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

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

[23] M. Wu and D. Gajski, A programming aid for hypercube architectures, *J. of Supercomputing*, 2(1988), 349-372.

[24] J. Yang, L. Bic and A. Nicolau, A mapping strategy for MIMD computers, *Proc. of 1991 Inter. Conf. on Parallel Processing*, Vol I, 102-109.

[25] T. Yang and A. Gerasoulis, A fast static scheduling algorithm for DAGs on an unbounded number of processors, *Proc. of IEEE Supercomputing '91*, Albuquerque, 1991, 633-642,

[26] T. Yang and A. Gerasoulis, PYRROS: Static task scheduling and code generation for message passing multiprocessors, *Proc. of 6th ACM Inter. Conf. on Supercomputing*, Washington D.C., July, 1992, 428-437.

[5] M. Cosnard, M. Marrakchi, Y. Robert and D. Trystram, Parallel Gaussian elimination on an MIMD computer, *Parallel Computing*, 6(1988), 275-296.

[6] A. Darte, Two Heuristics for Task Scheduling, Laboratoire LIP-IMAG, Ecole Normale Superieure de Lyon, 69364, 1991.

[7] T.H. Dunigan, Performance of the INTEL iPSC/860 and nCUBE 6400 Hypercube, ORNL TM-11790, Oak Ridge National Laboratory, Oak Ridge, TN, Nov. 1991.

[8] H. El-Rewini and T.G. Lewis, Scheduling parallel program tasks onto arbitrary target machines, *J. of Parallel and Distributed Computing*, 9(1990), 138-153.

[9] A. Gerasoulis, I. Nelken. Static scheduling for linear algebra DAGs. *Proc. of 4th Conf. on Hypercubes*, Monterey, Vol. 1, 1989, 671-674.

[10] A. Gerasoulis, S. Venugopal, and T. Yang, Clustering task graphs for message passing architectures, *Proc. of ACM Inter. Conf. on Supercomputing*, Amsterdam, 1990, 447-456.

[11] A. Gerasoulis and T. Yang, A Comparison of clustering heuristics for scheduling DAGs on multiprocessors, To appear in *Journal of Distributed and Parallel Computing*, special issue on scheduling and load balancing, Dec. 1992.

[12] A. Gerasoulis and T. Yang, A static macro-dataflow scheduling tool for scalable parallel architecuteres, *Proc. of Summer School on Scheduling*, Debonas, France, Oct. 1992.

[13] R. L. Graham, Bounds for certain multiprocessing anomalies, *Bell System Tech. J.* , 45 (1966), 1563-1581.

[14] J.A. Hoogeveen, S.L. van de Velde, and B. Veltman, Complexity of scheduling multiprocessor tasks with prespecified processor allocations, CWI, Report BS-R9211 June 1992, Netherlands.

[15] T.C. Hu, Parallel sequencing and assembly lines problems, *Operations Research*, 9 (1961), 841-848.

[16] H. Kasahara and S. Narita, Practical Multi-processor scheduling algorithms for efficient parallel processing, *IEEE Trans. on Computers*, C-33 (1984), 1023-1029.

[17] S. J. Kim and J. C. Browne, A general approach to mapping of parallel computation upon multiprocessor architectures, *Proc. Int'l. Conf. on Parallel Processing*, 1988, 1-8.

[18] B. Kruatrachue and T. Lewis, Static task scheduling and grain packing in parallel processing systems, Tech Report, Dept. of Computer Science, Oregon State Univ., 1987.

[19] V. M. Lo, S. Rajopadhye, S. Gupta, D. Keldsen, M. Mohamed, J. A. Telle, OREGAMI: software tools for mapping parallel computations to parallel architectures, *Proc. Int'l. Conf. on Parallel Processing*, 1990, Vol. II 88-92.

[20] T. F. Ngai, Runtime resource management in concurrent systems, Tech. Report, CSL-TR-92-504, Stanford Univ., Jan. 1992.

Since $FC''_{s+1}(u) = \max(FC''_s(u), RT(s+1)) + \tau_{s+1}$ and we know $FC_s(u)$ is the minimum, $FC''_{s+1}(u) \geq \max(FC_s(u), RT(s+1)) + \tau_{s+1} = FC_{s+1}(u)$, which is a contradiction. Thus the statement for part 2 is true for $k = s+1$. ■

B Proof of Theorem 5

For Figure 8(b), we first find an optimal schedule, then we show RCP^* derives the same schedule.

The optimal schedule S for any DAG G can always be reversed to be also optimal for the reversed DAG as follows: 1) Reverse the edge directions of G, the reversed DAG is called G^r. 2) Reverse the execution order of tasks in each processor in schedule S, the new schedule is called G^r. Then S^r is an optimal schedule for G^r.

Notice that a join in Figure 8(b) is the reversed DAG of a fork shown in (a). In part 2 of case 2 of the proof for Theorem 4, we have shown an optimal schedule B for a fork in Figure 8(a), which executes leaf tasks in each processor in an increasing order of the ready time $\tau_x + c_{x,j}$. Then the join, its reversed DAG in Figure 8(b), will have an optimal schedule B^r that executes leaf tasks in each processor in a descending order of $\tau_x + c_{x,j}$.

When RCP^* is used to schedule this join, all leaf tasks are ready for execution at the beginning and root n_x cannot be started unless all leaf tasks have been executed. Thus for each processor that contains leaf tasks, leaf tasks will be executed in a descending order of their priorities $L^*(j)$. Since $L^*(j) = c_{j,x} + \tau_x$ for each leaf task n_j and $c_{j,x}$ in the join is the same as $c_{x,j}$ in the fork, then that schedule is the same as B^r. Therefore, RCP^* derives the optimum for a join.

Algorithm RCP uses $L(j) = \tau_j + c_{j,x} + \tau_x$ as priorities. It will result in the same execution order if $\tau_j = \tau_i$ for $i \neq j$, i.e. leaf tasks have the same task execution times. ■

References

[1] T. Adam, K.M. Chandy and J. R. Dickson, A comparison of list schedules for parallel processing Systems, *CACM*, 17:12 (1974), 685-690.

[2] F. Berman, Experience with an automatic solution to the mapping problem. In L. Jamieson, D. Gannon an R. Douglass, Eds., *The Characteristics of Parallel Algorithms* (MIT press, 1987) 307-334.

[3] S. Bokhari, On the mapping problem, *IEEE Trans. Comput.* C-30 3(1981), 207-214.

[4] E.G. Coffman and R.L. Graham, Optimal scheduling for two-processors systems, *Acta Informatica*, 3 (1972), 200-213.

We will show that both algorithms minimize FC for every processor and therefore find the optimum for this fork. We consider a cluster u for the following two cases.

Case 1: Root n_x is in u. Without loss of generality, we assume that other tasks in processor u are $n_1, n_2, ..., n_k$. Then the finishing time $FC(u)$ of this processor is always: $FC(u) = \tau_x + \sum_{1 \leq j \leq k} \tau_j$ which is the minimum.

Case 2: When n_x is not in processor u, assume that tasks assigned to u are $n_1, n_2, ..., n_k$. We prove the theorem in two parts. In part 1 we show that the length of the schedule derived by RCP or RCP^* (called schedule A) is the same as the one that executes tasks in increasing order of their ready times (called schedule B). In part 2, we show that schedule B is optimum.

Part 1 of case 2: We examine schedule A in u. When tasks become free after the execution of n_x, the leaf tasks in u may not be executed strictly in an increasing order of the task ready time. At time t, a ready task n_j with the highest $L(j)$ or $L^*(j)$ is executed and another ready task with earlier ready time could still remain in the ready list. Let us look at the sequence of task execution in processor u of schedule A. There could be some processor idle gaps in this schedule for tasks waiting for messages from n_x. Suppose tasks in this sequence are separated by idle gaps into groups $G_1, G_2, \cdots, G_s$. The ready time of tasks in any group G_j where $2 \leq j \leq s$ should be greater than that of tasks in group G_i for $1 \leq i \leq j-1$. For tasks within the same group G_j, we can always swap tasks in that group such that tasks are executed in an increasing order of their ready times, but the length of total execution time remains the same. Thus we can transform schedule A to schedule B by performing such a swapping for every group and the length of schedule A is equal to that of schedule B.

Part 2 of case 2: We now prove that schedule B is optimal by induction. Without loss of generality, we can assume that task execution order in processor u of schedule B is $n_1, n_2, \cdots, n_k$, i.e., $RT(n_1) \leq RT(n_2) \leq \cdots RT(n_k)$. Let $FC_k(u)$ be $FC(u)$ by schedule B when processor u has assigned k tasks. When $k = 1$, it is trivial. Assume $FC_k(u)$ is minimum for $k = s$.

When $k = s + 1$, task n_{s+1} has the largest ready time and is scheduled as the last task. Then from the induction hypothesis, the execution time $FC_t(u)$ for $n_1, n_2, \cdots, n_s$ is minimum. The finishing time of processor u with task n_{s+1} is $FC_{s+1}(u) = \max(FC_s(u), RT(s+1)) + \tau_{s+1}$.

Suppose there is another schedule B' having less parallel time $FC'_{s+1}(u)$ when executing these $s+1$ tasks. We examine the execution order of tasks in schedule B'. Notice that the ready times for tasks assigned to processor u are fixed: $RT(j) = \tau_x + c_{x,j}$ for $1 \leq j \leq s+1$.

If n_{s+1} is not executed as the last task in B', since it has the largest ready time, all tasks executed after that task have less or equal ready times. Thus there is no idle gap between them. Then we can always swap those tasks and make task n_{s+1} be the last task while the parallel time remains the same. We call the new schedule by such swapping B''. Thus the length of B'' is equal to that of B', and $FC''_{s+1}(u) = FC'_{s+1}(u) < FC_{s+1}(u)$.

	n=450 bsize=5	n=450 bsize=10	n=1000 bsize=10
p=2	1.97	1.9	1.99
p=4	3.8	3.7	3.9
p=8	7.3	6.9	7.8
p=16	12.8	11.9	14.4
p=32	19.0	12.9	25.7

Table 7: The speedup of PYRROS code over the sequential GE program on nCUBE-2.

7 Conclusions

If the task graph weight information is not utilized for scheduling, the case of $FRND$, the performance could be more than 50% worse than RCP^*.

For coarse grain DAGs the difference in performance of heuristics that utilize *accurate* critical path information is insignificant provided that a good processor assignment is given.

For fine grain DAGs: (a) Ready list scheduling is superior to free list scheduling in general. (b) The variation of the performance for heuristics that use ready list scheduling and critical path information is within 15% on average. This result is similar to Adam, Chandy and Dickson [1] where they have observed a 10% variation between CP and random priority list when communication is zero.

Finally, a simple ready list scheduling heuristic with low complexity such as RCP^* will work extremely well for the task ordering problem. We have used this algorithm in the PYRROS system.

Other papers on this work can be obtained via anonymous ftp from *cs.rutgers.edu* under the directory *~ftp/pub/gerasoulis*.

Acknowledgments

We thank Alain Darte for several useful suggestions and Weining Wang for programming the random graph generator. We also thank the referees for their useful comments.

A Proof of Theorem 4

For the fork in Figure 8(a), let the finishing time for processor u be $FC(u)$, then

$$PT = \max_{1 \le u \le p} FC(u), \qquad FC(u) = \max_{Proc(n_j)=u} CT(j).$$

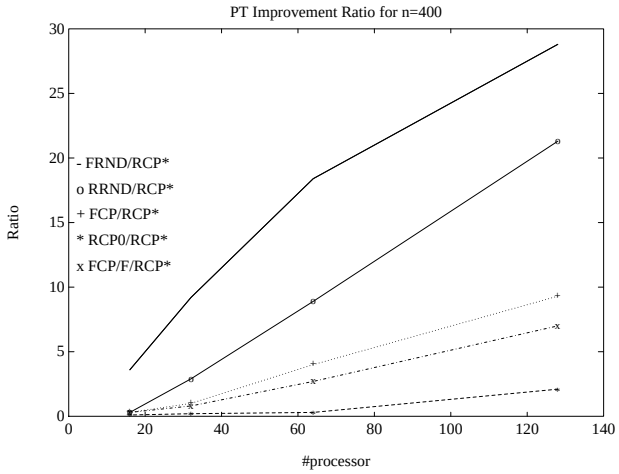


Figure 10: Scheduling Gaussian Elimination DAG with 80000 tasks.

have been used in the literature for GE. One is the level ordering (known as naive) where tasks are ordered for execution level by level. The other is the compute ahead ordering which is similar to level with the exception of the ordering of the pivot element which is given more of a priority Ortega, [21] pp. 241-242. The compute ahead ordering uses the regularity of the GE graph to detect important tasks such as the pivoting tasks, which is not a general method for an arbitrary graph.

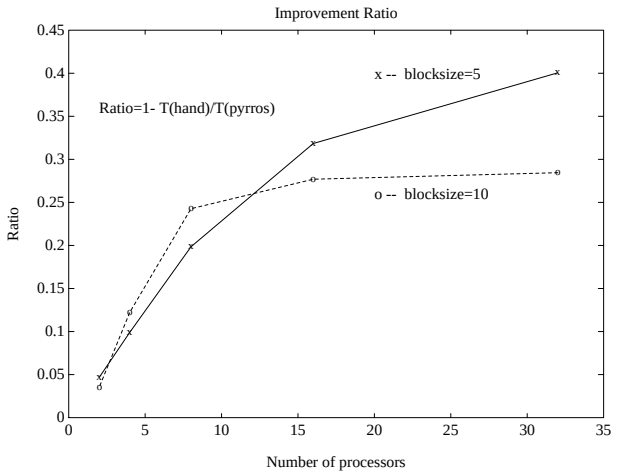


Figure 11: PYRROS/RCP vs. a hand-written program with level ordering on nCUBE-2.

In Figure 11 we compare the RCP ordering with the naive ordering for a matrix size of $n = 450$ using block-submatrix data partitioning. Blocksize 5 and 10 are the sizes of the submatrix partitioning. There is a performance improvement from 5% to 40% depending on the block size and the number of processors. Table 7 shows the speedup achieved by the complete scheduling and code generation system PYRROS for the GE example on the nCUBE-2 over a hand written sequential program.

- A3: By increasing communication the ready tasks are scattered in a much larger time period. Thus the number of ready tasks at each step decreases.

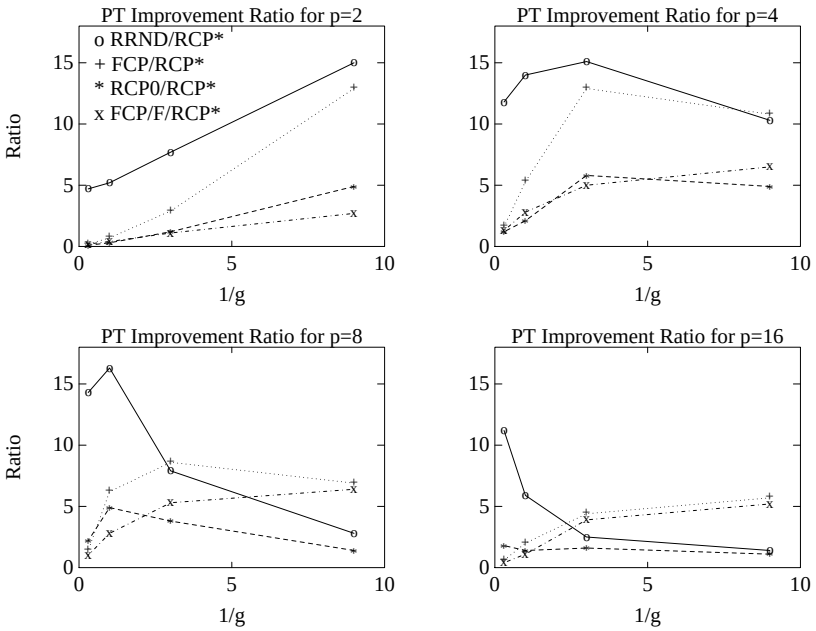


Figure 9: The improvement ratio when the number of processors varies.

We now explain the results in Figure 9. For $p = 2$ the performance is the same as the one observed previously in Tables 4, 5, 6. As the graphs become finer grained the improvement of RCP^* increases. For the case $1/g \approx 0.3$ the improvement is not as great because the effect of A2 comes into play.

For $p = 4$ and $p = 8$, we see that the improvement increases up to a certain point and then it changes direction and starts to decrease. This is because of the influence of A3.

For $p = 16$, the improvement variation is less than 5%. The exception is the $RRND$ heuristic only in the extreme coarse grain case. This is explained by A1 and A3.

The reason that we have chosen small graphs previously is that we wanted to test RCP with an $O(v^2)$ algorithm, that uses task insertion. In Figure 10 we show experiments for a much larger task graph the Gaussian Elimination graph [5] with 80000 tasks and 160000 edges. The granularity is about 1. The improvement is quite large compared to FRND and RRND and close for the other three. This is consistent with experiments for random DAGs.

6.3 Experiments on the nCUBE-2 hypercube

We present an experiment with a real parallel architecture, the nCUBE-2, to demonstrate the practicality of our algorithms using the BLAS-3 Gaussian Elimination algorithm. The standard clustering method is cyclic wrap of the columns of the matrix along the gray code ring of the hypercube [21]. Two orderings

information. For fine grain graphs, however, this is not the case where RCP^* outperforms all others. Again RCP^* uses much more accurate information in the level computation. $RCP0$ and $RRND$ do not include communication weight in their task priorities. As expected from the previous theoretical analysis the performance of RCP^* and RCP is similar, with RCP^* slightly better on average.

Also notice that ready list scheduling heuristics perform better than the free list scheduling. This is because of the idle gap created by communication wait in free list scheduling. Increasing $1/g$ by enlarging the communication weight, free list scheduling will create bigger idle gaps and results in a worse performance. Even the FCP/F which tries to fill the idle gaps, with an additional computational cost, cannot perform better than RCP^* for most of cases.

An interesting question is if the above observed performance behavior is true when we also vary the number of processors. The width of those 100 random graphs varies from 8 to 20, thus we select the number of processors from 2 to 16. In Figure 9 we show the results for $p = 2, 4, 8, 16$ and $1/g \approx 0.3, 1, 3, 9$. The RCP performance is not shown because it is close to that of RCP^*. Also the average improvement ratio of RCP^* over $FRND$ is not shown because of the large variation from 5% to 50% depending on the size of the granularity.

	RCP	$RRND$	$RCP0$	FCP	FCP/F	$FRND$
SG1	0.0%	7.9%	0.7%	0.5%	0.4%	8.2%
SG2	0.1%	11.4%	0.5%	1.1%	0.7%	12.8%
DG1	2.5%	13.7%	2.8%	3.3%	3.0%	15.2%
DG2	0.4%	14.2%	1.3%	1.3%	1.1%	15.5%
Average	0.8%	11.8%	1.2%	1.6%	1.3%	12.9%
#cases RCP^* is better, same, worse	35%, 60%, 5%	48%, 50%, 2%	100%, 0%, 0%	90 %, 10%, 0%	82%, 13%, 5%	100%, 0%, 0%

Table 6: The average improvement of RCP^* over other heuristics. $p = 4$ and coarser grain $1/g \approx 0.3$.

Notice that the RCP^* is always better than the other heuristics. However, the improvement ratio does not always increase as graphs become finer grained (i.e. $1/g$ increases). There are many reasons for this phenomenon. The performance is affected not only by the granularity but also by the degree of the parallelism and the average number of free tasks or ready tasks available for selection at each scheduling step.

The following facts can make the performance of the heuristics similar.

- A1: By increasing the number of processors the number of ready (or free) tasks in the priority list is reduced at each scheduling step. Thus the selection choice is limited.
- A2: By decreasing the number of processors the number of ready tasks increases at each step and any heuristic has enough tasks to schedule.

3. DG1: DAGs where $70 < v < 210, 400 < e < 1400$. Average, $v = 146$, $e = 810$.

4. DG2: DAGs where $230 < v < 500, 1000 < e < 5000$. Average, $v = 354$, $e = 2620$.

Given a DAG, we first fix the processor assignment using PYRROS algorithms. Then we run the above seven algorithms under the same processor assignment. We will use the improvement ratio to compare RCP^* with another algorithm, say, A:

$$Improvement = 1 - \frac{PT_{RCP^*}}{PT_A}.$$

We first report the results for $p = 4$ and $1/g \approx 1$. Afterwards we keep the same number of processors and vary the granularity. Finally we vary both the granularity and the number of processors.

	RCP	$RRND$	$RCP0$	FCP	FCP/F	$FRND$
SG1	0.1%	11.4%	2.2%	4.4%	2.0%	19.6%
SG2	0.3%	13.8%	1.0%	5.0%	2.2%	20.6%
DG1	1.7%	15.4%	3.5%	6.0%	4.2%	24.5%
DG2	0.5%	15.3%	1.9%	5.6%	3.0%	23.9%
Average	0.7%	14.0%	2.1%	5.3%	2.8%	22.2%
#cases RCP^*						
is better	33%,	100%,	62%,	100%,	95%,	100,%
same	62%,	0%,	35%,	0%,	5%,	0%,
worse	5%	0%	3%	0%	0%	0%

Table 4: The average improvement of RCP^* over other heuristics. $p = 4$ and $1/g \approx 1$.

Table 4 shows the average improvement of RCP^* over other algorithms when the number of processors $p = 4$ and $1/g \approx 1$. In Table 5 we reduce the granularity by a factor of 3 so that $1/g \approx 3$, and in Table 6 we increase the granularity by a factor of 3 for every DAG so that $1/g \approx 0.3$, by increasing and reducing the communication weights by a factor of 3.

	RCP	$RRND$	$RCP0$	FCP	FCP/F	$FRND$
SG1	-0.1%	15.5%	8.0%	14.2%	6.1%	34.8%
SG2	0.3%	15.9%	1.5%	12.1%	4.3%	39.4%
DG1	0.1%	11.5%	8.1%	11.6%	4.6%	36.7%
DG2	0.0%	17.6%	5.7%	13.5%	4.7%	41.7%
Average	0.1%	15.1%	5.8%	12.9%	5.0%	38.2%
#cases RCP^*						
is better	35%,	100%,	84%,	100 %,	97%,	100%
same	50%,	0%,	13%,	0%,	3%,	0%,
worse	15%	0%	3%	0%	0%	0%

Table 5: The average improvement of RCP^* over other heuristics. $p = 4$ and finer grain $1/g \approx 3$.

From the above tables we see that for coarse grain DAGs the performance of heuristics based on accurate critical path information is close. The exception are FRND and RRND which do not use accurate

6.2 Experiments with random graphs

In this section we perform an experimental comparison of the seven heuristic algorithms:

1. RCP^* – ready task list scheduling with highest modified level first.
2. RCP – ready task list scheduling with highest level first.
3. $RCP0$ – it is the same as RCP except assuming edge communication cost as 0 in the level computation.
4. $RRND$ – ready task list scheduling with random numbers as task priorities.
5. $FRND$ – free task list scheduling with random numbers as task priorities, i.e. an algorithm that does not use any heuristic and only produces a valid schedule which satisfies the precedence constraints among tasks.
6. FCP – free task list scheduling with highest level first. It selects a highest level free task at each step. Since the selected task may not be ready at that time, an idle gap may be left.
7. FCP/F – it is the same as FCP but it will check whether there is a gap left. If there is, it tries to find some tasks to fill that gap. The algorithm is time consuming and its complexity exceeds $O(v^2)$. This is the key idea used in the insertion heuristic algorithm ISH by Kruatrachue and Lewis [18].

We have used a program to generate random DAGs. The parameters that can be varied are the number of tasks v, the probability of having a precedence edge between two nodes, the task execution time and communication weights. The program first generates a number of task nodes within the given bounds randomly, and then generates edges between them randomly. Finally it assigns random numbers to node and edge weights.

To have a complete study of performance we vary the DAG parameters that may affect the performance of an algorithm. These are the size of the DAG, the sparsity of the edges (if $v < e < 3v$ then this is considered relatively sparse otherwise relatively dense), the ratio between the average communication and the average computation in DS which is an estimate of the reciprocal of the granularity of the DAG, $1/g$ [12], and the number of processors.

We generate the following 4 groups of DAGs with 25 random DAGs in each group and $1/g \approx 1$. SG1 and SG2 are relatively sparse and SG1 and DG1 are relatively small DAGs. The degree of parallelism, also known as the size of maximal independent task set or the width of the DAG, varies from 10 to 20.

1. SG1: DAGs where $70 < v < 210, 100 < e < 400$. Average, $v = 143$, $e = 264$.
2. SG2: DAGs where $230 < v < 500, 350 < e < 800$. Average, $v = 317$, $e = 609$

Proof: The proof uses similar analysis and definitions as in the previous theorem and we only need to show $PT^i_{RCP^*} - PT^i_{RCP} \leq 0$.

Assume that n_y is the task with the highest $L^*(y)$ among ready tasks and it is selected by algorithm RCP^*. Also assume that ready task n_w is selected by RCP and $n_y \neq n_w$. Thus $L^*(w) \leq L^*(y)$ but $L(w) \geq L(y)$ implying $\tau_w \geq \tau_y$.

Consider $PT^i_{RCP^*} - PT^i_{RCP} \leq \max(D, \ \ S_{z,y} - S_{z,w} | n_z \in FLIST(u) - \{n_y, n_w\})$.

For the second term in the above right-hand side, $S_{z,y} - S_{z,w} \leq \max(\tau_y - \tau_w, 0) \leq 0$.

Now consider $D = \max(H_y, S_{w,y}) - \max(H_w, S_{y,w})$. If $D \leq 0$, then $PT^i_{RCP^*} - PT^i_{RCP} \leq \max(0,0) \leq 0$. If $D > 0$, by $L^*(w) \leq L^*(y)$ and several other facts used in the previous theorem we have

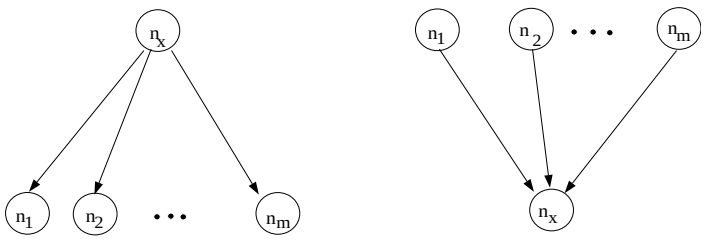
$$D = \tau_y - \tau_w + L(w) - L(y) = L^*(w) - L^*(y) \leq 0.$$

Hence, $PT^i_{RCP^*} - PT^i_{RCP} \leq \max(D, \ S_{z,y} - S_{z,w} | n_z \in FLIST(u) - \{n_y, n_w\}) \leq \max(0,0) = 0$. ■

6 Theoretical and Experimental Comparisons

6.1 Optimality on fork and join DAGs

Finding the optimal ordering for a general case is NP-hard, even for chains of tasks [14]. However, for fork and join DAGs shown in Figure 8, we show that they are solvable in a polynomial time by RCP and RCP^*. The proofs of the following theorems are given in the appendix.



(a) Fork (b) Join

Figure 8: Fork and join structures

Theorem 4 *Given a processor assignment of tasks in a fork DAG, RCP and RCP^* find an optimal schedule.*

Theorem 5 *Given a processor assignment of tasks in a join DAG, RCP^* finds an optimal schedule. RCP finds the optimum when sizes of tasks in a join DAG are equal.*

Assume at time $lc(u)$ of step i, ready task n_x at cluster u is selected for execution on physical processor u. Then the next available time for processor u at step $i+1$ is $lc(u)+\tau_x$. Thus

$$PT^i = \max_{1\le j\le p}\{\max(F(j),W(j))\} = \max(T_u(x), T_{others})$$

where $T_{others} = \max_{j\neq u}\{\max(F(j),W(j))\}$. And

$$T_u(x) = \max(F(u),W(u)) = \max(H_x, \quad S_{z,x}|n_z \in FLIST(u) - \{n_x\})$$

where H_x is the length of longest path that goes through n_x:

$$H_x = \max(lc(u), RT(x)) + L(x) = lc(u) + L(x)$$

and $S_{z,x}$ is the longest path that goes through free tasks other than n_x in processor u:

$$S_{z,x} = \max(lc(u)+\tau_x, RT(z)) + L(z).$$

Now assume that n_y is the task with the highest level among ready tasks selected by RCP and ready task n_w is the optimal choice for processor u that minimizes $PT^i - PT^{i-1}$ over all ready tasks in $RLIST(j)$. Then according to Lemma 1,

$$\begin{aligned} PT^i_{RCP} - PT^i_{lopt} &= \max(T_u(y), T_{others}) - \max(T_u(w), T_{others}) \\ &\le \max(T_u(y) - T_u(w), 0) \quad \le T_u(y) - T_u(w) \\ &\le \max(D, \quad S_{z,y} - S_{z,w}|n_z \in FLIST(u) - \{n_y, n_w\}) \end{aligned}$$

where $D = \max(H_y, S_{w,y}) - \max(H_w, S_{y,w})$.

For task $n_z \in FLIST(u), \quad z \neq w, z \neq y$, using Lemma 1:

$$\begin{aligned} S_{z,y} - S_{z,w} &= \max(lc(u)+\tau_y, RT(z)) + L(z) - \max(lc(u)+\tau_w, RT(z)) - L(z) \\ &\le \max(\tau_y - \tau_w, 0) \quad \le \Delta T. \end{aligned}$$

If $D \le 0$, then $PT^i_{RCP} - PT^i_{lopt} \le \max(0, \Delta T) \le \Delta T$.

If $D > 0$, since $S_{y,w} \ge H_y, \quad S_{w,y} \ge H_w$, then $S_{w,y} \ge S_{y,w} \ge H_y$, $S_{w,y} \ge H_w$.

Thus $L(w) \le L(y)$ and $RT(w) \le lc(u), \quad RT(y) \le lc(u)$ implying:

$$\begin{aligned} D &= S_{w,y} - \max(H_w, S_{y,w}) \quad = S_{w,y} - S_{y,w} \\ &= \max(lc(u)+\tau_y, RT(w)) + L(w) - \max(lc(u)+\tau_w, RT(y)) - L(y) \\ &= \tau_y - \tau_w + L(w) - L(y) \quad \le \Delta T \end{aligned}$$

Hence, $PT^i_{RCP} - PT^i_{lopt} \le \max(D, S_{z,y} - S_{z,w}|n_z \in FLIST(u) - \{n_y, n_w\}) \le \Delta T$. ■

Theorem 3 *The RCP^* heuristic is δ-lopt with $\delta \le \Delta T$.*

which imply that $PT^4 = 6$.

The schedule in Figure 7(c) is based on the priority order $\{n_1, n_2, n_3, n_4, n_6, n_5, n_7\}$. This priority order is derived by using the task level values and assuming that all edges have nonzero communication even when they are within a processor. After the execution of n_1, the tasks n_2 and n_3 are both free and ready. Since n_2 has higher priority than n_3, n_2 is scheduled first. Since n_3 has been preassigned in processor 1, it has to be scheduled after the completion of n_2 even though it is ready and processor 2 is idle. As a result, the ready time of n_6 is delayed. The overall parallel time is 1 time unit longer than that of Figure 7(b). *This example also demonstrates the importance of accurate level information which is difficult to compute if the clusters to processor assignment is not known in advance.*

step i	*gclock*	Selected	$RLIST(1)$	$lc(1)$	$RLIST(2)$	$lc(2)$	PT^i
0			$\{n_1\}$	0	$\{\}$	0	6
1	0	n_1	$\{n_3, n_2\}$	1	$\{\}$	0	6
2	1	n_3	$\{n_2\}$	2	$\{n_6\}$	3	6
3	2	n_2	$\{n_4\}$	3	$\{n_6\}$	3	6
4	3	n_4	$\{n_5\}$	4	$\{n_6\}$	3	6
5	3	n_6	$\{n_5\}$	4	$\{\}$	4	6
6	4	n_5	$\{n_7\}$	5	$\{\}$	4	6
7	5	n_7	$\{\}$	6	$\{\}$	4	6

Table 3: RCP scheduling corresponding to Figure 7(b).

5.1 *δ-lopt* analysis for ready list scheduling

As we mentioned earlier, the problem with FLS is that it creates an idle gap when scheduling a free task on a processor. In this section we only present a *δ-lopt* analysis for a ready list scheduling heuristic. We will compare the performance of FLS and RLS in Section 6.2.

We consider two RLS heuristics: the RCP based on the highest level $L(x)$ as mentioned in the last subsection and the RCP^* based on $L^*(x)$ priorities. Both of these heuristics can be implemented in $O(v \log v + e)$ time. For tie breaking between tasks we can use the MISF (Most Immediate Successor First) principle [16].

The *δ-lopt* definition here is slightly different that one given for list scheduling with zero communication. *The local optimum parallel time PT^i_{lopt} is derived over all possible schedules of ready tasks in $RLIST(j)$ for a selected processor j.* Then the following theorems are true.

Theorem 2 *The RCP heuristic is δ-lopt with $\delta \leq \Delta T$.*

Proof: We need to prove that $PT^i_{RCP} - PT^i_{lopt} \leq \Delta T$.

We give an example for clarification.

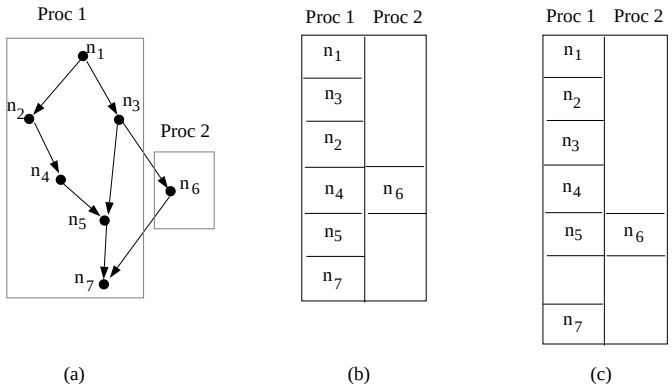


Figure 7: (a) A processor assignment of a DAG with unit computation and communication weights. (b) A schedule by FCP and RCP scheduling (c) Another schedule based on priority order $\{n_1, n_2, n_3, n_4, n_6, n_5, n_7\}$.

Figure 7(a) shows a DAG with unit computation and communication weights. Without a processor assignment given, the levels of tasks n_2 and n_3 are $L(2) = 7$ and $L(3) = 5$. When a processor assignment is given, as shown in Figure 7(a), then the edges within a processor are zeroed and $L(2) = 4$ and $L(3) = 5$. The schedule shown in Figure 7(b) is derived by RCP or FCP. The CP priority order is $\{n_1, n_3, n_2, n_4, n_6, n_5, n_7\}$ and a detailed description of each merging step is shown in Table 2 for FCP and Table 3 for RCP. The only difference between FCP and RCP for this example is the task selection at step 4 derived as follows: At the completion of step 3 in Table 2, $lc(1) = 3$ and $lc(2) = 2$, and the global time is set to $gclock = 2$. Processor 2 is available at time 2 but processor 1 is not since it executes n_2 at that time. Thus n_6 is picked up for scheduling and $lc(2) = 2 + 2 = 4$ where the first 2 is the old clock time and the second 2 is due to both communication and computation delay for this task. For Table 3 step 4 the differences is in $gclock = 3$ because of $lc(1) = 3$ and $lc(2) = 3$ at the completion of step 3. Thus at time 3 both n_4 and n_6 can be chosen. We choose n_4 for scheduling because of the alphabetical tie breaking rule.

step i	$gclock$	Selected	$FLIST(1)$	$lc(1)$	$FLIST(2)$	$lc(2)$	PT^i
0			$\{n_1\}$	0	$\{\}$	0	6
1	0	n_1	$\{n_3, n_2\}$	1	$\{\}$	0	6
2	1	n_3	$\{n_2\}$	2	$\{n_6\}$	2	6
3	2	n_2	$\{n_4\}$	3	$\{n_6\}$	2	6
4	2	n_6	$\{n_4\}$	3	$\{\}$	4	6
5	3	n_4	$\{n_5\}$	4	$\{\}$	4	6
6	4	n_5	$\{n_7\}$	5	$\{\}$	4	6
7	5	n_7	$\{\}$	6	$\{\}$	4	6

Table 2: FCP scheduling corresponding to Figure 7(b).

For the computation of the parallel time at the completion of step 4 of FCP table 2, we have

$$F(1) = \max(lc(1), RT(4)) + L(4) = \max(3,\ 3) + 3 = 6, \qquad F(2) = lc(2) = 4$$

$$W(1) = ST(2) + L(2) = 2 + 4 = 6, \qquad W(2) = ST(6) + L(6) = 6$$

and an idle gap is created in the schedule. On the other hand, if a ready task is chosen for scheduling then this task can start its execution immediately. So in the presence of communication we have two choices of list scheduling:

- Free list scheduling (FLS). Each processor maintains a free list of tasks and the highest priority free task is scheduled at the time that this processor becomes available.
- Ready list scheduling (RLS). Each processor maintains a ready list of tasks instead of free list and the highest priority ready task is scheduled at the time that this processor becomes available.

We call the free list scheduling with the CP priority as FCP (Free Critical Path), the ready list scheduling with the CP priority as RCP (Ready Critical Path). The ready list scheduling step is the same as the free list scheduling step with the exception that we replace *ready* list instead of *free* list in each step above.

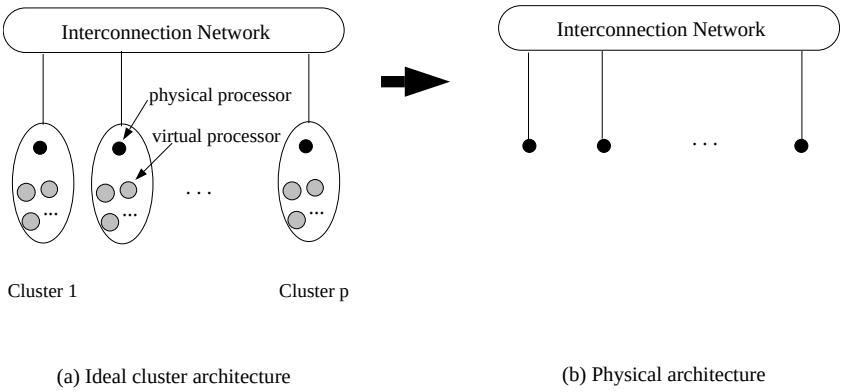


Figure 6: Transformation from a virtual architecture to the physical architecture.

Following the model presented in Section 3.1, we can think of the above list scheduling as a sequence of virtual to physical processor merging operations depicted in Figure 6. The parallel time computation PT^i is similar to the computation described in the previous section. The main difference is that here each physical processor initially owns a set of virtual processors corresponding to the tasks pre-assigned to that physical processor, and it maintains a local clock and a free task list at each step. *Each virtual processor at each step now can start execution of a task only after the time of the local clock for this processor.*

The parallel time can be computed as follows. Assume that before starting step $i+1$, that is at the completion of step i, $FLIST(j)$ and $RLIST(j)$ are the local free and ready task lists of processor j, for $1 \leq j \leq p$. Then

$$PT^i = \max_{1 \leq j \leq p} \{\max(F(j), W(j))\}$$

where

$$F(j) = \begin{cases} lc(j) & FLIST(j) = \emptyset \\ \max_{n_k \in FLIST(j)} \{\max(lc(j), RT(k)) + L(k)\} & otherwise \end{cases}$$

$$W(j) = \begin{cases} ST(x) + L(x) & n_x \text{ is the last task executed before time } lc(j) \text{ in Proc } j \\ 0 & \text{no such task exists.} \end{cases}$$

and INTEL iPSC/860 use wormhole routing, implying that this formula will overestimate the effect of the distance. Dunigan [7] provides a data table for computing transmission delays for such architectures. It should be mentioned, however, that there are other factors that could affect transmission delay such as communication contention. El-Rewini and Lewis [8] address the communication contention in their algorithm. However, it is hard to model such a situation in practice [7]. In PYRROS we perform some optimizations to reduce communication contention in the code generation part of the systems [26] but not in the scheduling part. Preliminary experiments show that this approach works well and an example will be given later for the nCUBE-2.

The purpose of this paper is to study algorithms for the third stage, the task ordering. The ordering algorithm is also important for other scheduling heuristics and systems. For example, HyperTool [23] uses the Bokhari's physical mapping algorithm after a processor assignment is derived. It would produce a better solution if task re-ordering was used based on the new physical mapping information.

5 Task Ordering with Nonzero Communication

In this section, we extend the analysis of Section 3 to the following task ordering problem with nonzero communication: Given p clusters of unordered tasks mapped onto p physical processors, determine a schedule for these clusters. Figure 6 shows p clusters of unordered tasks (virtual processors) and p physical processors. Because the physical distance and inter-cluster communication delay are known in advance of execution, the level information $L(x)$ that includes interprocessor communication cost is deterministic. We can therefore extend the list scheduling to this problem as follows:

- First assign a priority value to each task. For example, the CP priority can be assigned by using the level information $L(x)$ that includes both the task computation and edge communication in the longest path.
- Maintain a local *free* task list and a local clock for each processor. The local clock time $lc(j)$, $j = 1 : p$ is set to the maximum between the earliest time that the processor is available for execution and the earliest time that the local free list becomes nonempty. The global clock time is defined as

$$gclock = \min_{j=1:p} lc(j) \text{ and processor } j \text{ has at least one free task.}$$

- For each scheduling step the processor that is available at time $gclock$ and has the highest priority free task is chosen and that task is scheduled. In the case of a tie in priorities then the processor with the smallest processor number in $1 : p$ is chosen, i.e. in alphabetical order.

When communication delay is zero, a task selected by the classical list scheduling algorithm is ready for execution as soon as it becomes free. In the presence of communication, however, this is no longer true. If a free, but not-ready, task is scheduled it must wait for the data to arrive before it can start its execution

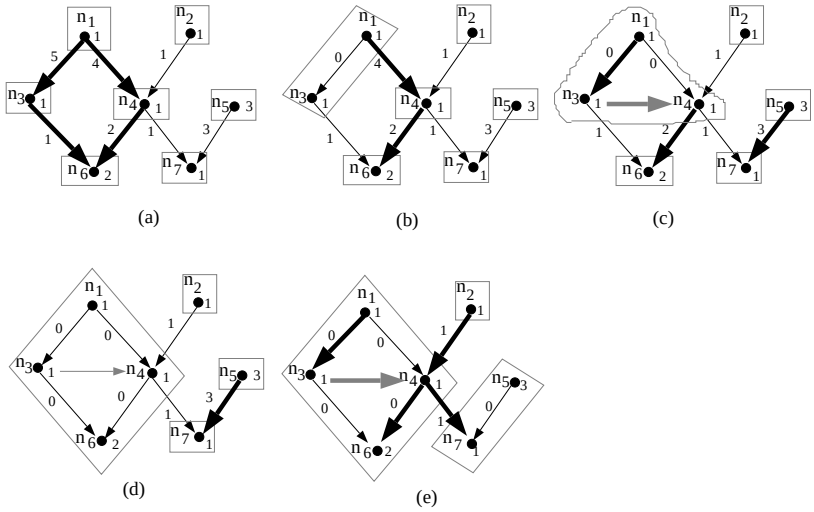


Figure 5: The clustering refinements in DSC.

processors. At each step DSC tries to zero the first unexamined edge in DS to reduce the parallel time. In Figure 5, we show how DSC clusters the example of Figure 2(a). Figure 5(a) is the initial clustering. The DS is shown in thick arrows. There are two dominant sequences in Figure 5(a) with $PT = 10$. In the first step, the first edge (n_1, n_3) in one DS is zeroed as shown in Figure 5(b). The new DS is $< n_1, n_4, n_6 >$ and $PT = 10$. This zeroing is accepted since PT does not increase. In the second step (n_1, n_4) is zeroed and the result is two new DS $< n_1, n_3, n_4, n_6 >$ and $< n_5, n_7 >$ shown in Figure 5(c) with $PT = 7$ and this zeroing is also accepted. In the third step (n_4, n_6) is zeroed as shown in Figure 5(d) and this zeroing is accepted since $PT = 7$ determined by DS $< n_5, n_7 >$. Next (n_5, n_7) is zeroed and the PT is reduced to 5. Finally (n_2, n_4) and (n_4, n_7) cannot be zeroed because zeroing them will increase the parallel time. Thus three clusters are produced.

Notice that in the third step shown in Figure 5(c) an ordering is imposed between n_3 and n_4 and then the parallel time must be computed to get the new DS. One of the key ideas in the DSC algorithm is that it computes the schedule and parallel time incrementally from one-step to the next in $O(\log v)$ time. Thus the total complexity is $O((v + e) \log v)$. If the parallel time were not computed incrementally, then the total cost would be greater than $O(v^2)$ which will not be practical for large graphs.

For the processor assignment stage, PYRROS uses the well known arithmetic load balancing heuristic. A comparison of the load balancing algorithm and more sophisticated algorithms proposed by Sarkar [22] for this stage is given in Gerasoulis and Yang [12]. If the architecture is not completely connected then a physical mapping that takes into account the network topology is needed. Currently PYRROS uses a modification of Bokhari's [3] mapping algorithm.

Once the processor assignment is fixed then the communication weights must be re-adjusted to incorporate the processor distance. A common approximation to edge weights is the linear model $d(\alpha + m\beta)$, where d is the distance between two processors [21], m is the size of the message and α and β are machine parameters known as the start up time and transmission speed. The newest parallel architectures such as nCUBE-2

based on task duplication. They have implemented their algorithms in a software scheduling system named TaskGrapher, which is the part of their PPSE system. PYRROS does not allow task duplication since it is not clear how to achieve a low complexity algorithm with duplication. Duplication also increases the space complexity in executing parallel programs.

We use an example from El-Rewini and Lewis [8], pp. 143, to examine how competitive low complexity PYRROS algorithms are to higher complexity algorithms that use duplication. The MH algorithm of TaskGrapher with a complexity over $O(v^2)$ produces a schedule of length 26 on a hypercube network of 7 processors. PYRROS in an almost linear complexity obtains a scheduling of length 27 on 4 or 7 processors of the hypercube. The scheduling is shown in Figure 4(b) and (c). Only task ordering within each processor is listed and a DS that determines parallel time is shown in thick arrows. In [8] it is assumed that the communication delay between processors is $d \times c$ where d is the processor distance and c is the original edge weight. Thus in Figure 4(b), the delay between task 2 and 10 is 2×1 since processor 4 and 0 have a distance of 2 and so is between task 10 and 15. Another comparison to Kruatrachue and Lewis [18]'s DSH heuristic is given in [25] for a tree DAG. For that example PYRROS gives the same parallel time.

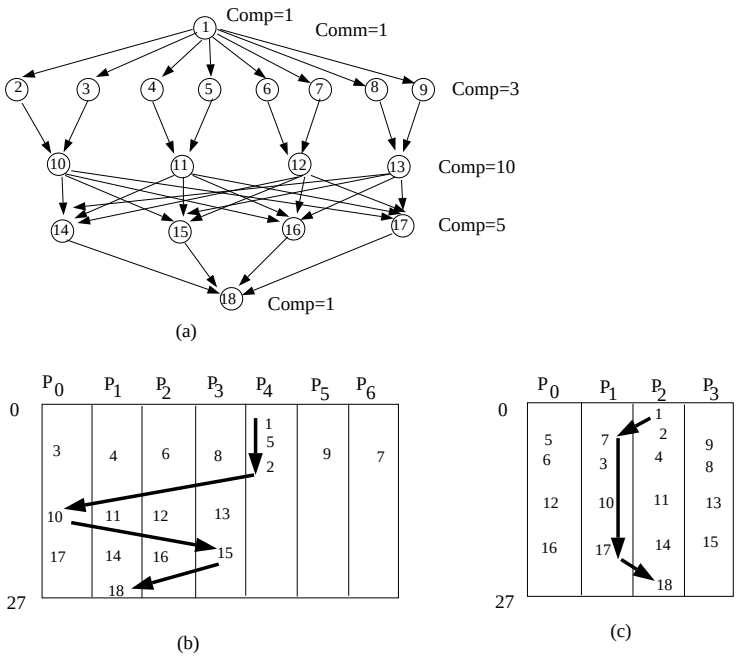


Figure 4: (a) A DAG with all edge weights 1. Tasks in the same layer have the same computation weights. (b) and (c) are the task mapping and ordering results of PYRROS for $p = 7$ and $p = 4$ on a hypercube. They are not Gantt Charts. Thick arrows show DSs.

Because of the inter-dependence between the stages in a multistage method it is critical that each step produces the best possible results in a low complexity. For the clustering stage PYRROS uses the DSC algorithm [25]. Here we only describe the DSC algorithm briefly. A detailed experimental and theoretical comparison of DSC with other existing algorithms is in [11]. DSC has been found to outperform several other algorithms.

The DSC performs a sequence of clustering refinement steps. Initially all tasks are mapped in separate

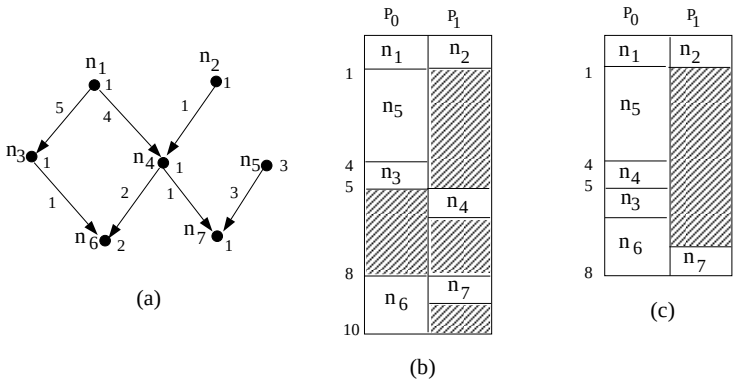


Figure 2: (a) A DAG. (b) The schedule by CP. (c) The schedule by MCP.

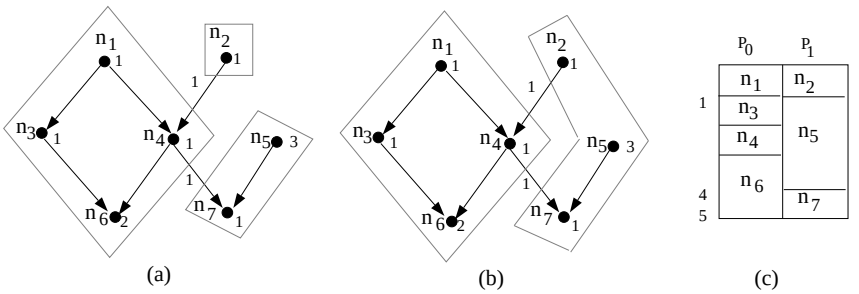


Figure 3: (a) The clustering result. (b) Clusters after merging. (c) The schedule with $PT = 5$.

that allows the start of a task as early as possible. In Figure 2(c) task n_4 is not executed in P_1 because it can start execution earlier in P_0 after n_5. MCP is an improvement over CP at the expense of higher complexity $O(v^2 \log v)$. But MCP also suffers from the non-determinism of the priority computation.

Kim and Browne [17], Sarkar [22], Gerasoulis and Nelken [9] and others in the literature have advocated a multiple stage method to solve the scheduling problem in the presence of communication: 1)Cluster the DAG first. All tasks in a cluster will execute in the same processor. 2) Assign the clusters to physical processors. 3)Order the execution of the tasks in each processor. Kim and Browne [17] have performed an extensive comparison of the one-stage and the multistage method, and found substantial improvement in the performance of the multistage method. An explanation of this performance improvement lies in the fact that more accurate task priorities can be computed from one stage to the next. This multistage approach was recently used by Ngai [20] in runtime resource management of concurrent systems, demonstrating its versatility beyond compile-time scheduling.

In Figure 3, we show how a multistage methods works. In Figure 3(a) a clustering of the DAG from Figure 2(a) is given. Figure 3(b) shows a merging of three clusters down to two clusters (processors). Once the final mapping of tasks to processors has been determined as shown in Figure 3(b) then the communication information becomes deterministic. As a result, a better ordering can be derived. Figure 3(c) shows a schedule that is shorter than that of CP and MCP in Figure 2.

Kruatrachue and Lewis [18] and El-Rewini and Lewis [8] have proposed one-stage scheduling methods

Note that $n_w, n_y \in FLIST$, and $L(y)$ is the largest, so that

$$T_{free}(w) = S_w = clock + \Delta clock_w + L(y), \text{ and } \quad S_y \geq H_w, \quad S_w \geq H_y$$

implying $D = \max(H_y, S_y) - S_w$.

If $D \leq 0$, then $PT^i_{CP} - PT^i_{lopt} \leq \max(0, 0) = 0$.

If $D > 0$, then $S_y \geq H_y$, because $S_w \geq H_y$. Thus

$$D = S_y - S_w = \Delta clock_y - \Delta clock_w + \max_{n_x \in FLIST - \{n_y\}} L(x) - L(y).$$

Since $|\Delta clock_y - \Delta clock_w| \leq \Delta T$ and $L(y)$ is the largest, $PT^i_{CP} - PT^i_{lopt} \leq \Delta T$. ■

The following corollary is a direct result of the last theorem.

Corollary 1 *CP scheduling is locally optimum for DAGs with equal weights.*

Local optimality, of course, does not imply optimality but we expect a near optimum performance of heuristics that possess this property. Reference [1] has provided the experimental evidence for such a performance for a locally optimum heuristic. Furthermore, for a tree DAG with equal weights, CP scheduling is optimal on an arbitrary number of processors [15], and for any DAG with equal weights, CP is optimal on two processors [4].

4 Scheduling with Nonzero Communication

In this section we discuss scheduling in the presence of communication. The scheduling problem can be thought of as consisting of two parts: 1) the task to processor assignment and 2) the task execution ordering within a processor. A list scheduling heuristic solves both problems at once and is known in the literature as a *one-stage* method. While low complexity one-stage methods such as CP perform very well when communication is zero, this is not the case when communication is nonzero. This is because the edge weights are no longer deterministic before the processor assignment has been determined. Communication is zero when two tasks are mapped in the same processor, and nonzero when they are mapped in two different processors. As a result, the task priorities depend on the task assignment, and they cannot be estimated accurately using a one-stage method.

Figure 2 shows a DAG with edge communication weights computed by assuming that each task is mapped in a different processor of a completely connected architecture. The computation weights are also shown next to each task. Figure 2 (b) shows the schedule produced by CP using as priorities the levels that include both computation and communication cost in a path. An improvement of the CP method has been proposed by Wu and Gajski [23] for HyperTool, called MCP. Instead of scheduling the highest priority free task in the first available processor as in CP, the MCP schedules a free task in a processor

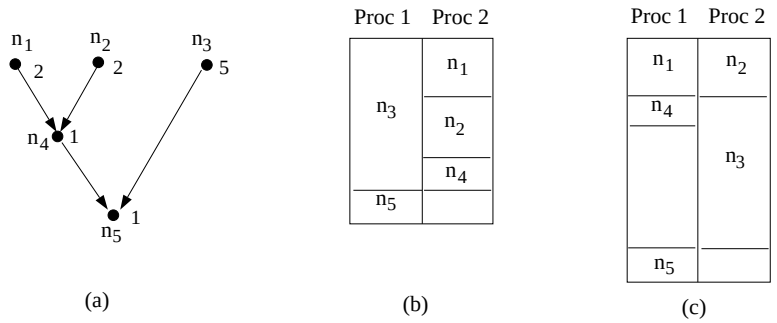


Figure 1: The impact of δ on performance of list scheduling. (a) A DAG. (b) CP results in an optimal schedule with $PT = 6$ and $\delta = 0$ (c) $HLFNET$ results in $PT = 8$ and $\delta = 2$.

since scheduling either n_1 or n_2 instead will result in the same parallel time.

step i	selected	$FLIST$	$WLIST$	$clock$	PT^i	PT^i_{lopt}
0		$\{n_3, n_1, n_2\}$	$\{\}$	0	6	6
1	n_3	$\{n_1, n_2\}$	$\{n_3\}$	0	6	6
2	n_1	$\{n_2\}$	$\{n_3\}$	2	6	6
3	n_2	$\{n_4\}$	$\{n_3\}$	4	6	6
4	n_4	$\{n_5\}$	$\{\}$	5	6	6
5	n_5	$\{\}$	$\{n_5\}$	5	6	6

Table 1: Merging steps corresponding to CP scheduling in Figure 1(b).

The following theorem gives an upper bound for δ for the CP heuristic.

Theorem 1 *The CP heuristic is δ-lopt with $\delta \leq \Delta T = \max_{n_i, n_j \in V} |\tau_i - \tau_j|$.*

Proof: At step i, a task n_y with the highest level among free tasks is selected by CP and the corresponding virtual processor is merged with an idle physical processor, so that

$$PT^i_{CP} = \max(T_{others}, clock + \max_{n_x \in FLIST} L(x)) = \max(T_{others}, T_{free}(y))$$

where

$$T_{free}(y) = \max(clock + L(y), clock + \Delta clock_y + \max_{n_x \in FLIST - \{n_y\}} L(x))$$

$$\Delta clock_y = \text{the next processor available time} - clock.$$

Let task n_w be the optimal choice that results in PT^i_{lopt}. Using Lemma 1,

$$\begin{aligned} PT^i_{CP} - PT^i_{lopt} &= \max(T_{others}, T_{free}(y)) - \max(T_{others}, T_{free}(w)) \\ &\leq \max(0, T_{free}(y) - T_{free}(w)). \end{aligned}$$

Let $D = T_{free}(y) - T_{free}(w) = \max(H_y, S_y) - \max(H_w, S_w)$ where

$$H_z = clock + L(z), \quad S_z = clock + \Delta clock_z + \max_{n_x \in FLIST - \{n_z\}} L(x).$$

- At step i, a task is selected and is scheduled onto a physical processor. The corresponding virtual processor is merged with this physical processor and the total number of virtual processors is decreased by one to $v-i$. The clock is reset to the next earliest time when a physical processor becomes available for the execution of another task and also there is a free task available. PT^i is determined by executing all scheduled tasks on p physical processors and executing unscheduled tasks on $v-i$ virtual processors. *Each virtual processor can only start execution of an unscheduled task after the current global clock time.*

- Finally after step v, the number of virtual processors reaches zero and a schedule for p processors is produced with $PT = PT^v$.

At the completion of step i and before step $i+1$ begins, let *clock* be the current global clock time. Let $FLIST$ be the list of free tasks and $WLIST$ be the list of tasks that are being executed at time *clock*. By being executed we mean that the tasks have started but not completed their execution at time *clock*. Then the parallel time can be derived from the following expression:

$$PT^i = \max(T_{others}, clock + \max_{n_x \in FLIST} L(x))$$

where

$$T_{others} = \begin{cases} \max_{n_j \in WLIST}\{ST(j) + L(j)\} & WLIST \neq \emptyset \\ 0 & otherwise. \end{cases}$$

All heuristic algorithms studied in [1] can be considered in this framework. We assume non-backtracking to avoid the high complexity, then the values of $PT^0, PT^1, \ldots, PT^v$ are monotonically increasing. *A greedy approach would be to have the minimum increase in the parallel time from one step to the next.*

Definition. Let PT^i_{lopt} be the optimum parallel time at step i, which is the solution of $\min\{PT^i - PT^{i-1}\}$ where the minimum is taken over all possible choices of free tasks in $FLIST$ at step i. Then a heuristic is called *δ-lopt* if $\max_i\{PT^i - PT^i_{lopt}\} \leq \delta$ where δ is a given constant. If $\delta = 0$ then the *δ-lopt* heuristic is called a *local optimum.*

The important question is, of course, if local optimality implies near-optimality in the sense of Adam, Chandy and Dickson [1]. While it is difficult to prove such result theoretically, we do know that the CP heuristic is near optimal [1]. Therefore, it would be of interest to see if CP is locally optimum. We expect locally optimum heuristics to perform better on average than heuristics with a large nonzero δ. We present an example to demonstrate this point.

In Figure 1, we show results of two list scheduling algorithms [1]: the CP and HLFNET (highest levels first with no estimated computation times). The priority list for CP is $\{n_3, n_1, n_2, n_4, n_5\}$ since $L(3) = 6 \geq L(1) = 4 \geq \ldots \geq L(5) = 1$. On the other hand the priority list for HLFNET is $\{n_1, n_2, n_4, n_3, n_5\}$ since it assumes that each task weight is equal to unit time.

In Table 1, we show the merging sequences in CP list scheduling. For example, in step 1 n_3 is selected and scheduled in processor 1 because of its high priority. This choice is also the locally optimum choice

Prioritylist = $\{n_1, n_2, ..., n_v\}$ sorted in a descending order of task priorities;
clock = 0;
While Prioritylist is not empty **Do**
    Remove the left-most free task in the list and schedule it to an idle processor if available;
    clock = the next earliest time when a processor becomes available.
ENDWHILE

What is interesting about list scheduling is that the performance is within 50% of the optimum independently of the priority list as shown by Graham [13]. An important question is what choices of priority lists will *consistently* give schedules that are "close" to the optimum schedule. Experimentally, Adam, Chandy and Dickson [1] answered the above question by conducting an extensive empirical performance study of the CP algorithm with other four priority list scheduling algorithms. Their conclusion is that the CP heuristic is superior to others since it is near-optimal (i.e. within 5 percent of the optimal in 90 percent of random cases.) Theoretically, Coffman and Graham [4] show that CP is optimal for scheduling DAGs of equal task sizes on two processors. Also Hu [15] shows that CP is optimal for scheduling tree DAGs of equal task sizes on p processors.

Previous research emphasis has been in uncovering the worst case performance in list scheduling and determining experimentally good heuristics. Experimental results in [1], however, raise an interesting question. Are there any underlying properties that characterize the near optimality of certain list scheduling algorithms such as CP? Furthermore the proof of optimality of CP for equal weight DAGs raises another interesting question. Why does CP perform so well when the tasks sizes are equal? We will investigate these two questions next.

3.1 List scheduling performance characteristics

To identify the important properties of list scheduling heuristics with good performance we need to introduce a mathematical framework. We consider list scheduling as a sequence of *merging* operations controlled by a *clock*. Initially every task is mapped into v virtual processors and there are p physical processors. At each step a virtual processor is merged with a physical processor and the schedule is derived for this step. In the final step, the virtual architecture (DAG) is converted into a physical architecture of p processors. Assuming that PT^i is the parallel time of the new schedule at the completion of merging step i, then the merging sequence produces a sequence of parallel times $PT^0, PT^1, \ldots, PT^v$. The parallel time at each step is determined as follows:

- Initially, there are v virtual processors and p idle physical processors after the last scheduled task in that processor. The parallel time for executing the DAG is the length of the critical path, i.e. $PT^0 = CPL$.

- $Proc(x)$ is the processor assignment for task n_x.
- $ST(x)$ is the starting time of task n_x.
- $RT(x)$ is the ready time of task n_x. $RT(x) \leq ST(x)$.
- $CT(x) = ST(x) + \tau_x$. The completion time of task n_x.
- DS stands for the Dominant Sequence which is the longest path in the scheduled graph.
- $PT = \max_{n_x \in V} CT(x)$. The parallel time which is the length of DS.
- PT_A is the parallel time for scheduling algorithm A. PT_{opt} is the length of the optimal schedule.

We will need the results of the following Lemma.

Lemma 1

$$a - \max(c, d) \leq a - d$$

$$\max(a, b) - \max(c, d) \leq \max(a - c, b - d)$$

$$\max(a_1, ..., a_n) - \max(b_1, ..., b_n) \leq \max(a_1 - b_1, ..., a_n - b_n).$$

Proof: We have that

$$a - \max(c, d) = \begin{cases} a - d & \text{when } c \leq d \\ a - c \leq a - d & \text{when } c > d. \end{cases}$$

Using this inequality, we obtain

$$\max(a, b) - \max(c, d) = \max(a - \max(c, d), b - \max(c, d)) \leq \max(a - c, b - d).$$

It can be generalized further by induction as follows:

$$\max(a_1, ..., a_n) - \max(b_1, ..., b_n) \leq max(a_1 - b_1, \max(a_2, ..., a_n) - \max(b_2, ..., b_n))$$

$$\leq \max(a_1 - b_1, ..., a_n - b_n).\blacksquare$$

3 List Scheduling with Zero Communication

In priority list scheduling, initially a list of tasks is ordered according to some given priority. At each scheduling step, the list scheduling heuristic schedules the highest priority free task on the first available processor. A generic procedure of list scheduling is described below.

heuristics the RCP and RCP^*. Free list scheduling is not used since it creates idle gaps. In Section 6, we present two optimality results for RCP and RCP^*. We also present an experimental comparison for various choices. We have found that RCP^* has the best performance among those heuristics tested which is in agreement with our theoretical results.

2 Preliminaries

A directed acyclic task graph (DAG) is 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 becomes zero if both nodes are mapped to the same processor. The value $\tau_i \in \mathcal{T}$ is the execution time of node $n_i \in V$.

Throughout this paper we make the following assumptions:

- *Duplication* of the same task in separate processors is *not allowed*. Duplication increases the space complexity since both data and programs must be duplicated.
- The *task model* is the compile time *macro-dataflow* model. A task receives all input before starting execution in parallel, executes to completion without interruption, and immediately sends the output to all successor tasks in parallel, see Wu and Gajski [23], Sarkar [22].

The following definitions are used in this paper:

- p is the number of physical processors available for scheduling a given DAG.
- $L(x)$ is the level of a task n_x, which is the length of the longest path from n_x to an exit node, including nonzero communication delays in the path. The recursive definition is

$$L(x) = \tau_x + \max_{n_j \in SUCC(n_x)} \{c_{x,j} + L(j)\}$$

  where $SUCC(n_x)$ is the set of all immediate successors of n_x.
- $L^*(x) = L(x) - \tau_x$. The modified level excluding its execution time.
- $CPL = \max_{n_x \in V} L(x)$. The length of the critical path of the given DAG.
- $\Delta T = \max_{n_j, n_i \in V} |\tau_j - \tau_i|$. The maximum difference between the sizes of all tasks in the DAG.

A *scheduling* of a DAG is a mapping of the task nodes onto the processor set, and a starting time mapping of each task node to the set of real numbers. Given a scheduling then we define the following:

communication is that it can be used on message passing MIMD architectures. For such architectures the communication is much higher than shared memory architectures. As a result, much more sophisticated algorithms are needed to take into account high communication cost and to make sure that communication overlaps with computation.

Automatic scheduling is very useful in developing software tools that generate code for parallel architectures. PREP-P by Berman [2] is a software system that automatically maps undirected communication graphs onto CHiP machine architectures. The OREGAMI/LaRCS software tool by Lo et al. [19] is used for the mapping of directed task graphs onto processors. We have also built an automatic static scheduling and code generation system called PYRROS [26] for parallel architectures. PYRROS is targeted at scheduling weighted *directed acyclic* task graphs, which takes task precedence into account and uses the minimization of parallel time as a cost function. Precedence is not considered in PREP-P and OREGAMI/LaRCS.

Two other systems that are similar to PYRROS and take precedence into account are the HyperTool by Wu and Gajski [23] and TaskGrapher by El-Rewini and Lewis [8]. PYRROS has emphasized the development of algorithms with complexity of almost linear to the size of the graph. This is not the case with other systems which have complexity at least $O(v^2)$, where v is the number of tasks in the DAG. Notice that for most program dependence graphs, the number of edges e is usually in the same order of v. Low complexity algorithms are necessary for a practical system. Consider for example the medium grain Gaussian Elimination (GE) DAG [5]. For a matrix dimension n, $v = n^2/2$ and $e = n^2$, implying that an $O(v^2)$ algorithm will have $O(n^4)$ complexity which is higher than a sequential GE algorithm.

PYRROS scheduling algorithms use a multistage method [17, 21, 22, 24] that includes clustering and task ordering. In this paper, we will give a brief discussion of the PYRROS approach and analyze in detail the task ordering problem. The task ordering problem is to determine the execution order of tasks within each processor after a processor assignment is fixed so that task precedences are satisfied and the parallel time is minimized. The task ordering problem is also important for other scheduling heuristics and systems. For example, task ordering is useful for HyperTool [23] after its physical mapping stage.

This task ordering problem is NP-hard in general [14]. Therefore, the main question is how to devise heuristics with good performance and low computational complexity. Since this problem is closely related to the classical list scheduling, we will first analyze the characteristics of classical list scheduling algorithms and then use this analysis to propose low complexity heuristics for ordering.

In Section 2, we give some preliminary definitions. In Section 3, we introduce a framework that considers list scheduling with zero communication as a series of processor mergings and study the theoretical properties of classical list scheduling algorithms. In Section 4, we briefly discuss PYRROS's multistage algorithms and use examples to compare with other approaches. In Section 5, we address the task execution ordering problem. Here we have two choices of list scheduling. One is based on scheduling tasks from the free list and the other scheduling tasks from the ready[1] list. We introduce two such choices of ready list scheduling

[1] A task becomes *free* if all of its predecessors have finished their execution. A task becomes *ready* if it is free and all of the data that it needs from its predecessors have arrived locally to the processor that it has been assigned.

List Scheduling with and without Communication Delays*

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

January 1992, Revised August 1992

Abstract

Empirical results have shown that the classical critical path (CP) list scheduling heuristic for task graphs is a fast and practical heuristic when communication cost is zero. In the first part of this paper we study the theoretical properties of the CP heuristic that lead to near optimum performance in practice. In the second part we extend the CP analysis to the problem of ordering the task execution when the processor assignment is given and communication cost is nonzero. We propose two new list scheduling heuristics, the RCP and RCP^* that use critical path information and ready list priority scheduling. We show that the performance properties for RCP and RCP^*, when communication is nonzero, are similar to CP when communication is zero. Finally, we present an extensive experimental study and optimality analysis of the heuristics which verifies our theoretical results.

1 Introduction

The processor scheduling problem is of considerable importance in parallel processing. Given a task graph representation of the parallelism, determining a schedule that minimizes the parallel time is NP-complete in general. At an earlier stage of research the scheduling problem was simplified by assuming that communication cost is zero. Even under this simplification the problem remains NP-complete, but there is an interesting upper bound in performance. Any list scheduling heuristic is within 50% of the optimum, Graham [13]. List scheduling has actually an even better average performance. Adam, Chandy and Dickson [1] demonstrated experimentally that the critical path (CP) list scheduling heuristic is within 5% of the optimum in 90% of the times. In this paper we study the properties of list scheduling that provide an explanation to the near optimality of CP.

In the presence of communication the scheduling problem is much harder. List scheduling no longer provides the 50% performance guarantee. Only recently considerable attention has been paid to this problem, e.g. Sarkar [22], El-Rewini and Lewis [8], Kim and Browne [17], Wu and Gajski [23], Cosnard et al. [5], Darte [6] and many others. One of the reasons for the renewed emphasis on scheduling with

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