[3] Ph. Chretienne, " A Polynomial Algorithm to Optimally Schedule Tasks over an ideal Distributed System under Tree-like Precedence Constraints," *European Journal of Operational Research*, Vol. 2:43, pp. 225-230, 1989.

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

[5] J. Y. Colin and Ph. Chretienne, *C.P.M. Scheduling with Small Communication Delays and Task Duplication,* Report, M.A.S.I. 90.1, Universite Pierre et Marie Curie, 1990.

[6] M. Cosnard, M. Marrakchi, Y. Robert, and D. Trystram, " Parallel Gaussian Elimination on an MIMD Computer," *Parallel Computing,* Vol. 6, pp. 275-296, 1988.

[7] M. R. Garey and D.S. Johnson, Computers and Intractability, *A Guide to the Theory of NP-completeness*, W.H. Freeman and Company, 1979.

[8] G. A. Geist and M.T. Heath, " Matrix Factorization on a Hypercube Multiprocessor," *Hypercube Multiprocessors*, SIAM, pp. 161-180, 1986.

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

[10] A. Gerasoulis, S. Venugopal, and T. Yang, " Clustering Task Graphs for Message Passing Architectures," *Proceedings of ACM International Conference on Supercomputing,* Amsterdam, pp. 447-456, 1990.

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

[12] A. Gerasoulis and T. Yang, *A Comparison of Clustering Heuristics for Scheduling DAGs on Multiprocessors,* Report, Dept. of Computer Science, Rutgers Univ., August 1991.

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

[14] R. L. Graham, " Bounds on Multiprocessing Timing Anomalies," *SIAM J. Appl. Math.,* vol. 17, pp. 416-429, 1969.

[15] J. K. Lenstra and A. H. G. Rinnooy Kan, " Complexity of Scheduling under Precedence Constraints," *Operation Research,* Vol. 26:1, 1978.

[16] H. Kasahara and S. Narita. " Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers,* Vol. C-33, pp. 1023-1029, 1984.

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

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

[19] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software,* pp. 23-32, Jan. 1988.

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

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

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

[23] C. Picouleau, *Two new NP-Complete Scheduling Problems with Communication Delays and Unlimited Number of Processors,* M.A.S.I, Universite Pierre et Marie Curie Tour 45-46 B314, 4, place Jussieu, 75252 Paris Cedex 05, France, 1991.

[24] Y. Saad, "Gaussian Elimination on Hypercubes," *Parallel Algorithms and Architectures,* Cosnard, M. et al. Eds., Elsevier Science Publishers, North-Holland, 1986.

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

[26] H. Stone, *High-Performance Computer Architectures*, Addison-Wesley, 1987.

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

[28] Min-You Wu and D. Gajski, "A Programming Aid for Hypercube Architectures," *The Journal of Supercomputing,* Vol. 2, pp. 349-372, 1988.

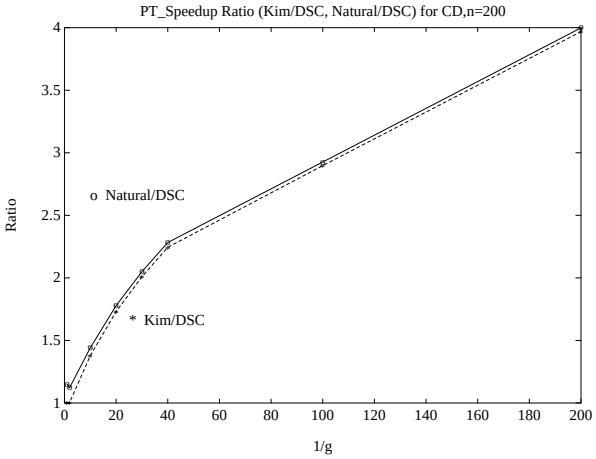
PT_Speedup Ratio (Kim/DSC, Natural/DSC) for CD,n=200
Ratio
1/g
o Natural/DSC
* Kim/DSC
4
3.5
3
2.5
2
1.5
1
0
20
40
60
80
100
120
140
160
180
200

Figure 9: Scheduling Choleski decomposition DAG.

takes hours. This of course is expected because of the complexities of the algorithms, $O(e(v+e))$ for Sarkar's vs. $O((v+e)\log v)$ for DSC.

5.2 Choleski Decomposition DAG

In the second example we use a well known numerical computing DAG, the Choleski decomposition (CD) DAG given in Cosnard et al. [6], Gerasoulis and Nelken [9]. The *natural clustering* is a special clustering widely used in the literature for the solution of CD in hypercube architectures, see Saad [24], Ortega [21], Geist and Heath [8] and others. It is derived by assuming that in each cluster all tasks update the same row (or column). The natural clustering is a linear clustering.

In addition to the natural clustering we use Kim's algorithm which is also a linear clustering algorithm, and DSC. In Figure 9 we show the ratio of the parallel times of Kim's and natural clustering over DSC which we call the *PT_speedup* ratio shown as (Kim/DSC and Natural/DSC). The matrix is of dimension 200×200 which implies that the graph has about $v = 2000$ nodes and $e = 4000$ edges. The x-axis is the inverse granularity ratio $1/g$ of the graph. The larger the $1/g$ the finer the granularity of the DAG, and the finest granularity is when $1/g = 200$.

When $1 \leq 1/g \leq 10$, the *PT_speedup* value is between 1 to 1.4 and the *PT_improv* is about 20%. The *PT_speedup* becomes larger along with the increase of $1/g$. This is expected because linear clustering is not appropriate for fine grain DAGs, which verifies our analysis in [11], see also Theorem 4.1.

In terms of complexity performance Kim's algorithm costs $O(v(v+e))$ which is impractical for very large DAGs.

5.3 DSC vs. a task duplication scheduling heuristic

The final result is the scheduling produced by DSC for a matrix multiplication tree DAG studied in Kruatrachue and Lewis [19] and shown in Figure 10(a). This is a fine grain DAG with all communication edge cost are equal to 212, the leaf nodes from 1 to 8 have computation times 101 and times for the other nodes from 9 to 15 are equal to 8.

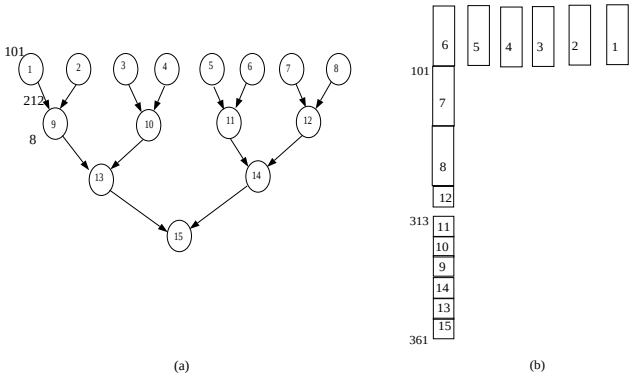

Figure 10: A matrix multiplication DAG and DSC schedule.

In figure 10(b) we show the scheduling derived by our DSC algorithm. The parallel time is equal to 361 which is equal to the parallel time of the duplication scheduling heuristic DSH of Kruatrachue and Lewis. This result is surprising since DSC algorithm does not use task duplication and also its complexity is only $O(v\log v)$ for the tree DAG vs. $O(v^4)$ for the DSH algorithm.

6 Conclusions

We have demonstrated that DSC is a superior algorithm in terms of both complexity and performance. Considering the importance of communication for the new scalable architectures, such as nCUBE II and INTEL i860, we expect DSC to be useful in scheduling such architectures. Particularly, since the new communication technology, such as wormhole communication, makes the aforementioned architectures look like fully connected architectures. Preliminary results with the nCUBE II hypercube are encouraging. DSC can also be used as the first step in the two step scheduling method as discussed in Section 2.

Acknowledgments

We thank Vivek Sarkar for providing us with the programs of his system in [25] and his comments and suggestions on this paper. We also thank Stewart Samuels from nCUBE corporation for providing us with time on the nCUBE II hypercube.

References

[1] F. D. Anger, J. Hwang, and Y. Chow, " Scheduling with Sufficient Loosely Coupled Processors," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 87-92, 1990.

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

that $CT(n_1) + c_{1,x}$ is the maximum value. Then DSC will zero edge (n_1, n_x) and assign

$$ST(n_x) = \max(CT(n_1), \max_{2 \le j \le m}\{CT(n_j) + c_{j,x}\}).$$

To prove $ST(n_x)$ is optimal, we need to compare it with an optimal schedule. Since the tree is coarse-grain, by Theorem 4.1, there exists a linear clustering with an optimum schedule S^*. Let $CT^*(n_j)$ be the completion time of n_j in schedule S^*, then $CT^*(n_j) \ge CT(n_j)$ for $1 \le j \le m$ from the above results. Now we will show that $ST^*(n_x) \ge ST(n_x)$ where $ST^*(n_x)$ is the starting time of n_x in S^*. There are two cases:

Case 1) If in S^* the zeroed incoming edge of n_x is (n_1, n_x).

$$ST^*(n_x) = \max(CT^*(n_1), \max_{2 \le j \le m}\{CT^*(n_j) + c_{j,x}\})$$

$$\ge \max(CT(n_1), \max_{2 \le j \le m}\{CT(n_j) + c_{j,x}\}) = ST(n_x).$$

Case 2) If in S^* the zeroed incoming edge of n_x is not (n_1, n_x), say it is (n_m, n_x). Thus

$$ST^*(n_x) = \max(CT^*(n_m), \max_{1 \le j \le m-1}\{CT^*(n_j) + c_{j,x}\})$$

$$\ge \max(CT(n_m), \max_{1 \le j \le m-1}\{CT(n_j) + c_{j,x}\}).$$

Because we have assumed

$$CT(n_1) + c_{1,x} \ge \max_{2 \le j \le m}\{CT(n_j) + c_{j,x}\},$$

then

$$ST^*(n_x) \ge CT(n_1) + c_{1,x} \ge ST(n_x).$$

That shows the starting time of n_x is optimal for a coarse grain in-tree with $d = k$. ■

DSC solves an in-tree in time $O(v \log v)$ where v is the number of nodes in this tree. Chretienne [3] developed an $O(v^2)$ dynamic programming algorithm and Anger, Hwang and Chow [1] proposed an $O(v)$ algorithm for tree DAGs with the condition that all communication edge weights are smaller than the task node execution weights, which are special cases of coarse grain tree DAGs. These two algorithms are only specific to this kind of trees.

For an out-tree, we can always reverse the DAG and use DSC to schedule the reversed graph. Then the optimal solution can be obtained. We call this approach the backward dominant sequence clustering.

5 Experimental results and comparisons

In this section, we describe our experiments and comparisons of DSC with other clustering algorithms. We have implemented the DSC algorithm in our automatic scheduling and code generation tool PYRROS.

5.1 Sarkar's vs. DSC on random DAGs

We have run Sarkar's algorithm and the DSC algorithm using 100 DAGs. These graphs are produced by randomly generating the number of tasks and their dependence edges and assigning random numbers to edge and task weights. The results are summarized in the following table.

Group	G1	G2	G3
#cases	22	47	31
Min-Max v	44-98	103-198	250-540
Average v	70.1	148.7	329.0
Average e	311.3	502.2	3430
Min-Max C/R	0.83-5.6	0.27-7.6	1.68-8.7
Min-Max PT_Impro	4.6%-33.5%	3.3%-31%	21%-37.8%
Average PT_Impro	17.8%	21.6%	25.8%
Complexity speedup	92.5	229.6	1854.5

The explanation are described below:

- In the first row the task are categorized into three groups, G1, G2 and G3 in terms of their number of nodes. G1:$v < 100$, G2: $100 \le v < 200$ and G3: $v \ge 200$.
- Row 2 is the number of cases for each group. Row 3 is the minimum and maximum number of nodes v for each randomly generated case of a group. Row 4 is the average value of v. Row 5 is the average value of edge number e.
- Row 6 is the minimum and maximum values of C/R where C/R is the ratio of communication over computation, which we consider as an estimation of the inverse of the granularity $1/g$ of the graph. It is measured by dividing the total transmission delay by the total computation in the critical path.
- Row 7 is the minimum and the maximum of the parallel time improvement ratio. Row 8 is the average value. The improvement ratio of DSC over algorithm A is defined as

$$PT\_Improv = 1 - \frac{PT_A}{PT_{dsc}}.$$

- Row 9 is the complexity speedup of the algorithm. This represents the ratio between the computing time spent by Sarkar's algorithm and the time spent by DSC.

This experiment shows that the parallel time improvement ratio of DSC over Sarkar's algorithm is about 20%. However, in terms of computing cost, DSC is superior particularly for large graphs. For a DAG with hundreds of nodes, it is shown that the algorithm speedup is about 1854. For such kind of DAGs, DSC takes few seconds to schedule while Sarkar's algorithm

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

The *grain* of a task is defined as

$$g_x = \min\{g(F_x), g(J_x)\}$$

and the *granularity* of a DAG as

$$g(G) = \min_{n_x \in V}\{g_x\}.$$

We call a DAG *coarse grain* if $g(G) \geq 1$, otherwise *fine grain*. If $\tau_k = R$ and $c_{i,k} = C$ then the grain of every task and the granularity of the DAG reduces to the ratio R/C which is the same as Stone's definition [26]. For coarse grain DAGs each task receives or sends a small amount of communication compared to the computation of its adjacent tasks.

In [11], we prove the following theorem:

Theorem 4.1 *For any nonlinear clustering of a coarse grain DAG, there exists a linear clustering with less or equal parallel time.*

This theorem shows that an optimal linear clustering will give the optimum solution for scheduling a coarse grain DAG. Using the above theorem and the NP-completeness results for coarse grain DAGs by Picouleau [23], we can show that the linear clustering problem is NP-complete for the minimization of the parallel time cost function. Fortunately, for coarse grain DAGs, DSC or any other linear clustering algorithm, guarantee performance within a factor of two of the optimum, see Gerasoulis and Yang [11]. These results provide a partial explanation of the popularity of linear clustering in the literature, e.g. see Kung [20], Kim and Browne [18], Saad [24], Ortega [21].

4.2 Performances on join and fork

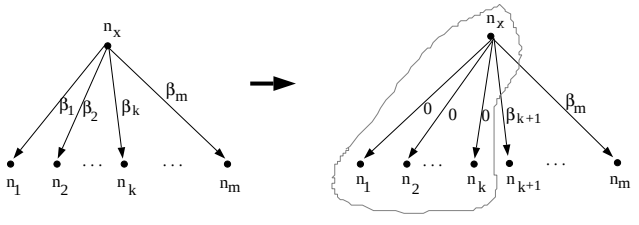

Figure 8: Fork clustering

Theorem 4.2 *DSC derives the optimal solution for a fork and join.*

Proof: A fork is shown in Figure 8(a). After n_x is scheduled, DSC will examine free nodes $n_1, n_2, \cdots, n_m$ in a decreasing order of their priorities. Assume without loss of generality that that $\beta_k + \tau_k \geq \beta_{k+1} + \tau_{k+1}$ for $1 \leq k \leq m-1$. Then $priority(n_k) = \beta_k + \tau_k$ implying that the nodes will be sorted as $n_1, n_2, \cdots, n_m$ in FL.

We now determine the optimum time for the fork and then show that DSC achieves this optimum. Assume the optimum parallel time to be PT_{opt}. If $\tau_x + \tau_k + \beta_k > PT_{opt}$ for some k then $\beta_i = 0,\ i = 1:k$, otherwise we have a contradiction. This implies that all edges to the left, see Figure 8, which have priority greater than PT_{opt} must be zeroed by any optimum algorithm. By assuming $\beta_{m+1} = \tau_{m+1} = 0$, then the optimum value PT_{opt} is:

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

DSC zeroes edges from left to right as many as possible up to the point k such that:

$$\sum_{j=1}^{k}\tau_j \leq \beta_k + \tau_k, \qquad \sum_{j=1}^{k+1}\tau_j \geq \beta_{k+1} + \tau_{k+1}.$$

Then the parallel time after zeroing k edges in the left portion is reduced to the minimum value PT_{opt}.

For a join, the DSC uses the optimum algorithm [11] for its minimization and thus it is optimum by definition. ■

Chretienne [2] describes an algorithm for scheduling a fork with a complexity of $O(m \log B)$ where $B = min\{\sum_{i=1}^{m}\tau_i, \beta_1 + \tau_1\} + \tau_x$. This algorithm is for forks only. For a comparison, the DSC costs $O(m \log m)$ for the fork.

4.3 Performances on in and out-trees

Scheduling in and out-trees is still NP-complete in general as shown by Chretienne [4] and DSC will not give the optimal solution. However, DSC will yield the optimal solution for a coarse grain in-tree.

Theorem 4.3 *DSC gives the optimal solution for a coarse grain in-tree.*

Proof: We claim that for any in-tree, DSC will give a schedule where every node has the minimum starting time. We prove it by induction on the depth of the tree (d).

When $d = 0$, it is trivial.

Assuming it is true for $d < k$.

When $d = k$, let the predecessors of root n_x be $n_1, \cdots, n_m$. Since each sub-tree rooted with n_i has the depth $< k$ and the disjoint subgraphs cannot be clustered together by DSC, DSC can obtain the minimum starting time, and hence the minimum completion time $CT(n_j)$, for each n_j where $1 \leq j \leq m$ according to the induction hypothesis.

When n_x is free, its startbound is

$$startbound(n_x) = \max_{1 \leq j \leq m}\{CT(n_j) + c_{j,x}\}.$$

When n_x is selected, $startbound(n_x)$ is minimized by DSC. DSC will only zero one edge because the graph is coarse grain[5]. Without loss of generality, assume

[5]For a coarse grain join with root n_x, zeroing more than one edge cannot decrease $startbound(n_x)$ but it could increase $startbound(n_x)$.

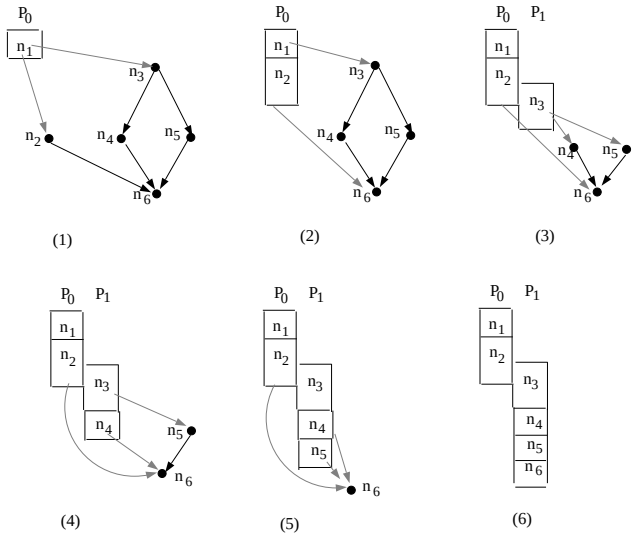

Figure 6: DSC clustering steps for the DAG shown in Figure 1(a).

- Initially
  $USG = \{n_1, n_2, n_3, n_4, n_5, n_6\}$ $PT_0 = 10.5$
  $FL = \{n_1^{0+10.5}\}$ $PFL = \{\}$
- Step 1, n_1 is selected, $startbound(n_1) = 0$, it cannot be reduced so n_1 is scheduled to P_0. Then
  $USG = \{n_2, n_3, n_4, n_5, n_6\}$ $PT_1 = 10.5$
  $FL = \{n_2^{4+6.5}, n_3^{2+8}\}$ $PFL = \{\}$
- Step 2, n_2 is selected ($n_x = n_2$, $n_y = NULL$, and $startbound(n_2) = 4$). By zeroing the incoming edge (n_1, n_2) of n_2, $startbound(n_2)$ reduces to 1. Thus this zeroing is accepted and after that step,
  $USG = \{n_3, n_4, n_5, n_6\}$ $PT_2 = 10$
  $FL = \{n_3^{2+8}\}$ $PFL = \{n_6^{6.5+1}\}$
- Step 3, n_3 is selected ($n_x = n_3$ with $priority(n_3) = 2 + 8 = 10$ and $n_y = n_6$ with $priority(n_6) = 7.5$). By zeroing the incoming edge (n_1, n_3), $startbound(n_3)$ increases from 2 to 3.5. Thus this edge cannot be zeroed and a new processor P_1 is opened for n_3. Then
  $USG = \{n_4, n_5, n_6\}$ $PT_3 = 10$
  $FL = \{n_4^{7+3}, n_5^{7+3}\}$ $PFL = \{n_6^{6.5+1}\}$
- Step 4, n_4 is selected and its incoming edge (n_3, n_4) is zeroed so that it can start at 4.5 instead of 7. Then
  $USG = \{n_5, n_6\}$ $PT_4 = 10$
  $FL = \{n_5^{7+3}\}$ $PFL = \{n_6^{6.5+1}\}$
- Step 5, n_5 is selected and its incoming edge (n_3, n_5) is zeroed so that it can start at 5.5 instead of 7.
  $USG = \{n_6\}$ $PT_5 = 8.5$
  $FL = \{n_6^{7.5+1}\}$ $PFL = \{\}$
- Step 6, n_6 is selected and its incoming edge (n_5, n_6) dominates the $startbound(n_6)$ and by zeroing that edge n_6 can start at time 6.5 instead of 7.5.
  $USG = \{\}$ $PT_6 = 7.5$
  $FL = \{\}$ $PFL = \{\}$

Finally two clusters are generated with $PT = 7.5$.

4 Optimality of DSC for primitive task graphs

In this section we study the performance of DSC for some special classes of DAGs: join, fork and coarse grain trees. The reason for considering such primitive structures is that a DAG is composed of sets of join and fork components. Also spanning trees of a DAG are in and out-trees. Therefore, by studying the DSC performance on such structures we can further understand its behavior. Other general clustering algorithms proposed by Sarkar [25], Kim [17], Kim and Browne [18], and Wu and Gajski [28] have complexity $O(v^2)$ or higher. As we will show DSC attains the optimum for the above mentioned primitive structures. As far as we know no other general algorithm has this property. A detailed comparison of several clustering algorithms with DSC is given in [12].

4.1 Definition of primitive structures and coarse grain DAGs

An *in-tree* is a directed tree in which the root has outgoing degree zero and other nodes have the outgoing degree one.

An *out-tree* is a directed tree in which the root has incoming degree zero and other nodes have the incoming degree one.

A *join* is an in-tree of depth 1 as shown in Figure 7(a).

A *fork* is an out-tree of depth 1 as shown in Figure 7(b).

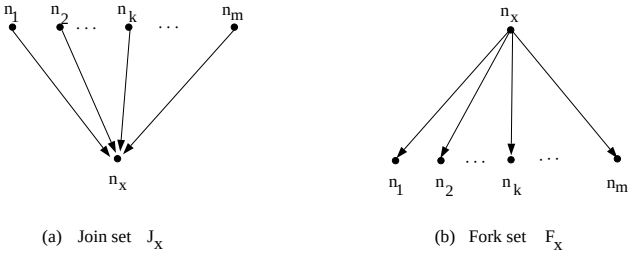

Figure 7: Join and fork.

A DAG consists of *fork*, *join* or both. In [11], we have defined the grain of DAG as follows:
For a join J_x and a fork F_x let

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

$USG = V$;
WHILE $USG \neq \emptyset$ **DO**
  $n_x = head(FL)$; /*The free task with highest priority*/
  $n_y = head(PFL)$; /*The partial free task with highest priority. */
  IF $(priority(n_x) \geq priority(n_y))$ **THEN**
    Minimize $startbound(n_x)$ under 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, schedule n_x to a new processor.
  ELSE
    Minimize $startbound(n_x)$ under constraint CT1&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, schedule n_x to a new processor.
  ENDIF
  Put n_x into SG and $ST(n_x) = startbound(n_x)$.
ENDWHILE

Figure 5: DSC Algorithm.

Proof: We omit the proof here and simply show its correctness for the initial step ($i = 0$) where $SG = PFL = \{\}$ and $FL = \{$all entry nodes$\}$. Then $PT_0 = max\{priority(n_x)\} = level(n_x)$ = the length of the critical path. See [27] for details. ■

The last theorem is used by DSC to identify edges in DS incrementally at each step[4]. There are two cases, either there is a DS that goes through USG or there is not. If not, that indicates that all DS nodes have been examined. Since we have assumed non-backtracking these DS nodes cannot be re-examined. If there is a DS that is going through USG then the above theorem implies that

$$PT_i = \max\{priority(n_x),\ priority(n_y)\}.$$

Thus one DS must go either through the head of the free list or the head of the partial free list. Therefore we have the following two cases, corresponding to the **IF** part of the algorithm:

1. *DS is going through n_x, the head of FL.* In this case, $PT_i = priority(n_x) \geq priority(n_y)$. DSC will select n_x at step $i + 1$. Then DSC will try to reduce its $startbound(n_x)$. If successful, the length of DS is reduced.

2. *DS is only going through n_y, the head of PFL.* In this case $PT_i = priority(n_y) > priority(n_x)$. Which implies that no DS is going through FL and there is one DS passing through partial free node n_y. The DSC cannot schedule n_y at step i+1 because its predecessors have not been scheduled. Therefore the reduction of the length of this DS must be postponed until n_y becomes free.

   In the meantime, DSC has picked up to schedule the current free head node n_x which is not in DS. Since zeroing the incoming edges of n_x to minimize $startbound(n_x)$ could affect the reduction of $startbound(n_y)$, which is the important node at this step, we must make sure that such a situation does not occur. Therefore, we impose the following constraint on DSC:

   - CT2: Zeroing incoming edges of n_x to minimize $startbound(n_x)$ should not affect the reduction of $startbound(n_y)$, if it is reducible, in some future step of the algorithm.

   Detecting the reducibility of $startbound(n_y)$ is implemented in DSC by examining the result of the zeroing of the incoming edge of n_x in DS. If reducibility of $startbound(n_y)$ is detected at the present step then it will be reducible at some future step because of constraint CT2. Thus, when n_y becomes free we are guaranteed that $startbound(n_y)$ can be reduced at that time and hence the DS can be compressed at that time.

[4] The DS can be identified using the theorem without actually having to compute PT_i.

Another important part of the DSC algorithm is the minimization procedure for the $startbound(n_x)$. This is similar to the derivation of the optimal clustering for a join DAG shown in [11], see also section 4 for the fork optimum algorithm. We select for zeroing the edges of all predecessors which are the only tasks in a cluster and exclude predecessors which have been scheduled in clusters with more than one task. We then apply the optimum join clustering algorithm. We continue zeroing until $startbound(n_x)$ is reduced to the maximum possible degree as long as CT1, and CT1&CT2 are not violated.

We now determine the computational complexity of DSC. The incoming edge zeroing 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 estimate of $O(e \log v)$. Maintaining FL and PFL priority lists cost $O(\log v)$ for each step if the balanced search tree data structure is used. Since there are v steps the cost is $O(v \log v)$. 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 $O(v \log v + e)$.

3.3 A running trace of DSC

For the task graph in Figure 1(a), DSC clustering steps shown in Figure 6 are explained below. The superscript of a task in FL or PFL indicates its priority. A dash arrow in Figure 6 is an edge from a scheduled node to an unscheduled free node or partial free node and that edge affects the *startbound* value of that unscheduled end node.

try to reduce the length of the dominant sequence. We demonstrate this idea by showing few clustering steps on the DAG in Figure 1(a). Initially the dominant sequence is the same as the critical path which is $< n_1, n_2, n_6 >$. To reduce the length of that DS, we could zero anyone of the two edges or both edges. Assume one edge (n_1, n_2) is zeroed, then the dominant sequence changes to $< n_1, n_3, n_4, n_6 >$. At the next step, we need to identify a new DS and also decide which edge(s) will be zeroed. We present a brief discussion of some problems observed in this example and how DSC addresses these problems.

1. Given a DS, there is a decision to be made for selecting an edge(s) to be examined[2]. Note that zeroing one edge will suffice to change the dominant sequence. Thus it is not necessary to zero more than one edge per step. However, there are still many possibilities even for selecting one edge. One can select the top edge in DS or the edge with the highest weight and so on. Because it is hard to judge which edge should be selected, such that zeroing that edge will lead to the optimal solution, our edge selection criterion is to choose the one that will make it easier to determine a DS at the next step.

2. Determining a DS for a clustered DAG could take $O(v + e)$ time, if the computation is not done incrementally. Repeating this computation for all steps will result in at least $O(v^2)$ complexity. Such algorithms will be very time consuming for large DAGs. DSC uses an incremental mechanism that re-uses the previous information to determine the new DS, after each edge-zeroing step, in $O(\log v)$ time. This is because DSC zeros only one edge in DS and as a result only two nodes in DS need to be identified at each step.

3.2 A detailed algorithm description

The edge zeroing in the DSC algorithm is guided by an infinite-processor scheduling mechanism. This control mechanism performs v scheduling steps and at each step it selects a free task and tries to zero some of its incoming edges.

A node is called *scheduled* if it has been scheduled into a processor. SG is the set of all scheduled nodes and USG is the set of all unscheduled nodes. Initially, USG is the set V. A node is called *free*[3] if it is unscheduled and all of its predecessors have been scheduled. A node is called *partial free* if it is unscheduled and at least one of its predecessors has been scheduled but not all of them have been scheduled.

Let $level(n_x)$ be the length of the longest path from n_x to a bottom (exit) node, including nonzero communication edge costs in that path. $PRED(n_x)$ and $SUCC(n_x)$ are the sets of immediate predecessors and successors of n_x respectively.

For a scheduled task n_x, define:

- $ST(n_x)$ the starting time
- $CT(n_x) = ST(n_x) + \tau_x$ the completion time.

For a free or partial free task n_x, define:

- $arrivetime(n_j, n_x) = CT(n_j) + c_{j,x}$, where n_j is a scheduled predecessor of n_x.
- $startbound(n_x) = max\{arrivetime(n_j, n_x)\}$, where $n_j \in SG \bigcap PRED(n_x)$. This value is the lower bound for starting n_x and it is zero if n_x is an entry node.
- $priority(n_x) = startbound(n_x) + level(n_x)$.

At each scheduling step, we maintain two node lists, a partial free list PFL and a free list FL sorted in a descending order of their task priorities. The tie resolution strategy follows the most immediate successors first (MISF) principle [16]. 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 $priority(NULL) = 0$.

Notice that when a free task n_x is scheduled, its starting time must satisfy $ST(n_x) \geq startbound(n_x)$. However, $startbound(n_x)$ could be reduced if some of its incoming edges are zeroed. Thus, if n_x is in DS reducing $startbound(n_x)$ could allow this node to start earlier and hence the length of the current DS could be reduced. On the other hand, if the zeroing operation increases the $startbound(n_x)$, the length of DS will also increase. Therefore we impose a safe-guard constraint to avoid this increase:

- CT1: If the incoming edge zeroing operation for a node n_x increases $startbound(n_x)$, the DSC algorithm rejects such a zeroing.

The DSC algorithm is described in a compact form in Figure 5.

We continue with our explanations of the DSC clustering algorithm which is implemented as a scheduling algorithm. Each scheduling **WHILE** loop step corresponds to a clustering step. The corresponding clustered graph consists of all nodes in USG, each node constitutes a cluster, plus all nodes in SG for which the scheduled nodes in each processor constitute a cluster.

To determine the parallel time for a clustering at each step a scheduling algorithm must be used. We would like to use the already existing schedule for the nodes in SG to determine PT_i at the next step. Therefore, we have chosen the scheduling heuristic that either schedules a task after the last scheduled task in a cluster or schedules the task in a new cluster. This assumption implies the following result, where $n_x = head(FL)$ and $n_y = head(PFL)$ after step i:

Theorem 3.1 *The parallel time for executing the clustered graph after step i of DSC clustering is:*

$$PT_i = \max\{priority(n_x),\ priority(n_y),\ \max_{n_s \in SG}\{ST(n_s) + level(n_s)\}\}.$$

[2] Backtracking is not used to avoid high complexity.

[3] A free node is different than a ready node widely used in the literature. A ready node is free and all of its data have arrived locally. Therefore, it is ready to start its execution.

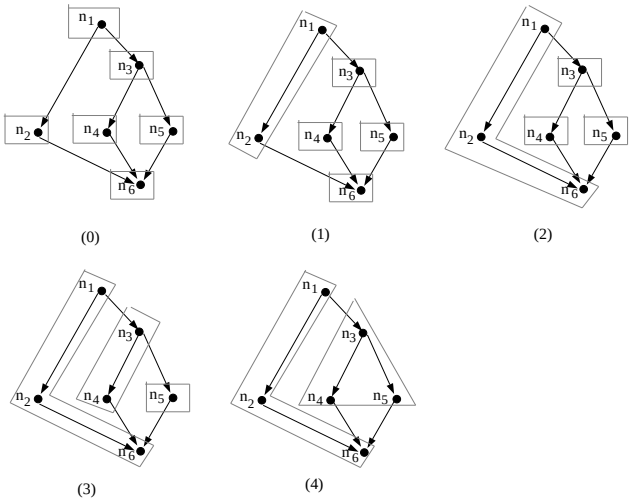

Figure 2: Clustering steps by Sarkar's algorithm for Figure 1(a).

after step i. Initially each task is in a separate cluster as shown in Figure 2(0) and the parallel time $PT_0 = 10.5$, which is the same as the length of the critical path.

At step 1, edge (n_1, n_2) is examined and the parallel time reduces to 10 if this edge is zeroed. Thus this zeroing is accepted. In step 2, 3 and 4, shown in Figure 2(2), (3) and (4), all examined edges are zeroed since each zeroing does not increase the parallel time. At step 5, edge (n_1, n_3) is examined and by zeroing it the parallel time increases from 8.5 to 9. Thus this zeroing is rejected. Similarly, at step 6 and 7 the zeroings are rejected. Finally two clusters are generated with parallel time 8.5.

step i	edge examined	PT if zeroed	zeroing	PT_i
0				10.5
1	(n_1, n_2)	10	yes	10
2	(n_2, n_6)	10	yes	10
3	(n_3, n_4)	10	yes	10
4	(n_3, n_5)	8.5	yes	8.5
5	(n_1, n_3)	9	no	8.5
6	(n_4, n_6)	9	no	8.5
7	(n_5, n_6)	9	no	8.5

Figure 3: Clustering steps corresponding to Figure 2.

3 Dominant sequence clustering algorithm

3.1 Dominant sequence and design considerations for DSC

As we saw in the previous section, Sarkar's algorithm zeroes the highest communication edge. This edge, however, might not be in the path that determines the parallel time and as result the parallel time might not be reduced at all. In order to design a better clustering algorithm, we must examine the schedule[1] of a clustered graph to identify those edges that contribute to the parallel time. For linear clusterings the parallel time can be easily identified by finding the critical path. However, if the clustering is nonlinear, the parallel time cannot be determined by the critical path. For instance, one critical path in Figure 1(c) is $< n_1, n_2, n_6 >$ with length 7.5 but it is impossible to execute this clustered DAG within that time.

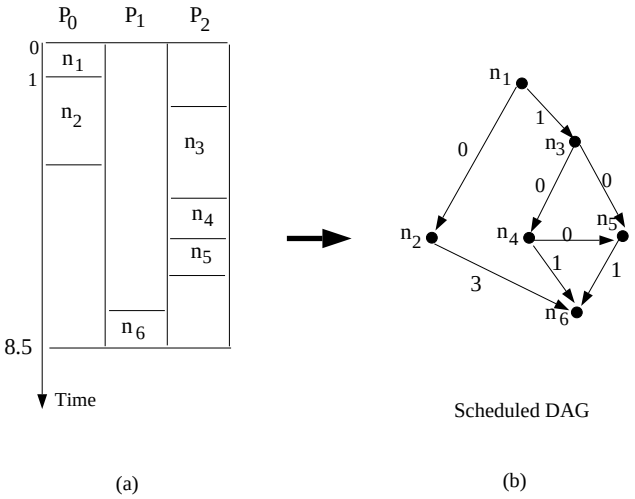

Figure 4: The Gantt chart and the scheduled graph corresponding to Figure 1(c).

The following procedure discovers the task sequence that determines the parallel time. A schedule for a clustered DAG defines the execution order of tasks within each cluster. We modify the clustered DAG as in [25] by adding zero-weighted edges between any pair of two nodes n_x 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. The modified graph is called the *scheduled DAG*. For example, Figure 4(a) is a schedule for the clustered graph of Figure 1(c) and Figure 4(b) is the corresponding scheduled graph. The critical path of the scheduled graph determines the parallel time. We call this path a *dominant sequence* of the clustered graph, *DS* for short. For instance, $< n_1, n_3, n_4, n_5, n_6 >$ is a dominant sequence of the clustered graph shown in Figure 1(c) corresponding to Figure 4. For linear clustering, a dominant sequence is the same as a critical path.

Since the parallel time is determined by a *DS*, reducing the length of *DS* can effectively reduce the parallel time. The basic idea behind the DSC algorithm is to do a sequence of edge zeroings and at each step

[1] Since the optimal schedule is intractable, a good approximation algorithm for scheduling a clustered DAG is required. Algorithms based on critical path information have been shown to perform well for a wide range of DAGs.

2 The clustering and scheduling problem

A directed acyclic task graph (DAG) is defined by a tuple $G = (V, E, \mathcal{C}, \mathcal{T})$ where 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 $n_i \in V$.

Given a DAG and an unbounded number of completely connected processors, the scheduling problem is equivalent to two subproblems: (1) processor assignment and (2) task ordering for each processor. The processor assignment is also known as the *clustering* problem when there is no limitation on the number of processors. Formally, a clustering is to determine a mapping of task nodes onto a labeled set of clusters $\{C_1, C_2, \cdots, C_r\}$. Tasks in the same cluster will be executed in the same processor.

Task clustering has been used in the two-step method for scheduling tasks on parallel architectures with a bounded number of processors which may or may not be completely connected, [9, 13, 18, 25, 28]. The two-step method is:

1. Perform task clustering
2. Schedule the clusters on p physical processors

Such an approach is widely used in parallel numerical computing [8, 9, 21, 24]. Clustering has also been used in VLSI processor array design where it is known as the processor projection, [20]. A common cost function for clustering is the minimization of the parallel time. A comparison of clustering heuristics is given in [10, 12]. Sarkar calls clustering the *internalization pre-pass* and his argument for using clustering is given in [25], p. 124:

> " .. if two tasks are assigned to the same processor in the best case situation with unbounded number of processors available and the lowest possible overhead, then they must be assigned to the same processor in any schedule on the target architecture".

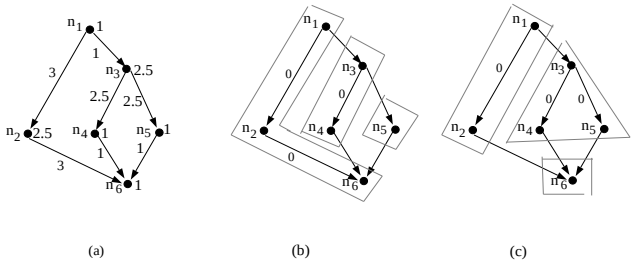

Figure 1: Clustering example

A DAG with a given clustering is called a *clustered graph*. A communication edge weight in a clustered graph becomes zero if the start and end nodes of this edge are in the same cluster. The *critical path* of a clustered graph is the longest path in that graph including nonzero communication cost. A clustering example is shown in Figure 1. Figure 1(a) shows a DAG with 6 tasks $n_1, n_2, \cdots, n_6$. Their execution times are in the right side of the bullets and edge weights are written in the edges. Figure 1(b) shows a clustered graph with 3 clusters. The critical path of that clustered graph is $< n_1, n_3, n_5, n_6 >$ with length 10.

A clustering is called *linear* if the nodes of each cluster constitute a linear path without considering the transitive edges. Otherwise it is called *nonlinear*, where independent tasks could belong in the same cluster. Figure 1(b) shows a linear clustering while Figure 1(c) shows a nonlinear clustering where independent tasks n_4 and n_5 are in the same cluster.

In determining the parallel time, the sequentialization of independent tasks in a nonlinear cluster is required. For Figure 1(c), the parallel time of scheduling this clustered graph is 8.5 when n_4 is executed either before or after n_5. From this example, we can see that linear clustering preserves the parallelism embedded in the DAG, but nonlinear clustering reduces parallelism by sequentializing parallel tasks. A further discussion of this issue can be found in [11].

2.1 Successive clustering refinements

Most of the previously developed clustering algorithms can be considered as performing a sequence of clustering refinements [10, 12]. At the initial step, each task is assumed to be in a separate cluster. Each step of a clustering algorithm tries to refine the previous clustering to achieve a designated goal. A typical refining operation is to zero an edge which connects two clusters. Such operation merges two clusters and zeros the communication cost represented by this edge since the start and end nodes of this edge will be scheduled in the same processor.

As an example of edge-zeroing clustering refinement, we consider Sarkar's algorithm [25], pp. 123-131. This algorithm sorts e edges in a decreasing order of edge weights first and performs e steps of clustering by examining edges from left to right in the sorted list. For each step, it examines one edge and zeros this edge if the parallel time does not increase.

The parallel time for a clustering with r clusters is determined by allocating r clusters onto r processors and finding the best execution ordering. This computation is still NP-complete, [7]. Sarkar gives an approximation algorithm which uses the level information, i.e. the longest path length from a task to the exit node, to execute the critical tasks first.

For the example shown in Figure 1(a), Sarkar's clustering steps are shown in Figure 2. The sorted edge list with respect to edge weights is $\{(n_1, n_2), (n_2, n_6), (n_3, n_4), (n_3, n_5), (n_1, n_3), (n_4, n_6), (n_5, n_6)\}$.

The table in Figure 3 traces the execution of this algorithm where PT stands for parallel time and PT_i is the parallel time for executing the clustered graph

A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors*

Tao Yang and Apostolos Gerasoulis
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
Email: tyang or gerasoulis@cs.rutgers.edu

Abstract

Scheduling parallel tasks on an unbounded number of completely connected processors when communication overhead is taken into account is NP-complete. Assuming that task duplication is not allowed, we propose a fast heuristic algorithm, called the dominant sequence clustering algorithm (DSC), for this scheduling problem. The DSC algorithm is superior to several other algorithms from the literature in terms of both computational complexity and parallel time. We present experimental results for scheduling general directed acyclic task graphs (DAGs) and compare the performance of several algorithms. Moreover, we show that DSC is optimum for special classes of DAGs such as join, fork and coarse grain tree graphs.

1 Introduction

Scheduling parallel tasks with precedence relations over distributed memory multiprocessors has been found to be much more difficult than the classical scheduling problem, see Graham [14] and Lenstra and Kan [15]. This is because data transferring between processors requires substantial transmission delays. Without imposing a limitation on the number of processors, the scheduling problem that ignores communication overhead is solvable in a polynomial time. When communication overhead is present, however, the problem becomes NP-complete, see Sarkar [25], Chretienne [2] and Papadimitriou and Yannakakis [22].

When task duplication is allowed and there is a sufficient number of completely-connected processors, Papadimitriou and Yannakakis [22] have proposed an approximate algorithm whose performance is within 50% of the optimum. Anger, Hwang and Chow [1], Colin and Chretienne [5] have also proposed polynomial optimal algorithms for special classes of task graphs in which communication is smaller than computation. Kruatrachue and Lewis [19] have given an algorithm that uses task duplication. It is not clear, however, whether the task duplication assumption is practical because it could result in a considerable increase in the space complexity.

*Supported by a Grant No. DMS-8706122 from NSF.

When task duplication is not allowed, Chretienne [3], Anger, Hwang and Chow [1] have proposed special algorithms to derive an optimal schedule for a coarse grain tree DAG. Less progress has been made in developing an optimum algorithm for graphs other than trees. Also as far as we know, there is no scheduling algorithm that works for general graphs but also finds optimal schedules for special classes of primitive graphs. In this paper, we propose such an algorithm.

We make the following assumptions:

1. Task duplication is not allowed
2. The number of available processors is unlimited
3. The processors are completely connected
4. The static macro dataflow model of computation, see Sarkar [25] and Wu and Gajski [28]. The task execution is triggered by the arrival of all data and at the completion of its execution the data are send in parallel to successor tasks.

Heuristic scheduling algorithms for arbitrary task graphs have been proposed in the literature, see Kim and Browne [18], Sarkar [25], Wu and Gajski [28]. However, the computational complexity for most of these algorithms is too high and also those algorithms cannot determine the optimum schedule for primitive DAGs. Our approach for solving the scheduling problem is to view the scheduling process as a *clustering* procedure with the goal of minimizing the overall parallel time, see Gerasoulis, Venugopal and Yang [10] and Gerasoulis and Yang [12].

Section 2 introduces the concept of clustering. Section 2.1 discusses a framework that considers a clustering procedure as performing a sequence of successive clustering refinements. Section 3 describes the heuristic used in DSC to achieve the goal of minimizing the parallel time. Section 4 shows that DSC can find optimal solutions for primitive DAGs such as join, fork and coarse grain trees. Section 5 presents experimental results that compares the performance of several algorithms from the literature.