Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler *

Shankar Ramaswamy and Prithviraj Banerjee

Center for Reliable and High Performance Computing
University of Illinois at Urbana-Champaign
Urbana, IL 61801
E-mail : {shankar,banerjee}@crhc.uiuc.edu

Abstract : *Functional or Control parallelism is an effective way to increase speedups in Multicomputers. Programs for these machines are represented by Macro Dataflow Graphs (MDGs) for the purpose of functional parallelism analysis and exploitation. Algorithms for allocation and scheduling of MDGs have been discussed along with some analysis of their optimality. These algorithms attempt to minimize the execution time of any given MDG through exploitation of functional parallelism. Our preliminary results show their effectiveness over naive algorithms.*
Keywords : *Macro Dataflow Graphs, Distributed Memory Multicomputers, Allocation and Scheduling, Parallelizing Compilers, Optimization.*

1 Introduction

Distributed Memory Multicomputers offer significant advantages over shared memory multiprocessors in terms of cost and scalability. Unfortunately, writing efficient software for them is an extremely laborious process for users. The PARADIGM compiler project at Illinois is aimed at devising a parallelizing compiler for distributed memory multicomputers that will accept sequential FORTRAN 77 programs as input. The completed PARADIGM compiler will automatically:

- Generate data partitioning specifications [1].
- Partition computations and generate communication for data parallel programs [2].
- Exploit high level communication primitives [3].
- Exploit functional parallelism [4, 5].

This paper deals with the last aspect of the PARADIGM compiler, viz., Functional parallelism exploitation. Functional or Inter loop parallelism has long been recognized as a technique to further increase speedups beyond Data or Intra loop parallelism. The two techniques are orthogonal to each other in that the application of one does not preclude the use of the other. In the context of functional parallelism, the notion of Macro Dataflow Graphs (MDG) has been proposed and used by researchers, for example, by Prasanna and Agarwal in [6]. They are a coarse grain representation of programs; nodes are typically loops or nested loops, and edges define precedence constraints among these loops. The weight of a node is its computation cost and the weight of an edges is the cost of communicating data between the nodes involved. Exploiting functional parallelism involves defining the degree of data parallelism for each node in the MDG and scheduling the execution of the nodes in a manner which allows the program to be executed in the shortest time.

Any compiler that wants to exploit functional parallelism therefore, has to:

- Identify and construct the MDG of the given program automatically. Girkar and Polychronopoulos have dealt with this aspect in [7].
- Allocate processors to each node and schedule the MDG in such a way as to minimize its execution time.

Our focus has been entirely on the second step in compilation outlined above. Obtaining an optimal solution to this step has been shown to be NP-complete by Lenstra and Kan in [8]. The two major approaches to the approximate solution of this problem have been; first, a bottom up approach like those used by Sarkar in [9], and Gerasoulis and Yang in [10, 11] and, second, a top down approach like the ones used by Prasanna and Agarwal in [6], Belkhale and Banerjee in [4, 5] and in this paper. A bottom up approach considers the MDG to be made up of lightweight nodes and coalesces these nodes together to form larger nodes during its construction of a schedule. Top down approaches start with the assumption of heavyweight nodes in the MDG and break them down during the process of constructing an optimal schedule. Top down methods take a more global view of the problem than the bottom up approaches, thus performing better.

We first look at the computation and communication cost models used in our work. A few definitions and assumptions are then stated, followed by a discussion of the scheduling algorithm and its optimality. Next we describe our allocation algorithms and present some preliminary results obtained using our algorithms. Finally, we discuss the the implications of our work and the future directions we will be taking.

*This research was supported in part by the Office of Naval Research under Contract N00014-91J-1096, and in part by the National Aeronautics and Space Administration under Contract NASA NAG 1-613.

2 Cost Models

2.1 Computation Model

For the computation model, we use Amdahl's laws, i.e., the execution time of a node (T_i) as a function of p_i is given by:

$$T_i(p_i) = (\alpha_i + \frac{1-\alpha_i}{p_i}) \cdot \tau_i \quad (1)$$

where τ_i is the execution time of the node on a single processor and α_i is the fraction of the node that has to be executed serially.

2.2 Communication Cost Model

A many to many communication of L_{ij} bytes involving p_i sending and p_j receiving processors is assumed to consist of $p_i \cdot p_j$ messages of $\frac{L_{ij}}{p_i \cdot p_j}$ bytes each, i.e. each of the p_i sending processors sends a message to each of the p_j receiving processors, all messages being of equal size. There are three cost components to this transfer, t_1 representing the startup and processing costs for p_j messages at the sending processors, t_2 representing the network cost for a message, and, t_3 representing the startup and processing costs for p_i messages at the receiving processors. We can write expressions for these components as:

$$t_1 = p_j \cdot (t_s + t_p \cdot \frac{L_{ij}}{p_i \cdot p_j}) \quad (2)$$

$$t_2 = t_n \cdot \frac{L_{ij}}{p_i \cdot p_j} \quad (3)$$

$$t_3 = p_i \cdot (t_s + t_p \cdot \frac{L_{ij}}{p_i \cdot p_j}) \quad (4)$$

where t_s denotes the startup cost required for a message at a processor, t_p denotes the processing cost per message byte at a processor and t_n denotes the network cost per message byte.

The total communication cost C_{ij} is a sum of these components and can be written down as:

$$C_{ij} = (p_i + p_j) \cdot t_s + L_{ij} \cdot (\frac{1}{p_i} + \frac{1}{p_j}) \cdot t_p + L_{ij} \cdot \frac{1}{p_i \cdot p_j} \cdot t_n \quad (5)$$

3 Assumptions

Listed below are a few assumptions made for our algorithms and their analysis.

Assumption 1 *MDG specification*

The ith node in the MDG is specified by a pair (α_i, τ_i) and the edge joining nodes i and j is specified by the length of the message transferred L_{ij}. Gupta and Banerjee have shown the estimation of computation and communication costs to be feasible in [1]. Our assumption therefore says that these estimated costs will fit the cost models we outlined earlier.

Assumption 2 *Communication cost parameters*
The values of t_s, t_p, and t_n which are required to compute communication costs can be determined for a particular architecture and are, therefore, available to our algorithms. This is a realistic assumption and has been made by numerous researchers in some form or the other.

Assumption 3 *Computation to Communication Ratio*
For the theoretical analysis that follows, we assume that the weight of any node will be at least an order of magnitude higher than the weight of any edge. This means we can neglect communication costs for the analysis.

Assumption 4 *Control Dependence*
Like many other researchers, we have considered only data dependences between nodes in the MDG. Control dependences are assumed to exist only within nodes and not across them. This is a reasonable assumption considering the nodes in the MDG represent sufficiently large computation.

4 Definitions

We present below the definitions of two terms that are used in the proofs of the theorems that follow in Section 5.

Definition 1 *Critical Path Weight*

The critical path is defined to be the path in the MDG whose nodes have the largest aggregate weight when at most k processors can be used by any node. This weight is called the critical path weight and is denoted by C_k.

Definition 2 *Average Finish Time*
The average finish time (A_k) of an MDG with n nodes, when each node can use at most k processors is given by:

$$A_k = \frac{1}{p} \cdot \sum_{i=1}^{n} T_i(p_i) \cdot p_i \quad (6)$$

where p is the maximum number of processors available at any given time in the given multicomputer system, p_i is the number of processors used by the ith node ($1 \leq p_i \leq k$), and $T_i(p_i)$ is the execution time of the node on p_i processors.

If the computation cost model is used in the expression above, we have:

$$A_k = \frac{1}{p} \cdot \sum_{i=1}^{n} \tau_i \cdot (1 + \alpha_i \cdot (p_i - 1)) \quad (7)$$

5 Scheduling Algorithm

5.1 PSA

Our allocation heuristics use a scheduling algorithm called the Prioritized Scheduling Algorithm (PSA) whose steps are outlined below:

Step 1 Do a Topological Sort of the MDG with the directions of all edges reversed (all other parameters remaining the same). The ordering produced decides the priority of each node.

Step 2 The next node to be processed is the one with the highest priority and whose precedence constraints have been met at the current time. Schedule the selected node at a time when its processor requirement is met. Repeat this step until no node is left unexecuted.

Step 3 The earliest time at which all the nodes finish executing is the execution time of the MDG.

Similar scheduling algorithms are described for example, by Liu in [12], by Garey, Graham and Johnson in [13], by Wang and Cheng in [14], and by Turek, Wolf and Yu in [15]. While these researchers have published results similar to Theorem 1 below, the assumptions used by them have been limiting. Our result is more general in its assumptions and implications. In fact, as shown in Corollaries 2 and 3, our result matches those obtained by earlier researchers for the specific assumptions they use.

5.2 Optimality of PSA

Theorem 1 *If each task uses at most x processors in a p processor system, the finish time obtained using the PSA (T_{psa}) is bounded by a factor of the optimum finish time (T_{opt}) as given by the following equation:*

$$T_{psa} \leq 2 + \frac{x-2}{p-x+1} * T_{opt} \tag{8}$$

under the assumptions made in Section 3.

Proof If each task uses at most x processors, then a maximum of $x-1$ processors will be forced to be idle, waiting for the xth processor to be free to start the next waiting task. This should mean at least $p-(x-1)$ or $p-x+1$ processors are busy at all times, if there is enough work for them. There not being enough work for $p-x+1$ processors means all other unexecuted tasks are dependent on one or more of the currently executing tasks. There are two cases in which this could happen:

Case 1 Due to poor scheduling by the PSA in spite of there being enough work (The optimal algorithm is able to keep all processors busy). By poor scheduling we mean the PSA has made some wrong choices as compared to the optimal algorithm and therefore, put off working on some of the tasks which precede all other unexecuted tasks. In this situation, the worst case time for which the PSA will be in such a situation is the maximum length of any chain of tasks from start to finish. This is the critical path weight (C_x) of the graph when no task uses more than x processors. Therefore, we can guarantee that at least $p-x+1$ processors will be busy for at least $T_{psa} - C_x$ units of time and at least 1 processor will be busy for C_x units. This means the area of useful work done is given by $(p-x+1)*(T_{psa}-C_x)+1*C_x$ in the worst case. The area covered by the optimum is $T_{opt} * p$. Since we have a worst case for the PSA algorithm, we have:

$$(p-x+1)*(T_{psa}-C_x)+1*C_x \leq p*T_{opt} \tag{9}$$

Case 2 Due to poor scheduling by the PSA, but the optimal algorithm may not have enough work either due to properties of the task graph. In this situation, if the optimal algorithm does not have enough work, the PSA could have work since it could put off working on some tasks. This means the idle time of the PSA for such situations is bounded by that of the optimal algorithm. However, due to poor scheduling, i.e. making the wrong choices, the PSA could have idleness where the optimal algorithm does not. Again, the duration of this situation, by the same argument as above, is bounded by C_x. This means the area $(p-x+1)*(T_{psa}-C_x)+1*C_x$ does include some idle time in the first term, but that is less than or same as the idle time in the term $p*T_{opt}$. Therefore, we can write:

$$(p-x+1)*(T_{psa}-C_x)+1*C_x \leq p*T_{opt} \tag{10}$$

It can be noted that both the cases above give us the same relationship between T_{psa}, T_{opt} and C_x. Further, T_{opt} must be at least C_x, i.e. $T_{opt} \geq C_x$. Therefore, using this fact and either Equation 9 or 10, we have:

$$(p-x+1)*T_{psa} \leq (2p-x)*T_{opt}$$

$$\Rightarrow T_{psa} \leq \frac{2p-x}{p-x+1} * T_{opt} \tag{11}$$

After some re-arrangement of terms, we obtain:

$$T_{psa} \leq 2 + \frac{x-2}{p-x+1} * T_{opt} \tag{12}$$

which is the required result □.

Some special cases of p and x are considered to show the validity of our result.

Corollary 1 $x = p = 1$

This is the uniprocessor case.

$$T_{psa} = T_{opt} \tag{13}$$

which is precisely the case.

Corollary 2 $x = 1$

Each task uses only one processor.

$$T_{psa} \leq (2 - \frac{1}{p}) * T_{opt} \tag{14}$$

this result matches those in [13] and [12].

Corollary 3 $x = \frac{p}{2} + 1$

Each task can use approximately half as many processors available.

$$T_{psa} \leq (3 - \frac{2}{p}) * T_{opt} \tag{15}$$

this result matches those in [15] and [14].

As we have seen in Theorem 1, the PSA algorithm is within a factor of the optimum, the factor being dependent on x. It may be construed at first sight that a good choice of x is one which minimizes this factor. There is, however another aspect to the choice of x. This aspect being the influence x has on the time T_{opt}. Theorem 2 below takes into account this aspect and provides a method of choosing x optimally.

Theorem 2 *The optimum value of the bound on the number of processors to be used by any node in a given MDG with n nodes is given by that value of x which minimizes the expression:*

$$\frac{2 \cdot p}{x} + \frac{(x-2) \cdot p}{(p-x+1) \cdot x} \tag{16}$$

under the assumptions of Section 3.

Proof Consider the perfect schedule time (T_{opt}) for a given processor allocation with a bound x being placed on the number of processors used by any node ($1 \leq p_i \leq x, i = 1, n$). It is limited only by the average finish time (A_x) and the critical path weight C_x.

$$T_{opt} = \max(A_x, C_x) \tag{17}$$

If the bound is changed to p, and the corresponding value of the perfect schedule time is T_{opt}^{min}, we can see that:

$$T_{opt}^{min} = \max(A_p, C_p) \tag{18}$$

It must be noted that for any given problem, T_{opt}^{min} represents the theoretically best possible finish time.

We now consider the relationship between T_{opt} and T_{opt}^{min}. In imposing a bound on the number of processors used by any node, we can reduce the number of processors used by a node from p to x in the worst case. From the computation model, we can see that this can blow up the computation costs of the node by a factor no greater than $\frac{p}{x}$. Since the quantities C_x and C_p represent the critical path weights when each node in the MDG uses at most x and p processors respectively, we can write:

$$C_x \leq \frac{p}{x} C_p \tag{19}$$

From the definition of the average in Equation 7, we can see that it decreases as the number of processors used by the nodes of the MDG are either decreased or kept the same. Since imposing the bound means either decreasing or keeping the number of processors used by a node, we have:

$$A_x \leq A_p \ \Rightarrow A_x \leq \frac{p}{x} \cdot A_p \tag{20}$$

since $\frac{p}{x} \geq 1$.

From Equations 19, 20, 17 and 18 we can write:

$$T_{opt} \leq \frac{p}{x} \cdot T_{opt}^{min} \tag{21}$$

Using Theorem 1, we can establish the relationship between T_{psa} and T_{opt}^{min} as:

$$T_{psa} \leq (2 + \frac{x-2}{p-x+1}) \cdot \frac{p}{x} \cdot T_{opt}^{min} \tag{22}$$

From this expression, we can see that the optimum choice of x is the value that minimizes:

$$\frac{2 \cdot p}{x} + \frac{(x-2) \cdot p}{(p-x+1) \cdot x} \tag{23}$$

which is the required result $\square$.

6 Allocation Algorithms

The allocation algorithms described below are all heuristics built around the PSA.

6.1 Heuristic Change Critical Tasks (CCT)

Step 1 Allocate one processor to each node in the MDG and schedule using the PSA. Denote the finish time obtained to be ft_{last}. Also mark all nodes as changeable.

Step 2a Find and mark all critical path nodes, i.e. all nodes which lie on at least one critical path of the MDG for the current processor allocation.

Step 2b Find the critical path node which offers the biggest change per processor increase and is marked changeable. Increase the node's processor allocation by one. Compute the finish time using PSA, call it $ft_{current}$.

Step 3 The following cases exist with respect to the difference $ft_{last} - ft_{current}$:

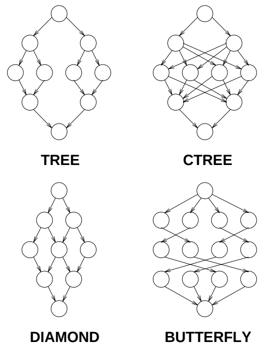

Figure 1: Some regular MDGs used as benchmarks

$ft_{last} - ft_{current} > 0$ which means increasing the processor allocation for the node is a good move. Therefore accept the change and mark all node using less than p processors changeable.

$ft_{last} - ft_{current} \leq 0$ which means increasing the processor allocation for the node is a bad move. Mark the node unchangeable and negate the increase in processor allocation done in Step 2b.

Step 4 Check if there is any node that is marked changeable, if so, go to Step 2a else go to Step 5.

Step 5 Schedule using the PSA and compute the final finish time.

6.2 Heuristic Optimal Bound Change Critical Tasks (OBCCT)

This heuristic uses the theory developed in Theorem 2 to come up with bounds on the maximum number of processors used by any node x. The value of the bound x is used in Step 3.a to mark all nodes using less than x processors changeable instead of the value of p processors used for CCT. All other steps for this heuristic are identical to the ones in CCT.

6.3 Other Heuristics

In addition to the heuristics described above, we also implemented two other heuristics called NAIVE, which allocates p processors to each node in the MDG and UNIP, which allocates only one processor per node in the MDG. These were used for comparison with our heuristics. NAIVE corresponds to what most data parallelizing compilers use currently.

7 Results

The heuristics proposed above were implemented and tried out on some synthetic MDGs. The aim of this exercise was to try and see the effectiveness of the heuristics over naive schemes for these graphs. We chose synthetic graphs since we did not have any MDGs generated from real benchmarks at this stage. The synthetic MDGs are varied and complex in structure. Four of them, Diamond, Tree, Ctree, and Butterfly are regular in structure. We have shown the shape of these graphs in Figure 1. For our experiments we used larger versions of these graphs. In addition, we also used two randomly generated graphs called Random1 and Random2 to make sure our methods worked well for

Benchmark	NAIVE	UNIP	CCT	OBCCT
Random1	1.0	3.15	0.75	0.76
Random2	1.0	3.70	0.76	0.79
Butterfly	1.0	1.86	0.67	0.68
Diamond	1.0	4.55	0.81	0.84
Ctree	1.0	2.43	0.77	0.79
Tree	1.0	2.02	0.89	0.90

Table 1: Execution times (normalized to NAIVE) for various heuristics

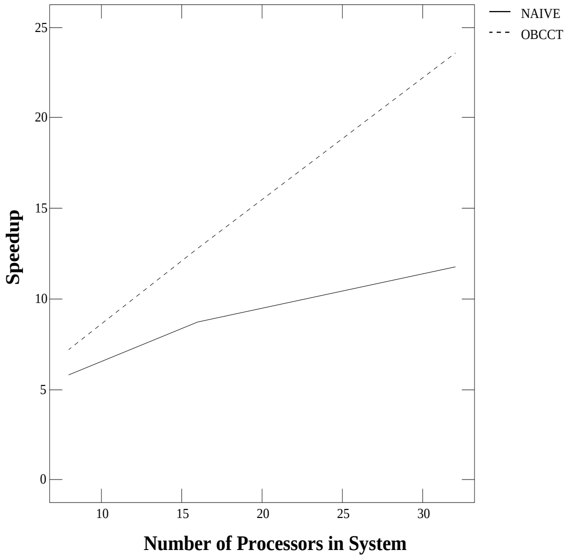

Figure 2: Speedup Curves for OBCCT and NAIVE

irregular MDGs. A pair (α, τ) was generated randomly for each node in the MDG. To check is there were benefits to using our algorithms even if α is low (nodes have almost linear speedups), we purposely kept it between 1 and 10 percent. In addition a communication size L_{ij} was generated randomly for each pair of nodes i and j that need to communicate.

The results obtained are shown in Table 1. From this table, it can be observed that both our allocation algorithms give an improvement over the NAIVE heuristic or the UNIP heuristic. The quality of improvement for the OBCCT algorithm is only slightly inferior to that of the CCT algorithm. Its worst case complexity (not derived due to lack of space), however, is lower than the CCT algorithm. This would make it favorable to use the OBCCT algorithm in practice.

Our algorithms are more effective as compared to the NAIVE heuristic when the number of processors in the system is increased. This effect is clear from Figure 2 showing the speedup curves for OBCCT and NAIVE for the Butterfly benchmark. This shows the OBCCT has a linearly increasing profile whereas the NAIVE heuristic is tending towards a saturating value.

8 Conclusions

Practically feasible algorithms have been presented for the allocation and scheduling of MDGs. Important theoretical results have been derived for with respect to the optimality of the scheduling algorithm used. Results obtained on synthetic benchmark graphs show the potential of our methods even when the nodes of the MDG are highly parallel.

Our future work is aimed at including communication costs in our theoretical analysis. Another area of interest is the development of optimal allocation algorithms to replace the heuristics we are currently using. After this has been done we propose to test our algorithms on real benchmarks.

References

[1] M. Gupta and P. Banerjee, "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers," *IEEE Transactions on Parallel and Distributed Computing*, pp. 179–193, March 1992.

[2] E. Su, D. Palermo, and P. Banerjee, "Automating Parallelization of Regular Computations for Distributed Memory Machines in the PARADIGM compiler," in *Proceedings of the International Conference on Parallel Processing*, 1993.

[3] M. Gupta and P. Banerjee, "A Methodology for High-level Synthesis of Communication on Multicomputers," in *Proceedings of the International Conference on Supercomputing*, 1992.

[4] K. P. Belkhale and P. Banerjee, "Approximate Algorithms for the Partitionable Independent Task Scheduling Problem," in *Proceedings of the International Conference on Parallel Processing*, pp. 72–75, 1990.

[5] K. P. Belkhale and P. Banerjee, "A Scheduling Algorithm for Parallelizable Dependent Tasks," in *Proceedings of the International Parallel Processing Symposium*, pp. 500–506, 1991.

[6] G. N. S. Prasanna and A. Agarwal, "Compile-time Techniques for Processor Allocation in Macro Dataflow Graphs for Multiprocessors," in *Proceedings of the International Conference on Parallel Processing*, pp. 279–283, 1992.

[7] M. Girkar and C. D. Polychronopoulos, "Automatic Extraction of Functional Parallelism from Ordinary Programs," *IEEE Transactions on Parallel and Distributed Computing*, pp. 166–178, March 1992.

[8] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, pp. 22–35, January 1978.

[9] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, 1989.

[10] T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," in *Proceedings of IEEE Supercomputing 91*, pp. 633–642, 1991.

[11] T. Yang and A. Gerasoulis, "A Parallel Programming Tool for Scheduling on Distributed Memory Multiprocessors," in *Proceedings of the Scalable High Performance Computing Conference*, pp. 350–357, 1992.

[12] C. L. Liu, *Elements of Discrete Mathematics.* McGraw-Hill Book Company, 1986.

[13] M. R. Garey, R. L. Graham, and D. S. Johnson, "Performance Guarantees for Scheduling Algorithms," *Operations Research*, pp. 3–21, January 1978.

[14] Q. Wang and K. H. Cheng, "A Heuristic of Scheduling Parallel Tasks and Its Analysis," *SIAM Journal on Computing*, pp. 281–294, April 1992.

[15] J. W. Turek, J. and P. Yu, "Approximate Algorithms for Scheduling Parallelizable Tasks," in *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pp. 323–332, 1992.