to clarify our definition of coarse and fine grain task graphs, see for more details [5, 6]. For each join and fork of a task we define

$$g(J_x) = \min_{k=1:m}\{\tau_k\}/\max_{k=1:m}\{c_{k,x}\}, \qquad g(F_x) = \min_{k=1:m}\{\tau_k\}/\max_{k=1:m}\{c_{x,k}\}.$$

The *grain*, g_x, of a task n_x and the *granularity* of a DAG are defined by

$$g_x = \min\{g(F_x), g(J_x)\}, \quad g(G) = \min_{n_x \in V}\{g_x\}.$$

We call a DAG *coarse grain* if $g(G) \geq 1$, otherwise we call it *fine grain.* For coarse grain DAGs each task communicates a small amount of data compared to the computation of its neighbors. Coarse grain graphs possess many interesting properties. For example, the ratio between the parallel time of any linear clustering and that of the optimum is less or equal to $1 + 1/g(G)$, [6]. This implies that for coarse grain graphs we can always be within a factor of two from the optimum by using linear clustering. As a matter of fact the optimum parallel time can be derived by a linear clustering, [6]. Therefore, we can exploit all parallelism in coarse grain graphs by using linear clustering. This is not the case for fine grain graphs where parallel tasks must be sequentialized to minimize the parallel time. If we look at the optimum fork algorithm in Figure 4, if the fork is coarse grain then the algorithm will stop after zeroing only the first edge, otherwise it will continue sequentializing parallel tasks by zeroing more edges.

6.1 Performance on primitive structures

An *in-tree* is a directed tree in which the root has outgoing degree zero and other nodes have outgoing degree one. An *out-tree* is a directed tree in which the root has incoming degree zero and other nodes have incoming degree one. A *join* is an in-tree of depth 1. A *fork* is an out-tree of depth 1.

All algorithms satisfy property $T1$, the monotonic reduction in the parallel time. For other performance features, Table V summarizes the comparative results on the primitive structures.

We now provide the proof for the above performance results.

6.1.1 Performance on Fork and Join

Proposition 6.1.1 (DSC Algorithm) *For the fork and join graphs the DSC algorithm derives the op-*

scheduled in P_0 otherwise its start time will be delayed. Thus n_4 is scheduled in a new processor P_1. Similarly, n_5 is scheduled in P_2. Then at the next step, n_6 will be scheduled in P_1 by zeroing edge (n_4, n_6). At the last step, (n_6, n_7) will be zeroed and the parallel time reduces to 10.5. The resulting clustering is: $M_0 = \{n_1, n_2, n_3\}$, $M_1 = \{n_4, n_6, n_7\}$, $M_2 = \{n_5\}$.

Complexity. Wu and Gajski have given a worst time complexity of $O(v^2 \log v)$ because of the cost in the tie breaking. If there are no ties then the complexity is similar to DSC.

5.5 Clustering Figures

In Figure 11 we summarize the results of three algorithms on our example. The KB/L result is given in Figure 1 (b) with $PT = 11.5$.

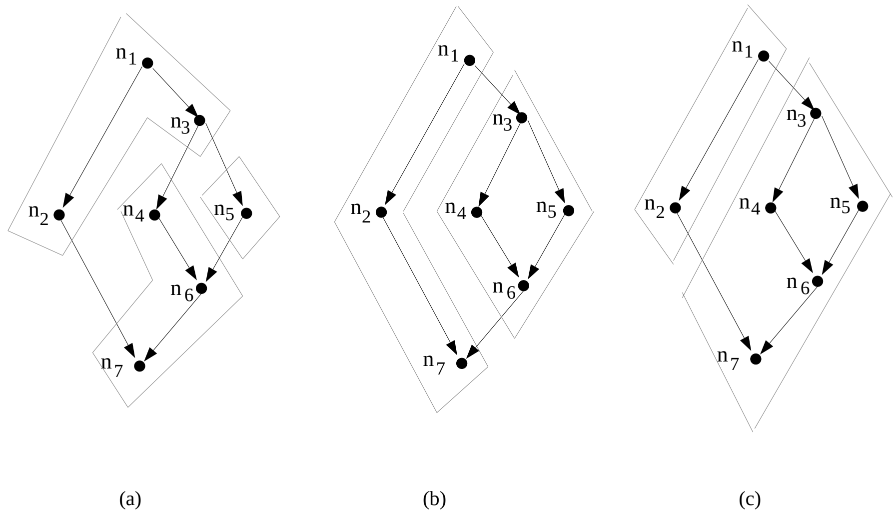

Figure 11: Clusterings of (a) Wu and Gajski's with $PT = 10.5$. (b) Sarkar's with $PT = 10$. (c) DSC with $PT = 9$.

6 Optimality Results for Primitive Structures

In this section, we will study the performance of the previous four algorithms in the clustering of some special primitive structures such as join, fork and coarse grain tree structures. The reason for studying the performance on primitive structures is that a DAG is composed of a set of join and fork nodes, and join and fork trees are spanning trees of a DAG. Therefore studying the performance of clustering algorithms on such structures will further enhance our understanding of their behavior.

CLU_HEU (MCP) : Tasks with the highest priority in the critical path should start execution at the earliest possible time.

SCH_ALG : A task is scheduled either after the last scheduled task of a cluster in SG_{i-1} or it is the first scheduled task in a new cluster.

$domain(op_i)$ = Unexamined edges.

$focus(op_i)$ = Incoming edge of a free node with the highest priority.

$Constraint = CT1$

$zero(op_i)$ = Incoming edge in $focus(op_i)$ if $constraint$ is true.

Termination criterion: All nodes have been scheduled.

Figure 10: Edge zeroing operations in MCP.

time. Since $ST(n_x) = top\_level(n_x, i)$, then reducing the starting time for each task implies $PT_i \leq PT_{i-1}$ and this algorithm satisfies $T1$. The algorithm does not satisfy $T2$ for a general DAG.

Example. We apply this algorithm on the task graph in Figure 1(a). The final clustering is shown in Figure 11(a). The stepwise result is shown in Table IV.

Table IV: Clustering steps of MCP for Figure 1(a).

step i	$domain(op_i)$	$focus(op_i)$	free node	$zero(op_i)$	PT_i
0					14
1	$\emptyset$	$\emptyset$	n_1	$\emptyset$	14
2	$(n_1,n_2),(n_1,n_3)$	(n_1,n_3)	n_3	(n_1,n_3)	14
3	$(n_1,n_2),(n_3,n_4),(n_3,n_5)$	(n_1,n_2)	n_2	(n_1,n_2)	12.5
4	$(n_3,n_4),(n_3,n_5)$	(n_3,n_4)	n_4	$\emptyset$	12.5
5	(n_3,n_5)	(n_3,n_5)	n_5	$\emptyset$	12.5
6	$(n_4,n_6),(n_5,n_6)$	$(n_4,n_6),(n_5,n_6)$	n_6	(n_4,n_6)	11.5
7	$(n_2,n_7),(n_6,n_7)$	$(n_2,n_7),(n_6,n_7)$	n_7	(n_6,n_7)	10.5

Initially, the tasks are mapped in separate clusters and the priorities are computed. The following priority list along with the priority tuples that include the bottom up level, the highest bottom up level of its child, and so on is easily derived: $\{n_1 < 14, 11.5, 8, \ldots, 1 >, n_3 < 11.5, 6.5, \ldots, 1 >$, $n_2 < 8, 1 >, n_4 < 6.5, 3, 1 >$, $n_5 < 6.5, 3, 1 >, n_6 < 3, 1 >, n_7 < 1 >\}$.

First n_1 is scheduled to processor P_0. At the second step, the free task n_3 is selected and is scheduled to P_0 since its starting time reduces from 2 to 1. At the third step, n_2 is scheduled in P_0 after n_3 according to SCH_ALG, since again this processor allows its earliest execution. Now the parallel time becomes $PT_3 = 12.5$ which is the length of the $DS =< n_1, n_3, n_4, n_6, n_7 >$. Next n_4 is considered but cannot be

estimate of $O(e \log v)$. Thus the total time complexity of DSC is $O((v+e) \log v)$. The space complexity is $O(v+e)$. For linear clustering the cost reduces to $O(v \log v + e)$.

5.4 Wu and Gajski's MCP $O(v^2 \log v)$ clustering algorithm

Algorithm. Wu and Gajski [17] have proposed two scheduling algorithms for a bounded number of processors. These are the MCP (Modified Critical Path) and MD (Mobility Directed). The MCP reduces to an edge-zeroing clustering algorithm when it is used as a scheduling algorithm on a completely connected architecture with unbounded number of processors. On the other hand, the MD algorithm is not an edge-zeroing algorithm and we will not compare it with the other algorithms in this paper.

The MCP algorithm is described below:

> Determine a priority list based on the "highest *bot_level* first" ordering of SG_0. If two tasks have the same level then break the tie by using the highest level of its successor tasks, the successor of its successors and so on.
>
> **WHILE** (There exists an unscheduled task in the priority list) **DO**
>   Find the free node with the highest priority.
>   Schedule this task to a processor (cluster) that allows its *earliest execution*.
> **ENDWHILE**

There are certain similarities between the DSC and the MCP algorithm. They are both implemented as scheduling algorithms that have the earliest starting time heuristic as a scheduling guide. There is however a major difference in the choice of the priorities in the free list. The DSC uses the sum of *top_level* and *bot_level* of SG_i while the MCP uses only the *bot_level* of SG_0. As a result the MCP cannot identify the dominant sequence and may not zero its edges. The view of MCP algorithm as an edge-zeroing algorithm is given in Figure 10.

Goal Transformation. The primary goal for this algorithm is G1, the minimization of the parallel time. The cost function for G1 is the parallel time which is equal to

$$PT = \max_j (ST(n_j) + \tau_j) \leq \max_j ST(n_j) + \max_j \tau_j.$$

This implies that minimizing the starting time of the last task could result in the reduction of the overall parallel time. Therefore the transformed goal is the minimization of the starting time of the output task. The MCP heuristic is trying to achieve this by starting the execution of every task at the earliest possible

```
USN_0 = V; i = 0
WHILE USN_i ≠ ∅ DO
    n_x = head(FL);/* The free task with the highest priority. */
    n_y = head(PFL); /* The partial free task with the highest priority.*/
    IF (priority(n_x) ≥ priority(n_y)) THEN
        Minimize top_level(n_x, i) under the constraint CT1
        by zeroing some of its incoming edges. Schedule a task after the
        last scheduled task in that processor (cluster).
        If no zeroing is accepted then schedule n_x in a new processor.
    ELSE
        Minimize top_level(n_x, i) under the constraint CT1 and CT2
        by zeroing some of its incoming edges. Schedule a task after the
        last scheduled task in that processor (cluster).
        If no zeroing is accepted then schedule n_x in a new processor.
    ENDIF
    Delete n_x from USN_i and add it into SN_i. Update FL and PFL.
    Set i = i + 1.
ENDWHILE
```

Figure 9: DSC as a priority scheduling algorithm.

above implies that DS must go through an immediate predecessor of n_y that belongs in SN_i. If the rest of the nodes in PFL depend on the head node n_x of FL, then we must dynamically maintain PFL since the *top_level* of its nodes will change at each step. This could cost $O(v+e)$ since it requires the re-computation of *top_levels*.

We slightly modify the algorithm to reduce the complexity without affecting the final result. Instead of $top\_level(n_k, i)$ we use

$$startbound(n_k, i) = \max\{top\_level(n_m, i) + \tau_m + c_{m,k}\}, \quad n_m \in SN_i \bigcap PRED(n_k, i)$$

where $PRED(n_k, i)$ is the set of immediate scheduled predecessors of n_k in SG_i. We can easily show that *startbound* is a lower bound of *top_level*, see Yang and Gerasoulis [16], by proving

$$startbound(n_k, i) \leq top\_level(n_k, i), \quad n_k \in PFL, \qquad startbound(n_y, i) = top\_level(n_y, i),$$

$$n_y = head(PFL), \qquad startbound(n_k, i) = top\_level(n_k, i), \qquad n_k \in FL.$$

Maintaining FL and PFL priority lists cost $O(\log v)$ and since there are v steps the cost is $O(v \log v)$. Adding the graph traversal cost of $O(v+e)$ for a total $O(v \log v + e)$.

The incoming edge zeroing minimization procedure can be computed in $O(|PRED(n_x)| \log |PRED(n_x)|)$ which is the cost for sorting the priorities of its predecessors. Summing over all tasks we get an upper bound

CLU_HEU (DSC) : Reduce the length of the Dominant Sequence.

SCH_ALG : A task is scheduled either after the last scheduled task of a cluster in SG_{i-1} or it is the first scheduled task in a new cluster.

$domain(op_i)$ = Unexamined edges.

$focus(op_i)$ = Incoming edges of a free node that belong to the longest path going through any of the free nodes.

$Constraint = CT1$, or $CT1$ & $CT2$

$zero(op_i)$ = Incoming edges of $focus(op_i)$ if $constraint$ is true.

Termination criterion: All nodes have been examined

Figure 8: Edge zeroing operations in DSC.

The free node with the highest priority will be scheduled to the processor that allows its earliest execution. If no such processor exists then it is scheduled into a new processor. At each scheduling step, we need to maintain two node lists: a partial free list PFL which contains nodes for which at least one of its predecessors has been scheduled but not all predecessors have been scheduled, and a free list FL whose elements are free nodes. Both lists are sorted in a descending order of their task priorities. We break a tie in the priorities by using the most immediate successor first (MISF) strategy [8]. Function $head(L)$ returns the first node in the sorted list L, which is the task with the highest priority. If $L = \{\}$, $head(L) = NULL$ and $priority(NULL) = 0$. We summarize the scheduling algorithm in Figure 9.

We need to show that the above algorithm identifies the DS at each step. To do that we must show that a DS with at least one unexamined edge must pass through the head nodes of either FL or PFL. We have that

$$PT_i = \max_{q=1:v}\{\ priority(n_q, i)\ \} = \max\{\ \max_{n_q \in SN_i} priority(n_q, i),\ \max_{n_q \in USN_i} priority(n_q, i)\ \}.$$

In Yang and Gerasoulis [16] we have shown that

$$\max_{n_q \in USN_i} priority(n_q, i) = \max\{priority(n_x, i),\ priority(n_y, i)\}$$

where $n_x = head(FL)$ and $n_y = head(PFL)$ which proves our result.

Complexity. We make a key observation regarding the complexity of the DSC algorithm. If the DS goes through the head of FL then updating both FL and PFL costs $O(\log v)$ per step if a balanced search tree data structure is used. On the other hand, if DS goes through the head n_y of PFL then the result

DS. If reducibility is detected at the present step then it will be reducible at some future step because of constraint $CT2$. If we find that DS is not reducible then $CT2$ is ignored, [15, 16].

The minimization procedure to achieve the shortest *top_level* **for** $CT1$**.** The DSC algorithm minimizes the $top\_level(n_x, i)$ at each step. The minimum is derived by using the optimum algorithm for the join set, which is similar to the fork set optimum clustering algorithm described in section 3.1. The join set used for the minimization includes those scheduled predecessors of n_x which have only one successor (i.e. n_x). The priorities used for sorting the edges are the lengths of the $top\_level(n_m, i) + \tau_m + c_{m,x}$, where n_m is a predecessor of n_x.

Example. Applying DSC algorithm to the example of Figure 1(a), we obtain two clusters with $PT = 9$ as shown in Figure 11(c). The clustering steps are shown in Table III.

Table III: Clustering steps of DSC for Figure 1(a).

step i	$domain(op_i)$	$focus(op_i)$	free node	$zero(op_i)$	PT_i
0					14
1	$\emptyset$	$\emptyset$	n_1	$\emptyset$	14
2	$(n_1, n_2), (n_1, n_3)$	(n_1, n_2)	n_2	(n_1, n_2)	13.5
3	(n_1, n_3)	(n_1, n_3)	n_3	$\emptyset$	13.5
4	$(n_3, n_4), (n_3, n_5)$	(n_3, n_4)	n_4	(n_3, n_4)	12.5
5	(n_3, n_5)	(n_3, n_5)	n_5	(n_3, n_5)	11.5
6	$(n_4, n_6), (n_5, n_6)$	$(n_4, n_6), (n_5, n_6)$	n_6	$(n_4, n_6), (n_5, n_6)$	10
7	$(n_5, n_7), (n_6, n_7)$	$(n_5, n_7), (n_6, n_7)$	n_7	(n_6, n_7)	9

At beginning $PT_0 = 14$. The first unexamined edge from top to bottom in DS is (n_1, n_2) and n_2 is free. Zeroing this edge will minimize its starting time and this zeroing is accepted and n_2 is scheduled after n_1. Next (n_1, n_3) is chosen for zeroing. Zeroing this edge will increase its starting time since it must be scheduled after n_2. Therefore, this zeroing is not accepted and n_3 is marked examined. Continuing with the algorithm we get the clusters $M_0 = \{n_1, n_2\}$, $M_1 = \{n_3, n_4, n_5, n_6, n_7\}$ and the parallel time is reduced to 9.

DSC as a priority scheduling algorithm. Under the above assumptions, the DSC algorithm can be implemented as a priority scheduling mechanism on unbounded number of processors having the node priorities as follows:

$$priority(n_k, i) = top\_level(n_k, i) + bot\_level(n_k, i).$$

us to consider the following strategy:

1. Suspend zeroing an unexamined edge (n_m, n_y) in DS until the head node n_y becomes free.

2. Choose a free node n_x which belongs to the *longest path going through any of the free nodes* in SG_{i-1}. Zero its incoming edge(s) provided that constraints $CT1$ and $CT2$ are satisfied.

3. $CT2$: Zeroing incoming edges of n_x to minimize $top\_level(n_x, i-1)$ should not affect the strict reduction of $top\_level(n_y, i-1)$ at some future step j, $i \leq j$.

4. If all edges in a DS have been examined and this DS continues to dominate in the next step, then recursively apply the above three steps on the next longest path (SubDS) to reduce the number of unnecessary processors.

Some explanations are in order. Constraint $CT2$ is closely related to *ptrw* property $T2$. If at the step op_i the $top\_level(n_y, i-1)$ can be strictly reduced, we should be able to get this reduction, or even more of a reduction, at some future step op_j. If we do not reduce this DS, then we will not be able to reduce the parallel time since this DS will continue to dominate. Therefore, we want to make sure that edge zeroings that are not in the current DS do not affect the reducibility of the current DS at some future step. Of course, $CT2$ is not equivalent to $T2$ constraint, since the strict reduction of $top\_level(n_y, i-1)$ does not imply the strict reduction of PT_{i-1}. In other words, the inequality $top\_level(n_y, i) < top\_level(n_y, i-1)$ implies $PT_i \leq PT_{i-1}$ rather than $PT_i < PT_{i-1}$ because $PT_i = \max_{k=1:v}\{\, top\_level(n_k, i) + bot\_level(n_k, i)\,\}$ and n_y may or may not belong in DS_i.

Scheduling Algorithm. The breadth first strategy above warranties that the *bot_levels* in USN_i do not have to be recomputed. We would also like to do the same for the *top_levels* of nodes in SN_i. To do that we must choose a scheduling heuristic that avoids insertion of tasks between already scheduled tasks in SG_{i-1}. Thus, SCH_ALG: A task is scheduled either after the last scheduled task of a cluster in SG_{i-1} or it is the first scheduled task in a new cluster.

Detecting the reducibility of DS for $CT2$. Now the question that arises is how the reducibility of DS can be detected. This is done by examining the result of the zeroing of the first unexamined edge in

Let us assume for a moment that when two clusters are merged, the tasks are ordered according to the highest $bot\_level(n_j, i-1)$ heuristic. Furthermore, assume that a zeroing is accepted only if property $T1$ is satisfied. In the example below we show how this algorithm works. Afterwards, we modify these assumptions to further reduce the computational complexity.

Example. Consider the example in Figure 1(a). At the beginning, $DS_0 =< n_1, n_2, n_7 >$ and $PT_0 = 14$. In the first step, we choose (n_1, n_2) and by zeroing this edge the new $DS_1 =< n_1, n_3, n_4, n_6, n_7 >$ with $PT_1 = 13.5$. Thus this zeroing is accepted. In the second step we choose (n_1, n_3) and by zeroing this edge the parallel time reduces to $PT_2 = 12.5$ by inserting n_3 before n_2 according to the highest bottom up level scheduling algorithm, since $bot\_level(n_3, 1) = 11.5 \ > \ bot\_level(n_2, 1) = 8$. At the third step we focus on (n_3, n_4) and by zeroing this edge the parallel time increases to 13.5 and as a result this zeroing is rejected. At the fourth step, we focus on (n_4, n_6) and by zeroing this edge the parallel time reduces to 11.5. Next the edge (n_3, n_5) is considered and its zeroing is rejected. Continuing this way we derive the clusters $M_0 = \{n_1, n_2, n_3\}$, $M_1 = \{n_4, n_6, n_7\}$, $M_2 = \{n_5\}$ with $PT = 10.5$.

The Dominant Sequence Algorithm (DSC). In the assumptions above we used $T1$ to decide if a zeroing should be accepted or not. Another more restrictive constraint that could be used instead, is to accept an edge zeroing if the starting time of its head node decreases. The constraint below automatically satisfies $T1$, since reducing *top_level* for each node results in the reduction of the parallel time:

($CT1$): *An edge zeroing is accepted if it reduces the top_level of its head node.*

Even though we have imposed a systematic edge zeroing by choosing the first unexamined edge from top to bottom in DS the complexity of the algorithm is still high. There are two problems. The first one is that the edge zeroing traversal could proceed in a *depth first* manner. Therefore, the *bot_levels* of the unscheduled predecessors in USN_i could change by an edge zeroing. The second one is that when two clusters are merged, the scheduling algorithm allows for node insertion between already scheduled nodes in a cluster, which implies that the *top_levels* of scheduled nodes in SN_i will also be affected. As a result determining the next DS could cost $O(v+e)$ per step since all levels must be recomputed. One way to avoid recomputing the *bot_levels* is to traverse the task graph in *breadth first* manner. This implies that we must compromise and zero edges that do not belong in DS, before we zero an edge in DS. This leads

mising on complexity. As we saw in the previous subsection, zeroing the edges in the dominant sequence will reduce the length of this DS, and if this DS is unique it will reduce the parallel time. The main idea behind a dominant sequence heuristic is to identify the DS at each step and then zero edges in that DS. In designing an algorithm that uses the DS as a zeroing guide the following questions must be addressed:

What is the cost of identifying DS_i at each clustering step. Given a node $n_x \in SG_i$, then $n_x \in DS_i$ if and only if the following condition is true: $top\_level(n_x, i) + bot\_level(n_x, i) = PT_i$. If SG_i is given, then identifying DS_i from scratch requires the computation of top_level and bot_level which costs $O(v + e)$. This time complexity is not practical for large task graphs with thousands of tasks. Thus, for lower time complexity clustering algorithms we must come up with an *incremental* way of identifying DS_i to avoid the re-computation of all levels at each step.

Once the DS_i is identified, which edges should be chosen for zeroing. A greedy heuristic would choose to zero those edges in DS_i that result in the largest possible decrease in the parallel time. Such edges belong in the $ptrw(op_i)$ set which we have defined in the characterization of clustering algorithms section. Therefore a greedy heuristic will have to compute the parallel time for each edge zeroing in DS_i which again results in high complexity. Furthermore, since a single edge zeroing could change DS in the next step, it is not necessary to zero more than one edge in DS per step.

Considering the above discussion and since we are interested in “almost linear” time complexity algorithms with good performance characteristics, we must zero edges systematically. Before we describe our systematic edge zeroing, we need to make few definitions. At the beginning of the algorithm, all edges are marked *unexamined*. After an edge has been considered for zeroing at op_i, it is marked *examined* and its head node is *scheduled*. A node is *free* if all of its predecessors have been scheduled.

From all DS edges, we choose for zeroing *the unexamined edge first from top to bottom in DS*. In case of two DS, we break the tie by choosing the first unexamined edge whose head node has the most immediate successors and so on. At the completion of clustering step op_i two sets of nodes are created, the *scheduled* set of nodes SN_i and the *unscheduled* set of nodes USN_i. At each step nodes from USN_i are deleted and added to SN_i. Initially, $SN_0 = \{\ The\ set\ of\ input\ nodes\}$, $USN_0 = V - SN_0$.

Table II: Clustering steps of Sarkar's algorithm for Figure 1(a).

step i	$domain(op_i)$	$focus(op_i)$	$zero(op_i)$	PT_i
0				14
1	E	(n_1, n_2)	(n_1, n_2)	13.5
2	$E - (n_1, n_2)$	(n_3, n_4)	(n_3, n_4)	12.5
3	$E - (n_1, n_2) - (n_3, n_4)$	(n_3, n_5)	(n_3, n_5)	11.5
4	$(n_1, n_3), (*, n_6), (*, n_7)$	(n_2, n_7)	(n_2, n_7)	11.5
5	$(n_1, n_3), (*, n_6), (n_6, n_7)$	(n_4, n_6)	(n_4, n_6)	11.5
6	$(n_1, n_3), (n_5, n_6), (n_6, n_7)$	(n_5, n_6)	(n_5, n_6)	10
7	$(n_1, n_3), (n_6, n_7)$	(n_1, n_3)	$\emptyset$	10
8	(n_6, n_7)	(n_6, n_7)	$\emptyset$	10

and the SCH_ALG must be used for the computation of the parallel time.(Sarkar uses a slightly different SCH_ALG, based on the latest starting task time to order tasks. The performance of both scheduling heuristics is similar). The bottom up levels of both n_2 and n_3 are 6.5 and the parallel time is $PT_3 = 11.5$ for either ordering of these nodes. Then at step 4, (n_2, n_7) is zeroed, and the parallel time remains the same $PT_4 = 11.5$. In the following steps, $(n_4, n_6), (n_5, n_6)$ will be zeroed and PT_4 reduces to 10. The edges (n_1, n_3) and (n_6, n_7) cannot be zeroed otherwise all nodes would be in the same cluster and the parallel time would increase to 13. Finally, Sarkar's algorithm obtains two clusters with $PT = 10$, as shown in Figure 11(b) $M_0 = \{n_1, n_2, n_7\}$, $M_1 = \{n_3, n_4, n_5, n_6\}$.

As it can be seen from the goal transformation inequality, minimizing CV might not reduce the parallel time at all, unless the corresponding zeroed edges belong to DS. For example in the fourth step above, the dominant sequence is $< n_1, n_3, n_4, n_5, n_6, n_7 >$. However Sarkar's algorithm is unable to identify this sequence and it zeros the edge $< n_2, n_7 >$ which is not in this DS and the parallel time remains unchanged. Had the edge (n_6, n_7) been zeroed instead, the parallel time would have been reduced to 9.

Complexity. Sarkar computes the levels at each clustering step, and then uses the level information to schedule the tasks and to determine the parallel time. The computation of the levels costs $O(v + e)$ at each step and since there are e such steps the total cost of the algorithm is at least $O(e(v + e))$.

5.3 An $O((e + v) \log v)$ Dominant Sequence Clustering algorithm

A general dominant sequence algorithm. In Yang and Gerasoulis [15, 16] a new clustering algorithm has been proposed. This algorithm combines the best features of several other algorithms without compro-

$O(v+e)$. Therefore the complexity of KB/L algorithm is $O(v(v+e))$. For a dense graph $e = v^2$ and the complexity becomes the upper bound $O(v^3)$ given by Kim.

5.2 Sarkar's $O(e(v+e))$ algorithm

Algorithm. Sarkar's algorithm [14], p. 129, can be summarized as follows: (1) Sort the edges of the DAG in descending order of edge costs. (2) Zero the highest edge if the parallel time does not increase. (3) Repeat step 2 until all edges are scanned. The clustering step of this algorithm is characterized in Figure 7.

Goal Transformation. Let us look at the inequality

$$PT = \sum_{n_j \in DS} \tau_j + \sum_{n_x, n_j \in DS} c_{xj} \leq \sum_{n_j \in DS} \tau_j + CV$$

where CV is the communication volume. Sarkar's primary goal is the minimization of the parallel time G1. The transformed goal and corresponding heuristic is to minimize CV without increasing PT.

> *CLU_HEU* (Sarkar) : Zero the highest communication edge if the parallel time does not increase.
>
> *SCH_HEU* : When two clusters are merged the tasks are ordered according to the highest $bot\_level(n_x, i-1)$ first heuristic.
>
> $domain(op_0) = E, \quad focus(op_0) = \emptyset.$
>
> $domain(op_i) = domain(op_{i-1}) - focus(op_{i-1}).$
>
> $focus(op_i)$ = Edge with maximum cost in $domain(op_i)$.
>
> $constraint$: $PT_i \leq PT_{i-1}$.
>
> $zero(op_i) = focus(op_i)$ if $constraint$ is satisfied; otherwise is $\emptyset$.
>
> Termination criterion: $domain(op_i) = \emptyset$.

Figure 7: Edge zeroing operations in Sarkar's algorithm.

.

Example. For Figure 1(a), Sarkar's algorithm sorts all edges first. The sorted list is:

$\{(n_1, n_2), (n_3, n_4), (n_3, n_5), (n_2, n_7), (n_4, n_6), (n_5, n_6), (n_1, n_3), (n_6, n_7)\}$.

The clustering steps are shown in Table II.

At the beginning, all tasks are assumed to be in separate clusters and $PT_0 = 14$. In the first two steps, (n_1, n_2) and (n_3, n_4) are zeroed and the parallel time is reduced to 12.5. At the third step, (n_3, n_5) is zeroed

CLU_HEU (KB/L) : Reduce the length of the longest path determined by a cost function.

SCH_ALG : No scheduling algorithm is needed since clustering is linear which implies that $SG_i = G_i$.

$domain(op_0) = E$.

$domain(op_i) = E_i = E_{i-1} - \{e_{j,k}\}, \quad n_j$ or $n_k \in CP_{i-1}$.

$focus(op_i)$ = Edges in the longest path CP_i among all paths in $domain(op_i)$.

$constraint$: Linearity of clustering.

$zero(op_i) = focus(op_i)$.

Termination criterion: $domain(op_i) = \emptyset$.

Figure 6: Edge zeroing operations in KB/L.

factors. Here we assume that $w_1 = 1/2$ and $w_2 = 1$ in which case the cost function reduces to the length of the critical path. Under this assumption Kim's algorithm can be considered as having the goal of finding the linear clustering with the minimum parallel time (G4/G1). Each op_i for KB/L algorithm is defined in Figure 6.

Example. Consider the example of Figure 1(a). The result of applying Kim's linear clustering algorithm is shown in Figure 1(b) with $PT = 11.5$. The clustering steps are shown in Table I. The symbols $(*, n_x)$ and $(n_x, *)$ represent the sets of all incoming and outgoing edges of node n_x respectively.

Table I: Clustering steps of KB/L for Figure 1(a).

step i	$domain(op_i)$	$focus(op_i)$	$zero(op_i)$	PT_i
0				14
1	E	$(n_1, n_2), (n_2, n_7)$	$(n_1, n_2), (n_2, n_7)$	13.5
2	$(n_3, *), (*, n_6)$	$(n_3, n_4), (n_4, n_6)$	$(n_3, n_4), (n_4, n_6)$	11.5
3	$\emptyset$			11.5

At the beginning, the critical path of the original DAG is $< n_1, n_2, n_7 >$ and these nodes are clustered together. By deleting those nodes from the graph, the remaining graph has 4 nodes (n_3, n_4, n_5, n_6). Its critical path is $< n_3, n_4, n_6 >$ and these nodes are clustered together. Finally, the resulting clustering is: $M_0 = \{n_1, n_2, n_7\}$, $M_1 = \{n_3, n_4, n_6\}$, $M_2 = \{n_5\}$.

Complexity. Kim, in page 40 of his thesis, has given the complexity of his *Linear-Cluster* algorithm as $O(v^3)$. The number of connected components is at most v and for each step finding the longest path costs

Multiplicity of edge zeroing. The number of edges that are zeroed at each step is another characteristic of a clustering algorithm. A clustering algorithm should strike a balance between performance and complexity goals.

Parallel Time Approximation. Given q clusters, the parallel time can be estimated by executing these clusters on q virtual processors. Since finding an optimal schedule is NP-complete, approximate scheduling algorithms must be used instead. It is important to choose a good scheduling algorithm so that clustering decisions can be made as accurate as possible.

Complexity. In some practical applications, the number of task nodes could run into thousands. Therefore, an algorithm with high time complexity would be computationally impractical for such task graphs.

5 A description of several clustering algorithms

In this section, we present four clustering algorithms from the class of edge zeroing algorithms. We first discuss these algorithms, and then use our framework to specify their edge-zeroing sequences. In the following two sections we analyze their performance both theoretically and experimentally.

5.1 Kim and Browne's $O(v(e+v))$ linear clustering algorithm

Algorithm. Kim [9] and Kim and Browne [10] proposed the following linear clustering algorithm, henceforth called the KB/L algorithm. Initially all edges are marked *unexamined*: (1) Determine the longest path CP_i composed of only *unexamined* edges, by using a weighted cost function $Cost\_function$. The nodes in this path constitute a cluster and their edge costs are zeroed. (2) Mark all edges incident to the nodes in CP_i *examined.* (3) Recursively apply steps 1 and 2 until all edges are examined.

Goal Transformation. In his thesis, Kim [9] uses the following cost function:

$$Cost\_function = w_1 * \sum \tau_i + (1 - w_1)(w_2 * \sum c_{i,j} + (1 - w_2) * \sum c_{i,j}^{adj})$$

for determining the length of CP_i, where w_1 and w_2 are the normalization factors and the sums are over all nodes in the path and $c_{i,j}^{adj}$ is the edge communication cost between a node in the path and all its adjacent nodes outside the path. Kim does not give a systematic way for the determinations of normalization

use the stronger condition $PT_i < PT_{i-1}$ in an algorithm to get the greatest reduction in the parallel time immediately, but then the algorithm might need to perform multiple edge-zeroing to avoid early termination. This complicates the design of non-backtracking algorithms. For the fork set example of the previous section, if $\tau_1 + \beta_1 = \tau_2 + \beta_2$ then there are two DSs, and if the while condition is changed to $PT_i < PT_{i-1}$ then this algorithm will stop without zeroing any edges.

We define the *parallel time reduction warranty* subset $ptrw(op_i)$ of DS_i at the completion of op_i, as the set of edges for which the parallel time will *strictly* decrease by zeroing any of its edges. So a *greedy heuristic* should zero edges in the $ptrw(op_i)$ set as soon as possible. Thus we define the following property:

$(T2)$ *For every step i for which the set $ptrw(op_i)$ is non empty, the clustering algorithm zeros at least one edge in $ptrw(op_i)$ at some future step j, where $i < j \leq k$ and k is the last step of the algorithm. Moreover, the zeroed edge also belongs to $ptrw(op_{j-1})$ and $ptrw(op_k) = \emptyset$.*

Determining each edge in $ptrw(op_i)$ from scratch requires the evaluation of the parallel time which cannot be used for algorithms with low time complexity. Even though we cannot use $ptrw$ to guide edge zeroing for low complexity algorithms, it is of interest to know for what classes of task graphs a given algorithm zeroes edges in $ptrw(op_i)$ at each step. If $j = i + 1$ then the algorithm strictly reduces the parallel time immediately in the next step. However, an algorithm could delay this zeroing to a future step to allow for more flexibility. It is not difficult to show that: *Every optimum linear clustering satisfies $T1$ and $T2$.* Let us look at the sorted fork set optimum clustering algorithm once more, where k is the last step of the algorithm. We have

$$ptrw(op_i) = \begin{cases} \{(n_x, n_{i+1})\} & i \leq k-1 \text{ and } \tau_{i+1} + \beta_{i+1} > \tau_{i+2} + \beta_{i+2} \\ \emptyset & \text{otherwise} \end{cases}$$

and this optimum sequence of zeroing satisfies $T1$ and $T2$, where $j = i + 1$.

Constraints. Constraints on the heuristics might be imposed to achieve their goals. For example the non-increase of the parallel time, $PT_i \leq PT_{i-1}$, constraint is used in the optimum fork clustering algorithm in the previous section. In addition to "goal achieving" constraints, other constraints might be imposed to reduce the computational complexity of the heuristic. For example non-backtracking is a constraint that considerably reduces the computational complexity.

is equivalent to solving:

$$PT_{opt} = \min_{i=1:m} \max(\sum_{j=1}^{i} \tau_j,\ \tau_{i+1} + \beta_{i+1}) + \tau_x.$$

Thus the new transformed goal becomes the minimization of the last function which can be achieved by zeroing β_{i+1} whenever $\tau_{i+1} + \beta_{i+1}$ is the maximum above. Repeating such zeroing yields an optimal zeroing sequence which satisfies $PT_0 \geq PT_1 \geq \cdots \geq PT_k$ and $PT_k < PT_{k+1}$. It just happened for this example that the achievement of the transformed goal also implies the achievement of the original goal. This is not true in general.

Another example of goal transformation is Sarkar's algorithm [14] which has as a primary goal G1. His transformed goal is "the minimization of communication volume G3 without increasing the parallel time".

Performance features. What are the special features that warranty good performance of clustering algorithms. Because the clustering problem is NP-complete for the performance goals G1/G2, it is extremely difficult to find the features that will warranty optimum clusterings. There are certain features however, which we believe are necessary to assure good performance of clustering algorithms.

Monotonic decrease of parallel time. Let us consider the non-backtracking clustering algorithms. To ensure that such algorithms produce a clustering whose parallel time is not worse than the initial parallel time, a safe-guard must be imposed. One such safe-guard is the *non-increase* of the parallel time at each step of the algorithm:

$(T1)\quad PT_i \leq PT_{i-1}.$

This condition assures that at least a *local minimum* of the parallel time of a clustering sequence will be derived. As a matter of fact if the task graph is coarse grain any local minimum of a clustering algorithm that satisfies $T1$ will be within a factor of two of the optimum clustering. This is because for coarse grain graphs every linear clustering is within a factor of two of the optimum, see [5, 6]. Since the initial clustering is linear then $T1$ assures that the parallel time does not increase. Thus, $T1$ is a reasonable constraint at the least for coarse grain DAGs.

Parallel time reduction warranty. Assumption $T1$ is not sufficient to enforce strict reduction in the parallel time at each step and a heuristic that satisfies $T1$ may not reduce the parallel time at all. We can of course

CLU_HEU : Minimize the parallel time.

SCH_ALG: Any ordering of $n_1, n_2, \ldots, n_i$ results in an optimal schedule.

$domain(op_i) = \{< n_x, n_i >, \cdots, < n_x, n_m >\}$

$focus(op_i) = \{< n_x, n_i >\}$

$constraint$: $PT_i \leq PT_{i-1}$

$zero(op_i) = focus(op_i)$ if $constraint$ is true otherwise $\emptyset$.

$PT_i = \max(\sum_{j=1}^{i} \tau_j, \tau_{i+1} + \beta_{i+1}) + \tau_x$

Termination criterion: When *constraint* is not true.

Figure 5: Edge zeroing operations in the fork clustering algorithm.

the *performance* type and the *non-performance* type. *Performance* goals could be the following: (G1) Minimization of the parallel time cost function on an unbounded number of processors. (G2) Maximization of the efficiency cost function. (G3) Minimization of the communication volume, $CV = \sum_{e_{i,j} \in E} c_{i,j}$, cost function. *Non-performance* goals on the other hand impose constraints on the structure and shape of the clustering rather than its performance. Examples are: (G4) Clustering is linear. (G5) Clusters have no cycles, i.e. they are convex, Sarkar [14]. (G6) Clustering satisfies the locality of data assumption [4].

One can use a combination of goals as long as they do not conflict with each other. Whenever conflicts occur, then a goal priority must be imposed. For example G4/G1 implies that G4 will be used first when there is a conflict and the result will be an optimum linear clustering. On the other hand, if we use G1/G4, the primary goal G1 could lead to nonlinear clusters if the corresponding parallel time is shorter. G2 and G3 alone are not *reasonable* goals, unless combined with another goal or constraint, since maximization of the efficiency and minimization of communication volume can both be achieved by mapping all nodes to a single cluster.

Goal Transformation. Because of the NP-completeness of the problems that have some of the above goals, these problems cannot be solved in polynomial time. Therefore, the goals must be transformed so that their cost functions are directly computable in polynomial time. For example the goal for our optimum fork clustering algorithm in the last section is G1. Because of the special structure of the graph, the goal

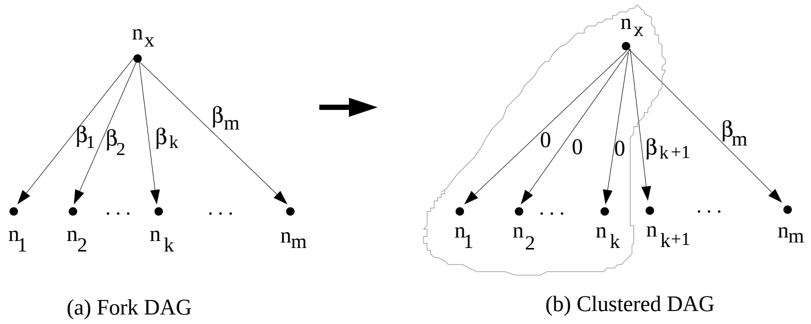

Figure 3: Clustering a fork DAG.

Let us now consider a fork DAG F_x with communication costs $c_{x,j} = \beta_j$, $j = 1 : m$ shown in Figure 3(a) to demonstrate an edge-zeroing clustering sequence. We present an *optimum clustering algorithm* for a fork DAG in Figure 4. For simplicity we assume that the nodes and edges have been sorted such that $\tau_j + \beta_j \geq \tau_{j+1} + \beta_{j+1}$, $j = 1 : m - 1$.

$i = 1;$
$PT_0 = \tau_1 + \beta_1 + \tau_x$
$PT_1 = \max(\tau_1, \tau_2 + \beta_2) + \tau_x$
WHILE $(PT_i \leq PT_{i-1})$ **DO**
  $\beta_i = 0;$
  $i = i + 1;$
  $PT_i = \max(\sum_{j=1}^{i} \tau_j, \tau_{i+1} + \beta_{i+1}) + \tau_x$
ENDWHILE
$k = i - 1;$

Figure 4: An optimum clustering algorithm for a fork DAG.

Initially each task is mapped on a separate processor of the clique. At each step i of the algorithm, corresponding to op_i, the focus is on the edge $< n_x, n_i >$ and if the parallel time reduces by zeroing that edge then this edge is zeroed, see Figure 3(b). Therefore, the task n_j, $j = 1 : i$ will be mapped in the same cluster if $PT_i \leq PT_{i-1}$. A summary of the algorithm is given in Figure 5. The proof of optimality is given in [15], Theorem 4.2.

4 A characterization of clustering algorithms

Clustering goals and cost functions. Clustering heuristics must have certain goals and must choose the corresponding cost functions for achieving those goals. We distinguish between two types of goals,

in a linear cluster, then such a merging will not increase the parallel time. If such a zeroing results in a nonlinear cluster then the reduction or not of the parallel time will depend on the the granularity of the DAG.

3.1 Clustering algorithms based on edge-zeroing

The edge-zeroing based merging algorithms constitute an important sub-class of clustering algorithms which we will study in detail. Such algorithms only operate on the connected component of the DAG and never merge tasks that are not connected. Edge-zeroing clustering will produce a sequence of graph transformations $G_i = (V, E, C_i, \mathcal{T}), \quad i = 0 : k$ of the initial DAG. The operation op_i only modifies the set $\mathcal{C}_{i-1}$ to $\mathcal{C}_i$ by edge zeroing while the sets V, E and $\mathcal{T}$ remain unchanged. For edge-zeroing clustering algorithms, we define:

- CLU_HEU : The CLUstering HEUristic which selects the edges to be zeroed.
- SCH_ALG: The SCHeduling ALGorithm.
- $domain(op_i)$: The set of edges in E to be examined by op_i.
- $focus(op_i)$: The set of edges that are candidates for zeroing at op_i.
- $zero(op_i)$: The set of edges that will be zeroed at the completion of op_i.
- SG_i : The scheduled DAG according to SCH_ALG. Initially, $SG_0 = G_0$.
- DS_i: The Dominant Sequence at the completion of op_i, which is the critical path of SG_i.
- CP_i: The set of all nodes in the Critical Path of $G_i = (V, E, C_i, \mathcal{T})$, $i = 0 : k$.
- $top\_level(n_x, i)$, $bot\_level(n_x, i)$: The length of the longest path between node n_x and the top (bottom) node in SG_i, including all the communication and computation costs in that path, but excluding τ_x from $top\_level(n_x, i)$.
- PT_i: The Parallel Time at the completion of op_i:

$$PT_i = \sum_{n_j \in DS_i} \tau_j + \sum_{n_j, n_m \in DS_i} c_{j,m} = top\_level(n_x, i) + bot\_level(n_x, i), \quad n_x \in DS_i.$$

3 A generic description of clustering algorithms

Clustering heuristics have certain goals and try to achieve them via a sequence of steps. Clustering algorithms perform a sequence of refinement steps op_i, $i = 0 : k$. An initial clustering is given at op_0. *Here we assume that initially each task is a cluster.* Each op_i performs a refinement of the previous clustering by *merging* some clusters and at the last step op_k, a final clustering is derived. At each step a good refinement must be performed so that the final clustering satisfies or it is "close" to satisfying the original goals. We only consider *non-backtracking* heuristic algorithms to avoid high complexity, i.e. once the clusters have been *merged* in op_i they cannot be *unmerged* afterwards. Then the number of clustering steps remains polynomially bounded with respect to the size of the DAG.

Let us demonstrate how the nonlinear clustering in Figure 1(c) could be derived as a sequence of merging operations op_i. We use the criterion that "two clusters are merged if the parallel time does not increase". Initially each task is mapped to a separate cluster shown in Figure 1(a) and the parallel time is equal to 14 which is the length of $DS_0 =< n_1, n_2, n_7 >$. The first step merges n_1 and n_2 and renames the resulting cluster as 1. Then $DS_1 =< n_1, n_3, n_4, n_6, n_7 >$ and the parallel time reduces to 13.5. In the next step clusters n_3 and n_4 are merged and renamed cluster 2. Then $DS_2 =< n_1, n_3, n_5, n_6, n_7 >$ and the parallel time reduces to 12.5. In the third step clusters n_5 and n_6 are merged to become cluster 3. Then $DS_3 =< n_1, n_3, n_5, n_6, n_7 >$ and the parallel time reduces to 10. In the final step clusters 2 and 3 are merged. Since n_4 and n_5 are two independent task assigned in the same processor an ordering must be used to determine the parallel time. In this case the parallel time remains equal to 10 if n_4 is executed before n_3 or vice versa.

Several interesting observation can be made: (1) When two clusters are merged, a scheduling heuristic might be needed to determine the new parallel time and measure the performance. (2) If there exists a nonzero edge connecting two clusters, then merging the two clusters will zero the edge cost. Equivalently, zeroing an edge cost will merge two clusters. (3) If there is no edge connecting two clusters then the parallel time *cannot* be reduced by merging these clusters, but it might be increased by such a merging because of sequentialization of independent tasks. (4) If zeroing the edges connecting two linear clusters results

and n_y of a cluster, if n_y is executed immediately after n_x and there is no data dependence edge between n_x and n_y, then we obtain what we call a *scheduled DAG*, see 2 (c). We call the longest path of the scheduled DAG the *dominant sequence* (DS) of the clustered DAG, to distinguish it from the *critical path* (CP) of a clustered but not scheduled DAG. For example, the clustered DAG in Figure 1(a) also shown in Figure 2 has $CP =< n_1, n_2, n_7 >$ with length 9, while a DS of a schedule given in Figure 2(c) is $DS =< n_1, n_3, n_4, n_5, n_6, n_7 >$ and has length 10. In the case of linear clustering, the DS and CP of the clustered DAG are identical, see Figure 1(b).

Clustering has been used as a first step to scheduling parallel architectures. More specifically, Sarkar [14] who calls clustering "internalization pre-pass" proposes a two step method to scheduling: (1) Perform clustering by scheduling on an unbounded number of processors of a clique. (2) Merge and schedule the clusters when the number of processors is smaller than the number of clusters. Sarkar gives the following justification for clustering: "If tasks are scheduled in the same processor on the best possible architecture with unbounded number of processors, then they should be scheduled in the same processor in any other architecture." Others areas in the literature where clustering has been used are the VLSI systolic schedules, Kung [11], where the clustering step is known as the processor projection step and numerical computing for message passing architectures, Ortega [12].

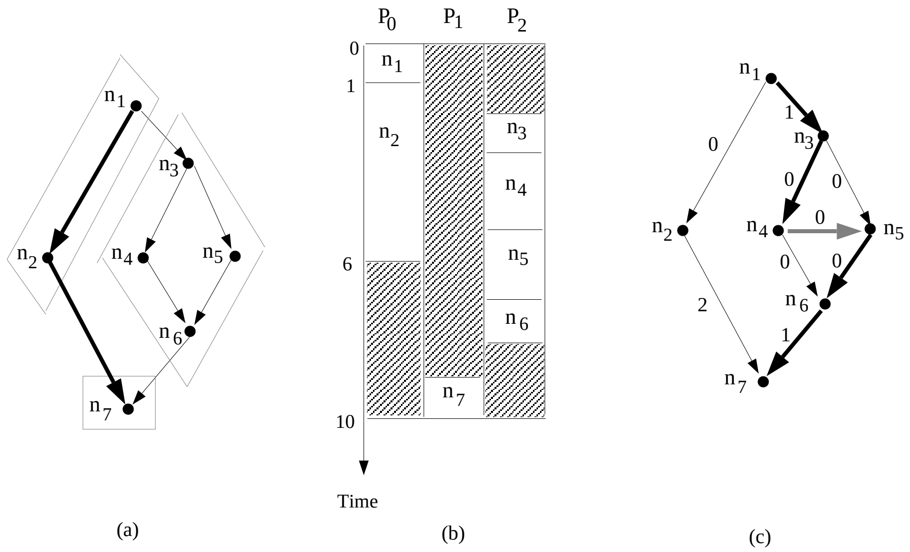

Figure 2: (a) The clustered DAG and its CP shown in thick arrows. (b)The Gantt chart of a schedule. (c) The scheduled DAG and the DS shown in thick arrows.

after completion of the task execution, the data is sent to its successor tasks. Data communication is done in *parallel.*

The architecture is a completely connected graph, with an *unbounded number* of homogeneous processors, i.e. a *clique* virtual architecture.

Clustering is a mapping of the *tasks* of a DDG onto *clusters.* A *cluster* is a set of tasks which will execute on the same processor. Clustering is also known as *processor assignment* in the case of an unbounded number of processors on a clique architecture, Sarkar [14]. Clustering has been shown to be NP-complete for the minimization of the parallel time cost function, Chretienne [1], Papadimitriou and Yannakakis [13], Sarkar [14]. As a result many heuristic algorithms have been proposed and analyzed in the literature. A clustering is called *nonlinear* if two independent tasks are mapped in the same cluster, otherwise is called *linear.* In Figure 1(a) we give a weighted DAG, in (b) a linear clustering with three clusters $\{n_1,\ n_2,\ n_7\}$, $\{n_3,\ n_4,\ n_6\}$, $\{n_5\}$ and in (c) a nonlinear clustering with clusters $\{n_1,\ n_2\}$, $\{n_3,\ n_4,\ n_5,\ n_6\}$ and $\{n_7\}$. Notice that for the nonlinear cluster the independent task n_4 and n_5 are mapped in the same cluster.

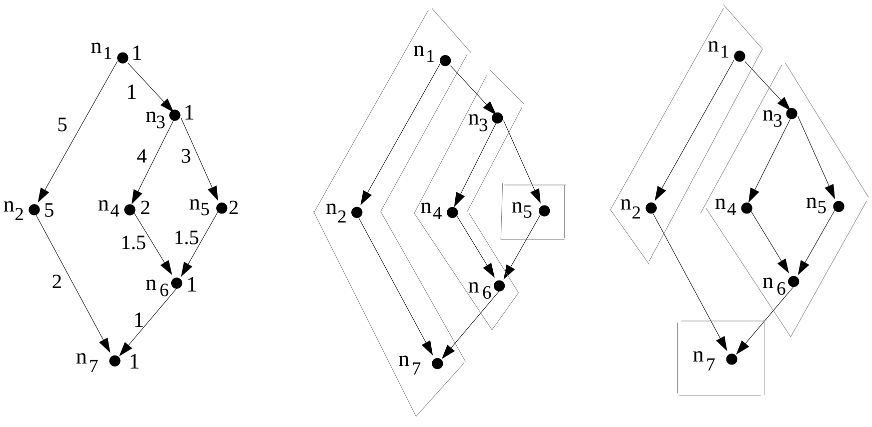

Figure 1: (a) A weighted DAG. (b) A linear clustering. (c) A nonlinear clustering.

Scheduling is a *task to processor assignment* and a *task to starting time mapping.* In general, the problem of finding the optimum scheduling that minimizes the parallel time has been shown to be NP-complete, Sarkar [14]. In Figure 2(b) we present the Gantt chart of a schedule for the nonlinear clustering of Figure 1(c). Processor P_0 has tasks n_1 and n_2 with starting times $ST(n_1) = 0$ and $ST(n_2) = 1$. If we modify the clustered DAG as in [14] by adding zero-weighted edges between any pair of two nodes n_x

scheduling algorithms are presented. Here, we introduce a general analytic framework and use it to express clustering heuristics so that comparisons can be made in a systematic fashion. The paper is organized as follows:

In section 2, we describe the basic terminology and assumptions used in clustering and scheduling algorithms. In section 3, we introduce a generic framework that visualizes a clustering algorithm as performing a sequence of clustering refinements so that a clustering algorithm can be presented in a systematic manner. In section 4, we describe the important characteristics and performance features of clustering algorithms so that we can clarify the differences and similarities and evaluate their performance through this framework. In section 5, we present four algorithms using our framework, and use an example to demonstrate their clustering steps. In section 6, we study their performance for special important primitive classes of DAGs such as fork, join and coarse grain trees. These are DAGs whose optimal solutions can be computed in polynomial time. In section 7, we present experimental results that verify our analytic results. Section 8 is the conclusion.

2 Problem definition and assumptions

We start with definitions of the task computation model and architecture:

A directed acyclic weighted task graph (DAG) is defined by a tuple $G = (V, E, \mathcal{C}, \mathcal{T})$ where $V = \{n_j, j = 1 : v\}$ is the set of task nodes and $v = |V|$ is the number of nodes, E is the set of communication edges and $e = |E|$ is the number of edges, $\mathcal{C}$ is the set of edge communication costs and $\mathcal{T}$ is the set of node computation costs. The value $c_{i,j} \in \mathcal{C}$ is the communication cost incurred along the edge $e_{i,j} = (n_i, n_j) \in E$, which is zero if both nodes are mapped in the same processor. The value $\tau_i \in \mathcal{T}$ is the execution time of node $n_i \in V$.

A task is an indivisible unit of computation which may be an assignment statement, a subroutine or even an entire program. The tasks are convex which means that once a task starts its execution it can run to completion without interruption, Sarkar [14].

The task execution model is the static macro-dataflow model, Sarkar [14], Wu and Gajski [17], El-Rewini and Lewis [3]. The task execution is triggered by the arrival of all data from its predecessors. Immediately

A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors

Apostolos Gerasoulis and Tao Yang
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903.

Abstract

Clustering of task graphs has been used as an intermediate step to scheduling parallel architectures. In this paper, we identify important characteristics of clustering algorithms and propose a general framework for analyzing and evaluating such algorithms. Using this framework, we present an analytic performance comparison of four algorithms: the Dominant Sequence Clustering (DSC) [15], Kim and Browne's [9], Sarkar's [14], and Wu and Gajski's [17]. We identify the common features and differences of these algorithms and explain why DSC is superior to other algorithms. Finally, we present some experiments to verify our analysis.

1 Introduction

In this paper, we consider the clustering problem for directed acyclic graphs (DAGs). Clustering is a mapping of the nodes of a DAG onto labeled clusters. A cluster consists of a set of tasks while a task is an indivisible unit of computation. All tasks in cluster must execute in the same processor. The clustering problem has been shown to be NP-complete for a general task graph and for several cost functions. For example, if the cost function is the minimization of parallel time on a completely connected virtual architecture with an unbounded number processors, then clustering is NP-hard in the strong sense, Sarkar [14], Chretienne [2], Papadimitriou and Yannakakis [13].

Several heuristic algorithms have been proposed in the literature for the general clustering problem. Kim and Browne [10] considered linear clustering which is an important special case of clustering. Sarkar [14] presents a clustering algorithm based on a scheduling algorithm on unbounded number of processors. Wu and Gajski [17] developed a programming aid for hypercube architectures using scheduling techniques. Yang and Gerasoulis [15, 16] have proposed a fast and accurate heuristic algorithm, the Dominant Sequence Clustering (DSC). However, there has been little experimental and theoretical comparisons of clustering algorithms. One exception is the recent paper by El-Rewini and Lewis [3], where experiments with some