Generalized Multiprocessor Scheduling for Directed Acylic Graphs

G. N. Srinivasa Prasanna
7D-311, AT&T Bell Laboratories
Murray Hill, NJ 07974-0636
prasanna@aloft.att.com

Bruce R. Musicus
Bolt, Beranek, & Newman, Inc.
Cambridge., MA 02138.
bmusicus@bbn.com

Abstract

This paper considerably extends the multiprocessor scheduling techniques in [1], and applies it to matrix arithmetic compilation. In [1] we presented several new results in the theory of homogeneous multiprocessor scheduling. A directed acyclic graph (DAG) of tasks is to be scheduled. Tasks are assumed to be parallelizable - as more processors are applied to a task, the time taken to compute it decreases, yielding some speedup. Because of communication, synchronization, and task scheduling overhead, this speedup increases less than linearly with the number of processors applied. The optimal scheduling problem is to determine the number of processors assigned to each task, and task sequencing, to minimise the finishing time.

Using optimal control theory, in the special case where the speedup function of each task is p^α, where p is the amount of processing power applied to the task, a closed form solution for task graphs formed from parallel and series connections was derived [1]. This paper extends these results for arbitrary DAGS. The optimality conditions impose nonlinear constraints on the flow of processing power from predecessors to successors, and on the finishing times of siblings. This paper presents a fast algorithm for determining and solving these nonlinear equations. The algorithm utilizes the structure of the finishing time equations to efficiently run a conjugate gradient minimization, leading to the optimal solution. The algorithm has been tested on a variety of DAGs. The results presented show that it is superior to alternative heuristic approaches.

1 Introduction

In this paper we extend the multiprocessor scheduling results in [1], and apply it to matrix arithmetic compilation [2]. We start with a set of *tasks* (also called nodes or macro nodes) and associated *precedence constraints*, a finite pool of processor resources, and specified *speedup functions* for each task as a function of applied processing power. The classical multiprocessor scheduling problem [3] is to specify work for each processor over time such that the entire set of tasks is computed in the shortest time, satisfying all precedence constraints, and using only the available processor resources. In [1] we generalized multiprocessor scheduling to include parallelism in tasks. We assume that any number of processors can be applied to each task at any time, and that the higher the parallel processing power applied to the task, the faster it can execute (higher its speedup). Because of scheduling, synchronization and communication overhead, the speedup is typically less than linear. Our problem is then to specify both an optimal number of processors applied to a task (task parallelism), as well as an optimal sequencing of tasks. Clearly this is a generalization of classical multiprocessor scheduling, which determines only task sequencing.

In [1], we used optimal control theory to tackle this problem. In the special case where the speedup function of each task is p^α, where p is the amount of processing power applied to the task, a closed form solution for task graphs formed from parallel and series connections was derived, using divide and conquer techniques. In addition, processing power was shown to behave much like electrical charge, passing from task to task along the arcs of the precedence graph. Processing power is released from each task only as it completes, and flows only to those successors of the task that become enabled to run at this moment. All tasks with no successors finish simultaneously at the end of the schedule. These results were applied in [2] for compiling matrix arithmetic.

In this paper we extend these results for arbitrary DAGS. We show that the optimality conditions yield a set of simultaneous nonlinear equations on the flow of processing power from predecessors to successors, and on the finishing times of siblings. This paper presents a fast algorithm for deriving and solving these nonlinear equations. The algorithm utilizes the structure of the finishing time equations to efficiently run a conjugate gradient minimization, leading to the optimal solution. The algorithm has been tested on a variety of DAGs. The results presented show that it is superior to alternative heuristic approaches.

1.1 Application to Matrix Computations

These multiprocessor scheduling results have considerable practical significance. They can be directly applied to compilation of matrix arithmetic, which form the kernel of much signal processing and numerical algorithms [4]. Expressions composed of matrix additions, dot products, matrix multiplies, and inverses commonly dominate the runtimes of these al-

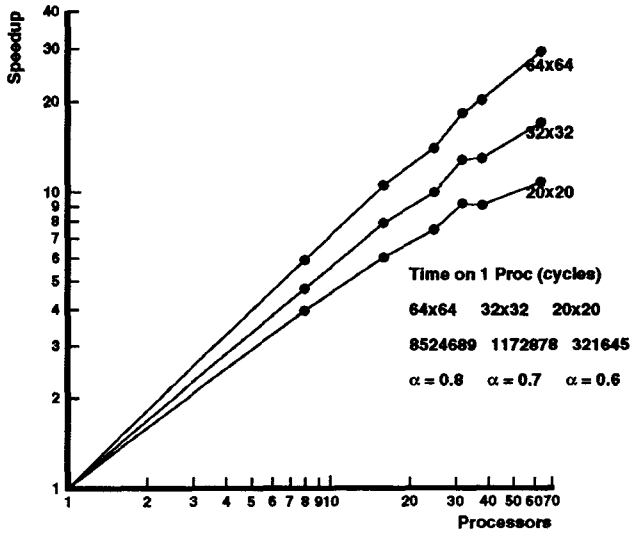


Figure 1: Speedup curves for matrix product on Alewife (drawn using log-log scales).

gorithms, and hence their optimal compilation on multiprocessors is of great interest.

Matrix operators can be systematically decomposed into large numbers of parallel operations. For example, an $N \times N$ matrix product can be decomposed into N^2 dot products, N^3 scalar multiplies and additions, etc. As more processors are applied to compute a matrix operator, its runtime decreases, but less than linearly because of overhead. Optimal compilation of matrix arithmetic involves partitioning each matrix operator into an optimal number of pieces (determining operator parallelism), assigning a processor to each piece, and scheduling the pieces. If the speedup functions of matrix operators can be approximated by p^α, for a suitably chosen α, our generalized scheduling results can be directly applied.

To investigate this issue, speedups of several matrix operators have been measured on the Alewife [5] distributed-shared memory multiprocessor, being constructed at MIT. For example, Figure 1 shows a log-log plot of the speedup curves, for a matrix multiply, for various matrix sizes. Functions of the form p^α will appear as straight lines on a log-log plot, whose slope is the desired parameter α. Since all the curves are roughly straight lines, they are well approximated by p^α. The slopes (α's), however, depend on the size of the matrices, and range from from 0.6 (for 20×20 matrices) to 0.8 (for 64×64 matrices). An average value of $\alpha = 0.7$ can be used for the matrix sizes above.

Based on these approximations, a matrix arithmetic compiler, incorporating the techniques in this work, has been written for the MIT Alewife machine [2]. The compiler takes a matrix expression in LISP like syntax, and first determines an optimal partition and schedule of each operator, using our techniques (Section 4). Then it uses this partition and schedule to generate Multilisp code for the Alewife machine. The runtimes on Alewife (not included) show that our scheduling techniques result in faster code than other heuristic approaches.

In related work, Banerjee et al. [6] have adapted convex programming techniques to solve a problem analogous to ours, using arbitrary speedup functions. In our case, the optimality properties of the p^α speedup function greatly simplifies the optimization, making convex programming techniques unnecessary. Others [7, 3, 8, 9] have primarily dealt with the complexity of scheduling task systems. However, their approaches, relying on approximately solving NP-Hard problems, are quite different from the one pursued here. They do not use speedup abstractions like ours. Sarkar [10] has investigated techniques for partitioning and scheduling program dataflow graphs for execution on multiprocessors. This implicitly involves solving a generalised scheduling problem. However, his approach relies on explicit knowledge of the internal structure (subtasks) of each task. General purpose graph partitioning and classical scheduling algorithms are applied to graphs derived from the original task graph by appropriate task expansion.

Section 2 formulates the generalized scheduling problem in the framework of optimal control theory, and summarizes the optimality results in [1] for p^α speedup. Section 3 uses these optimality constraints to derive a set of equations governing processor flows at each node (*node equations*), and another set of equations governing finishing times of siblings (*loop equations*). Section 4 presents an efficient algorithm for solving this coupled set of equations. Section 5 presents a couple of detailed examples. Section 6 presents some experimental results using this algorithm on a variety of dags. Section 7 concludes.

2 Optimal Control Formulation

We first present a simple intuitive characterization of the generalized scheduling problem. Then we present the optimal control formulation and summarize the results that emerge. A key result imposes strong constraints on the finishing and starting times, and processor allocations, of tasks and their successors.

2.1 Intuition

We start with a (acyclic) graph of tasks, with edges representing precedence constraints. We assume that each task can be treated as a dynamical system, whose state can be changed by applying processing power. Each task starts with state at 0, and must end with its state advanced to completion (state equal to length of task). Tasks which are aggregations of atomic nodes (ie macro nodes), commonly encountered in matrix arithmetic, are well characterised by this model [2]. The state at a time instant corresponds to the fraction of atomic nodes completed by that time instant. As more processors are applied, more atomic nodes are computed per unit time (state changes faster).

The intuition underlying our algorithms is that as we increase the number of processors allocated to a task, overhead of various kinds - scheduling, communication, synchronization - increases. Thereby, the incremental speedup obtained keeps falling, which implies that the speedup functions of the task are *convex*. The convex speedup implies that processing efficiency decreases as we increase the number of processors allocated to a task. Hence overall computation speed is maximized by running concurrently as many tasks as the available parallelism allows, using few processors for each task. In contrast, running the tasks one by one, using all the processors for each task, is slower. Essentially, running many tasks in parallel maximizes the granularity of the threads produced from each task, thus minimizing overhead. This intuition can be given a rigorous foundation using optimal control theory.

2.2 Control Theoretic Formulation

2.3 Formal Specification

Let $\Omega = \{1, \ldots, N\}$ be a set of N tasks to be executed on a system with P processors. Let task i have length L_i. That is, L_i denotes the execution time of the task on a single processor. A set of precedence constraints is specified, wherein task i cannot start until after all its predecessors have finished.

It is convenient to define the *state* $x_i(t)$ of task i at time t to be the amount of work done so far on the task, $0 \leq x_i(t) \leq L_i$. Let t_i be the earliest time at which all predecessors of i (if any) have finished, so that i can begin running. Thus $x_i(t) = 0$ for $t < t_i$, and $x_j(t_i) = L_j$ for all of i's predecessor task j. If task i has no predecessors, $t_i = 0$. The finishing time of task i is denoted by t_i^F.

Let $p_i(t)$ be the processing power (number of processors or processor assignment) applied to task i at time t, and let P be the total processing power available. The $p_i(t)$ are all non-negative, and must sum to at most P. Note that we have allowed the $p_i(t)$'s to be arbitrary continuous time varying functions, thus allowing arbitrary preemptive (generalized) schedules. The assumption of continuity is necessary to apply the optimal control techniques. Strictly speaking, we have to determine the optimal integral processing powers. These are generally close to the optimal continuous $p_i(t)$'s, and can be well approximated by discretizing the optimal continuous $p_i(t)$'s. Unless otherwise specified, all the optimality results stated refer to $p_i(t)$'s allowed to be in the continuous domain.

Finally, assume that once a task's predecessors have finished, the rate at which it proceeds, $dx_i(t)/dt$, depends in some nonlinear fashion on the amount of processor power applied, $p_i(t)$, but not on the state $x_i(t)$ of the task, nor explicitly on the time t. We call this the assumption of *space-time invariant dynamics*. Thus we can write:

$$\frac{dx_i(t)}{dt} = \begin{cases} 0 & \text{for } t < t_i \\ s_i(p_i(t)) & \text{for } t \geq t_i \end{cases} \qquad (1)$$

where $s_i(p_i(t))$ will be called the *speedup function*. It is clear that $s_i(p) > 0$ for $p > 0$. Also, it is a non-decreasing function, since adding more processors can only make the task run faster. Our goal is to finish all tasks in the minimum amount of time t^F, by properly allocating processor resources $p_i(t)$.

In all that follows, we consider the important special case

$$s_i(p) = p^\alpha$$

(all tasks have the same α) for which we had derived very powerful scheduling techniques in [1], which are summarized in Section 2.4. In this case, the runtime of a task i on p processors is L_i/p^α.

2.4 Results for $s_i(p) = p^\alpha$

In this case the optimal processor assignment $p_i(t)$ for any task i does not vary during its computation, but is constant. This processor assignment will be denoted by p_i itself for simplicity. A key result is the following:

Theorem 2.1 *(A) Once a task is runnable, it is assigned (constant) non-zero processor power until it finishes, $p_i(t) = p_i > 0$ for all $t_i < t < t_i^F$. Otherwise, $p_i(t) = 0$ for $t < t_i$ and $t > t_i^F$. This property implies that the optimal schedule can be found by determining a set of constants, the p_i's, instead of the processor assignment functions $p_i(t)$'s.*

(B) When task i finishes, either $t_i^F = t^F$ and the entire graph is finished, or else all the processing power originally allocated to i is reallocated to those successors of i which begin at the same time that i ends. The optimal schedule has to satisfy this flow conservation *property at each task in the graph. Also, the* finishing times *of all those predecessors of a task, which feed processing power to it, should be the same. This timing constraint results in a set of loop equations which the optimal schedule should satisfy.*

(C) The optimal solution for any arbitrary number of processors P can be found by solving for $P = 1$, and then scaling the resulting processor allocations by P. This is referred to as homogeneity.

It follows from this theorem (proposition B) that under p^α dynamics, we can treat processing power as if it were electric charge and precedence constraints as if they were wires. Tasks with no predecessors are initially allocated a given amount of processor "charge"; tasks with predecessors are given no initial processor power. The processing power allocated to an initial task does not change until the task finishes. At this time, the processor "charge" flows out of the task, into its successors. Not all successors of the task get this processor charge - only the ones which become enabled to run at this time because all of their predecessors are finishing. As these tasks finish, their processor power is released and pours into their enabled successors.

This flow continues until finally the tasks with no successors are all running and they all complete at the same moment, t^F.

This flow and finishing time property clearly results in a major reduction in the complexity of scheduling, since it reduces a functional optimization problem over the space of all $p_i(t)'s$, to one determining a set of constant processor allocations $p_i's$. These processor allocations in turn are completely determined by the flow p_{ij} of processing power along each edge from j to i. The homogeneity property (proposition C) implies that the optimal schedule for one total processor power P automatically yields the solution for all P.

For tree structured dataflow graphs, a closed form solution emerges [1]. The optimal schedule can be written down based on recursively decomposing the tree into series and parallel aggregates. The decomposition rules are simple. Tasks in series are equivalent to a single task of length equal to the sum of lengths of the individual tasks, and have the same processor allocation. Tasks in parallel merge to a single task, whose length is the $\ell_{1/\alpha}$ norm of the individual tasks. Processors are allocated in proportion to the $\ell_{1/\alpha}$ norm of task lengths, which equalizes run times of all the parallel tasks.

For general DAGS, closed form solutions do not emerge. The flow and finishing time constraints have to be explicitly solved.

3 Equations for the Optimal Schedule

The optimality theorem 2.1 imposes strong (nonlinear) constraints on the optimal solution. In this section we describe the nature of these constraints. A fast (polynomial time) gradient based technique for the solution of this system is described in Section 4. This takes time $O(E^2+EN+I(N+E))$ per iteration, where N is the number of nodes, E is the number of edges in the task graph, and I is a constant. In all that follows, we assume, without loss of generality, that the task graph has a unique start node (task) s and a unique final node f. The total number of processors is, as before, denoted by P.

3.0.1 Flow Conservation - Node Equations

At each task i, the processing power p_i allocated to those predecessors which finish when i starts, flows into i. These flow conservation equations are of the form

$$N_i = \sum_{j \in Pred(i)} p_{ij} - \sum_{j \in Succ(i)} p_{ji} = 0,\ i = 1 \ldots N-2 \tag{2}$$

at interior nodes, where $Pred(i)$ and $Succ(i)$ refer to those predecessors of i which feed processing power to it, and those successors of node i to which node i releases its processing power (see also Section 3.1).

At the start task s, we have

$$N_s = P - \sum_{j \in Succ(s)} p_{js} = 0$$

Finally, at the final task f,

$$N_f = \sum_{j \in Pred(f)} p_{fj} - P = 0$$

These equations are analogous to the Kirchoff's Current law (KCL) equations of electrical circuit theory. It is easily seen that $N-1$ of these equations are linearly independent.

3.0.2 Finishing Time Constraint - Loop Equations

At each node, the finishing times of all predecessors which feed this node must match. This constraint can be translated into a set of loop timing equations, in a manner analogous to the Kirchoff's voltage law of circuit theory. Basically, a breadth first search (BFS) spanning tree of the task graph is constructed. The $E-N+1$ fundamental loops resulting from this spanning tree yield a necessary and sufficient set of timing equations. These equations are of the form

$$LO_i = \sum_{j \in Loop_i} \pm t_{jr} = \sum_{j \in Loop_i} \pm \frac{L_j}{{p_j}^{\alpha}} = 0, i = 1 \ldots (E-N+1) \tag{3}$$

where

$$p_j = \sum_{k \in Pred(j)} p_{jk}$$

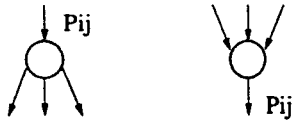
is the total processing power allocated to node j from its predecessors. L_j is the length of task j. We have also introduced the notation $t_{jr} = t_j^F - t_j$ to refer to the total run time of task j (equal to $\frac{L_j}{{p_j}^{\alpha}}$). The sign ($\pm$) assigned to the coefficient refers to the direction in which the particular task is being traversed in the directed loop. Note that the start and final nodes in the loop do not appear in the timing equation. This can in general lead to a multiplicity of optimal solutions.

The $N-1$ node flow conservation equations, together with the $E-N+1$ loop timing equations completely determine the E processor flows p_{ij}'s (see also Section 3.1). This set of equations can be solved very fast, as is shown in Section 4.

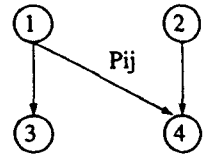
3.1 Non Essential Flows - Node Augmentation

The description above has been deliberately simplified to avoid a potential complication arising from the multiple edges available for flow from a task to its successors. Flow has to necessarily occur along certain edges, and may or may not appear on others.

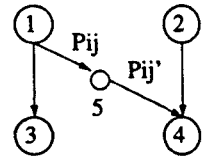
An edge can be classified as *flow-essential* or *non-flow-essential* depending on whether it can sustain a zero flow or not. In Figure 2 (a), the task has either a single incoming edge, or a single outgoing edge. In both cases the flow p_{ij} on that edge has to be necessarily non-zero, else the task either cannot get any



(a) Essential Flows Pij



(b) Non Essential Flow Pij



(c) Node Augumentation to convert Pij to essential flow

Figure 2: Essential and Non Essential Flows

processors to compute it, or cannot release its processors to successors after it is done.

In Figure 2 (b), the edge $1-4$ between tasks 1 and 4 is non-flow-essential, since task 1 can release its processing power to task 3 after task 1 is completed, and task 4 can obtain its processing power from task 2 after task 2 is completed. If non-zero flow p_{ij} does indeed occur along edge $1-4$, nodes 1 and 2 will have to finish together (Theorem 2.1), and this is automatically guaranteed by solving the loop equations. If flow does not occur along edge $1-4$, the precedence constraint implies that node 1 can finish no later than the start time of node 4. Thus, a non-flow-essential edge imposes a *latest-finish-time* constraint on its source node.

Latest-finish-time constraints cause the loop equations to become inequalities, complicating their solution. It is not possible to predict apriori which equations will be satisfied as equalities (this corresponds to non-zero flow on the corresponding non-flow-essential edge). All possible combinations will have to be searched in general - an exponential process. However, a simple graph conversion fixes this problem. We insert a new *dummy* node in each non-essential edge, with very small length ϵ, yielding a new *node-augmented* graph, without non-essential flows (Figure 2 (c)). It is clear that node augmentation does not unduly increase the optimal finish time or the size of the DAG. At most $O(E)$ new nodes are created.

```
processor_flow(graph)
  Phase 1:
  Remove_transitive_edges(Graph);
  Node_augment(Graph);
  Phase 2:
  N= Node_Equations(Graph);
  LO= Loop_Equations(Graph);
  Phase 3:
  ON = Orthonormal Basis(N);
  Flow=Initial_Flow(Graph);
  Repeat
    Raw Gradient = Gradient(ERR,Flow);
            find gradient at  flow vector
    Delta = Corrected Gradient =
         Remove_component(Raw Gradient,ON);
    F = linemin(ERR,Flow,Delta);
         update flow vector
         (Minimize ERR along (Flow + k Delta))
  until convergence
```

igure 3: Psuedo Code for Processor Flow Algorithm

4 Solution Method

The $E-N+1$ loop equations (Equations 3) are nonlinear in the processor flows $p'_{ij}s$. Solving a coupled nonlinear system is very difficult in general, but the strict convexity and monotonicity of the speedup function p^{α} greatly simplifies the task, by facilitating gradient based techniques.

The $E-N+1$ loop equations are solved simultaneously, using the sum of squares error criterion

$$ERR = \sum_i LO_i^2, i = 1 \ldots (E-N+1) \qquad (4)$$

This has the gradient

$$\nabla ERR = 2 \sum_i LO_i \nabla LO_i, i = 1 \ldots (E-N+1)$$

Note however, that the raw gradient cannot be directly used for minimization, since it in general violates the node equations. We have to use only that component which lies inside the subspace spanned by the node equations. Note that other methods like multidimensional Newton-Raphson can also be used for solving the nonlinear system.

4.1 Processor Flow Allocation Algorithm

The Constrained Gradient algorithm used to solve the system of Equations 2 and 3 is shown in a simplified form in Figure 3. Since there are E flows p_{ij}, we perform a E-dimensional minimization.

The first task is to remove transitive edges (which have zero flow anyway), and do node-augmentation. Node-augmentation can be skipped if there are only a few non-essential edges - a complete search will yield a

more accurate solution. Then, for the modified DAG, we determine the node equations N_i and loop equations LO_i in time $O(N+E+EN)$. A breadth first spanning tree (BFS tree) of the DAG is constructed in the process of determining the loop equations. Then, an orthonormal basis ON is determined for the set of node equations - the standard Gramm-Schmidt procedure takes time $O(EN^2)$. An initial flow is determined using a simple algorithm wherein the total inflow at a task is distributed equally amongst all successors, taking time $O(E+N)$.

The gradient descent begins by determining the gradient of the error criterion $ERR(F)$ at the current flow vector F. This takes time $O(E^2)$, since there are $O(E-N+1) \approx O(E)$ gradients to be summed, each with E components. This *raw gradient* in general violates the flow conservation equations. Hence the next step removes any component not in the hyperspace spanned by the node equations N_i. The projection of the raw gradient on each basis vector in ON is summed, yielding a corrected gradient Δ, and taking time $O(EN)$. This correction step can be eliminated, provided we use the node equations to eliminate $N-1$ extra variables in the loop equations apriori, yielding an even faster algorithm.

In the next step, the error ERR is minimized along the corrected gradient, using standard 1-dimensional line minimization techniques, ie., the constant k is determined such that

$$ERR(F + k\Delta)$$

is minimized. A variety of techniques can be used here, varying from simple golden section search, to Newton-Raphson methods [4]. These methods require efficient calculation of ERR and its (1-D) derivative with respect to k. At each iteration of the line minimization, ERR is efficiently calculated by computing the node finishing times along the BFS tree used to generate the loop equations, taking the differences in finishing time at the forward and reverse branches of each fundamental loop, squaring, and summing, taking time O(N+E) overall. It can be shown that the derivative of ERR can also be calculated in a similar fashion.

The algorithm converges when all loop timing errors are adequately small, relative to the computed finishing time of the graph.

The line minimization hence takes time O(I(N+E)) in all, where I is the number of iterations used in the 1-D minimization (which can be bounded by a moderate constant). The overall time taken for each update of the flow vector F is then

$$O(E^2 + EN + I(N+E))$$

which is a low order polynomial in the input size of the task graph. The sparsity of the task graph can be exploited to further reduce the run time.

As presented above, the minimization proceeds in the direction of the corrected gradient. In practice, the optimal direction to minimize is not exactly the gradient, but the conjugate gradient, yielding the well known conjugate gradient algorithm [4]. The algorithm uses only the corrected gradient at this iteration and the previous iteration, and hence does not differ materially from what is presented above. Our implementation uses the conjugate gradient algorithm.

4.2 Suboptimal Heuristics for Processor Allocation

We have compared the processor flow algorithm with two heuristic scheduling approaches – Naive and Greedy [1, 2]. The Naive heuristic runs each task in sequence, on all the available processors. Clearly this is very inefficient. However, the heuristic is considered because it is commonly used in scheduling matrix computation. The Greedy heuristic is an as-soon-as-possible greedy schedule. A task is run at the earliest time at which it is ready (all predecessors completed). All tasks that are ready at a certain time are started together and finished together. Computation proceeds as a wavefront picking up task sets that get ready in succession. This heuristic is selected, since it is quite good in many cases, especially if the nodes lengths are not very different (Section 6). These two heuristics are illustrated in detail in Section 5 below.

5 Detailed Examples

5.1 DAG with all Essential flows

Consider the dag in Figure 4 (a). It has five nodes (three matrix multiplies and two additions). All of its 8 edges are clearly flow-essential. Flow conservation at nodes s (start node), $1, 2, 3, 4$, and 5 respectively implies the node equations below:

$$\begin{array}{lcl;{5pt/5pt}lcl} p_{1s}+p_{2s} & = & P & p_{1s}-p_{31}-p_{41} & = & 0 \\ p_{2s}-p_{52} & = & 0 & p_{31}-p_{53} & = & 0 \\ p_{41}-p_{f4} & = & 0 & p_{52}+p_{53}-p_{f5} & = & 0 \end{array}$$

Figure 4 (b) shows a BFS tree, together with two fundamental loops - $s1352$ and $s14f52$. The loop equation corresponding to $s1352$ equalizes the finish times of nodes (tasks) 2 and 3. The loop equation corresponding to $s14f52$ equalizes the finish times of nodes 4 and 5. The respective loop equations are

$$\begin{aligned} \frac{L_1}{p_{1s}^{\alpha}} + \frac{L_3}{p_{31}^{\alpha}} - \frac{L_2}{p_{2s}^{\alpha}} &= 0 \\ \frac{L_1}{p_{1s}^{\alpha}} + \frac{L_4}{p_{41}^{\alpha}} - \frac{L_2}{p_{2s}^{\alpha}} - \frac{L_5}{(p_{52}+p_{53})^{\alpha}} &= 0 \end{aligned}$$

These 8 equations completely determine the processor flows along the 8 edges and hence the optimal schedule. This system was solved (Section 6 and Section 4) assuming that the task size of an $N \times N \times N$ matrix multiply is $M = 2N^3$ (N^3 multiplies and additions), and the task size of an $N \times N$ matrix addition is N^2 (only additions). 32×32 matrices and $P = 64$ processors were used. The resultant schedule, in Figure 4 (c), splits processors in such a way that the three matrix multiplies get roughly $P/3 = 21$ processors

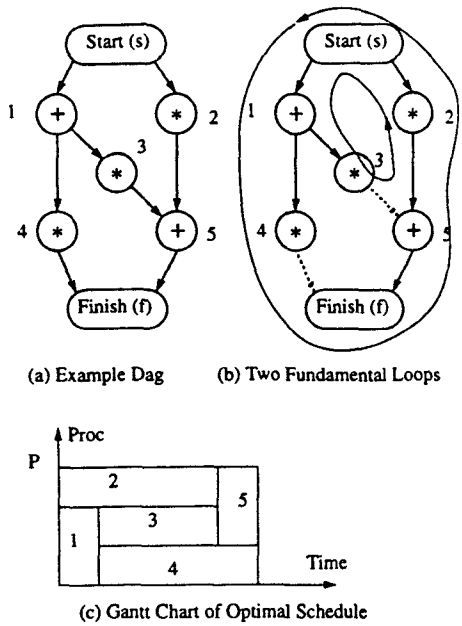


Figure 4: DAG with Essential flows

each. Note that the homogeneity property (Theorem 2.1 (C)) implies that this solution holds for arbitrary P, by a suitable scaling. The computed finishing time is approximately $2.18M/P^{\alpha}$, where $M = 2N^3$ is the time taken to perform a matrix multiply on 1 processor.

Figure 5 compares the optimal schedule with those obtained using the Naive and Greedy Heuristics. The Naive schedule in Figure 5 (c) runs all the five tasks in sequence, assigning all $P = 64$ processors to each. This clearly incurs the maximum overhead, and is the slowest possible schedule, with a finishing time of about $3M/P^{\alpha}$. The Greedy schedule in Figure 5 (d) runs matrix multiply 2 in parallel with matrix addition 1. Processors are distributed between the two tasks in such a way that they finish together - the multiply gets most of the processors. Then multiplies 3 and 4 are run in parallel, each with $P/2 = 32$ processors. Then the final addition is computed. The resultant schedule is again sub-optimal, with a finishing time of approximately $2.62M/P^{\alpha}$, since it missed the opportunity to run all the three matrix products together. The optimal schedule, with a finishing time of $2.18M/P^{\alpha}$ is 38 % faster than the Naive Heuristic, and 20 % faster than the Greedy Heuristic. Naive. This testifies to the utility of our scheduling techniques.

5.2 DAG with Non-Essential flows

Consider the dag in Figure 6 (a). It has four nodes (two multiplies and two additions). The edge between 1 and 4 is non-flow-essential.

Assume that non-zero flow exist on edge 14. Figure

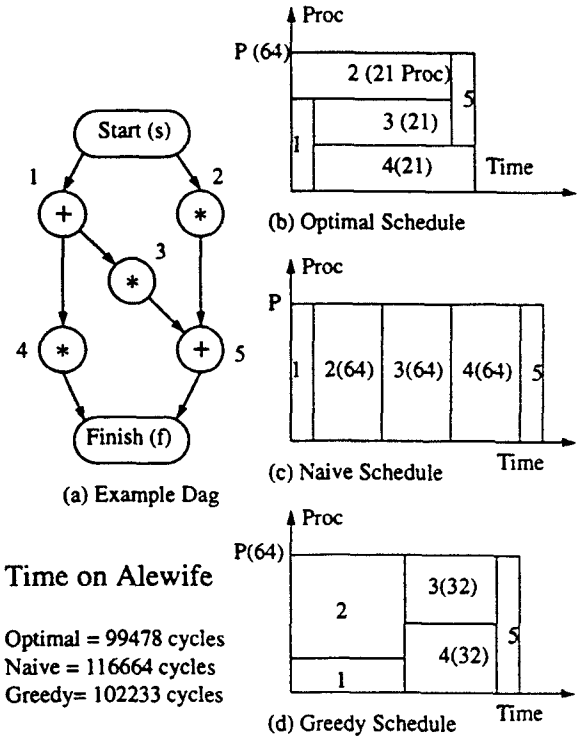


Figure 5: Comparison of Optimal, Naive, and Greedy Schedules

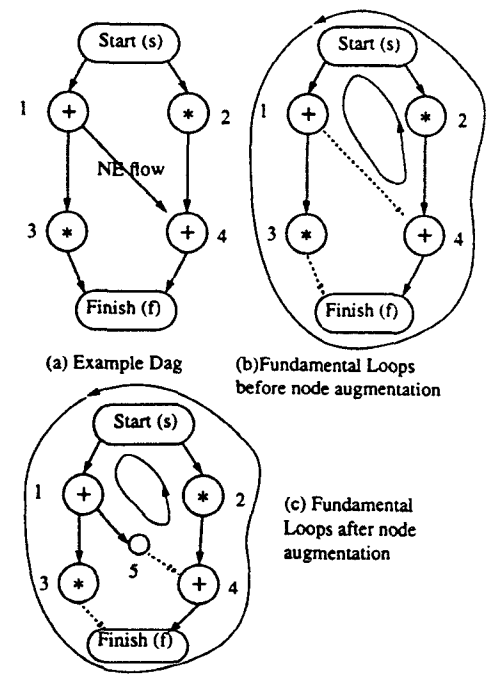


Figure 6: DAG with Non Essential flow

6 (b) shows a BFS tree, together with two fundamental loops - $s142$ and $s13f42$. The loop equation corresponding to $s142$ equalizes the finish times of nodes (tasks) 1 and 2. The loop equation corresponding to $s13f42$ equalizes the finish times of nodes 3 and 4. These two constraints immediately imply that nodes 1 and 2 run in parallel, finishing simultaneously, and then nodes 3 and 4 run in parallel. Since the multiplies are much larger than adds, the finishing time of the DAG is the roughly the time taken to compute two multiplies, $2M/P^\alpha$. However, this is clearly suboptimal, since if no flow occurs on edge 14, the two branches run together in parallel, using $P/2$ processors each, and maximizing efficiency. The run time is then $M/(P/2)^\alpha = 2^\alpha M/P^\alpha$.

Figure 6 (c) shows how node-augmentation enables the algorithm to solve for the optimal schedule, ignoring non-essentiality. A very small task, 5, is introduced in edge 14. Now task 1 is no longer constrained to finish together with task 2. It can finish faster, pouring most of its processors to task 3, and assigning a small amount to task 5, to make it finish together with task 2. Clearly this effectively allows the two branches to be computed independently, leading to a solution which is very close to optimal.

6 Algorithm Results

In this section we present scheduling results from the processor flow algorithm. The graphs chosen include both dags explicitly synthesized to evaluate the processor flow algorithm, as well as dags corresponding to frequently encountered matrix algorithms. All tasks correspond to matrix operators. Operator sizes are determined by a simple count of the number of adds, multiplies, etc. performed by the operator - eg. an $N \times N$ matrix product has N^3 scalar adds and multiplies (scalar adds and multiplies are assumed to take the same time). The matrix size N is taken to be 32×32, which is a reasonable choice for (dense) matrix algorithms. We compare the results from the processor flow algorithm with those from the Naive and Greedy Heuristics (Section 4.2).

Several parameters have to be chosen for the processor flow algorithm. Firstly, the *homogeneity* property of the p^α speedup function (Theorem 2.1) implies that the solution for any arbitrary number of processors P can be found by solving for $P = 1$, and then scaling the resulting processor allocations by P. Hence, in what goes below, the processor flow algorithm has been run with $P = 1$. Of course, in a real multiprocessor schedule, we would have to scale by P (typically large), to get the precise processor assignment. Next, the speedup parameter was taken to be $\alpha \approx 0.7$, based on measurements made on actual matrix multiplies of this size, on the Alewife multiprocessor (Section 1.1). Finally, the size of the dummy node used to perform node-augmentation was taken to be 2.5% of the largest task in the graph (Section 3.1). This is small enough not to unduly perturb the optimal solution, yet large enough not to cause undue convergence difficulties. Widely varying task sizes can cause the system of loop equations to become ill conditioned, causing convergence problems. To facilitate convergence, the minimum task size was initially restricted to be 5% of the largest. The solution obtained was used as a start point for the next stage, where the minimum task size was halved. This procedure was continued till the smallest task size was allowed.

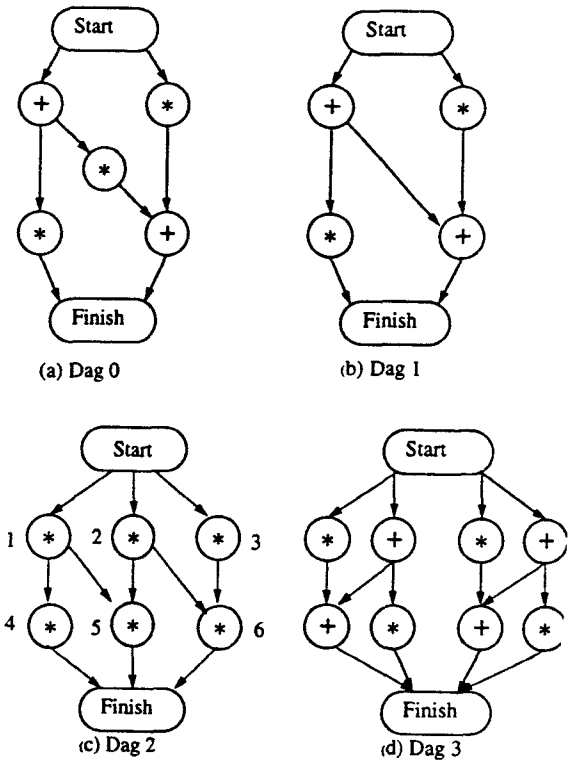


Figure 7: DAG dag_0, dag_1, dag_2, and dag_3

6.1 DAG's Used

Dags Dag_0 (Figure 7 (a)) is the dag discussed in Section 5. It has three matrix multiplies and two additions. All its edges are flow-essential. This provides a simple test of the processor-flow algorithm on non-trees. Dag_1 has a single non-essential edge. Dag_2 (Figure 7 (b)), with 3-outputs, has 3 non-essential edges, 1−5, 2−5, and 2−6. As such upto 8 combinations of non-essential flows are potential candidates for the optimal solution. This dag provides a simple test of the node augmentation technique of the processor flow algorithm. Dag_3 is two copies of Dag_1, run in parallel, and has 2 non-essential edges.

Lattices All edges of these two lattice structured dags ($Latt_1$, $Latt_2$ in Figure 8) are flow-essential. $Latt_2$ is basically two copies of $Latt_1$ run in series. The algorithm has also been tested on lattices which are 3, 4, and 8 copies of $Latt_1$ in series. The optimal flows vary widely in magnitude, and are

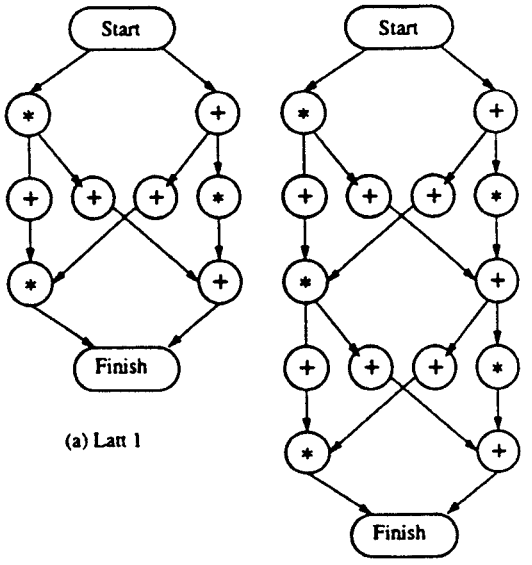


Figure 8: Lattices $Latt_1$ and $Latt_2$

very far away from the initial equally distributed flows. The forward edges from one stage to the next have significant processor flows, with the left side being larger than the other. The cross edges, containing only additions, have very small flow. Hence these furnish difficult convergence tests for the processor-flow algorithm.

Polys $Poly_8$ and $Poly_{16}$ (not shown) are matrix polynomials, of degree 8 and 16, commonly encountered in matrix arithmetic. $Poly_8$ has 2 non-flow-essential edges, while $Poly_{16}$ has 10. $Poly_{16}$ provides a stringent test of node-augmentation.

Strass Figure 9 shows a matrix inverse using Strassen's method [4, page 75]. Two inverses, six matrix multiplies and two matrix additions are being performed. This graph provides a practical example of an relatively complex unstructured matrix computation, for which our techniques are useful. It has both transitive and non-essential edges.

6.2 Analysis of Results

The results on each of these examples are tabulated in Table 1. We tabulate the optimal run time as determined by the algorithm (scaled by P^α), and the total number of iterations (updates of the flow vector). For purposes of comparison, the time as determined by the Naive and Greedy Heuristics is also tabulated. The last two columns (SPN and SPG) show the ratio of these times to the optimal, and indicate the gains obtained by using the optimal algorithm.

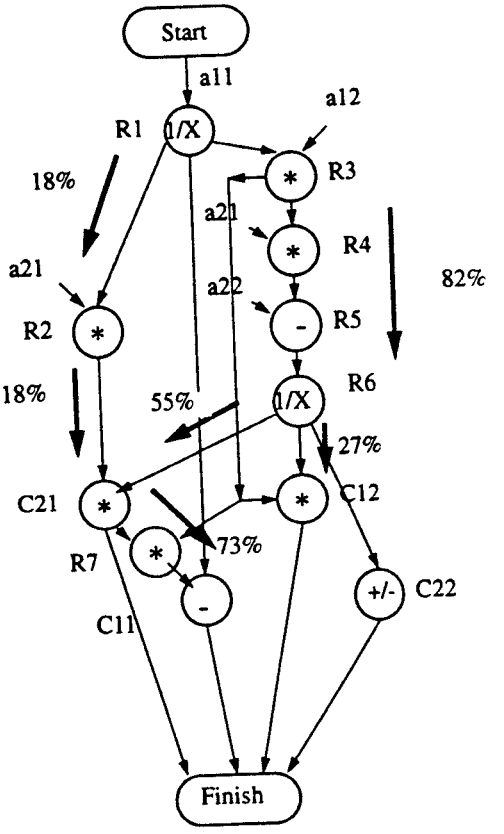


Figure 9: Strassen Matrix Inverse. Thick arrows show processor flow

Graph	T_{Opt}	Iter	T_N	T_G	SPN	SPG
Dag_0	2.18	5	3.03	2.64	1.39	1.22
Dag_1	1.66	20	2.05	2.02	1.23	1.22
Dag_2	4.4	16	6.0	4.4	1.36	1.0
Dag_3	2.7	89	4.1	3.25	1.50	1.2
$Poly_8$	5.28	7	7.02	5.28	1.33	1.0
$Poly_{16}$	9.68	248	15.02	9.68	1.57	1.0
$Strass$	6.88	6	8.06	7.28	1.17	1.06
$Latt_1$	2.57	189	3.08	3.01	1.20	1.17
$Latt_2$	4.24	278	5.14	5.01	1.22	1.18
$Latt_3$	5.86	205	7.20	7.01	1.23	1.20
$Latt_4$	7.42	345	9.27	9.01	1.25	1.21
$Latt_8$	14.05	648	17.52	17.01	1.25	1.21

Table 1: Results from the Processor Flow Algorithm

The speedups over the Naive algorithm vary from a low of 1.14 to a high of 1.57 (57%), while those over the Greedy algorithm vary from 1.0 to 1.22 (22%). Clearly the gains from the optimal algorithm are substantial (the gains can be even higher for lattice-structured graphs with more parallel paths than $Latt_1/Latt_2$). $Poly_{16}$ and *Strassen* nicely illustrate the properties of the optimal solution, and we analyse them below.

$Poly_{16}$ has 10 non-flow-essential edges. The processor flow algorithm correctly determined the correct set of non-zero flows using the node-augmentation technique of Section 3.1. The flows varied widely in size, from a low of 0.0015 to a high of 0.6, well over two orders of magnitude. Two edges had a very small flow of 0.0015. The rest had flows at least 20 times higher. This indicates that these two edges are non-flow-essential, testifying to the utility of node-augmentation.

Strassen's optimal schedule (Figure 9) shows a gain of 17 % over the Naive and a gain of 6 % over the Greedy heuristic. The processor flow algorithm assigns 82 % of the processors to the longer of the two branches $R_3 - R_4 - R_5 - R_6$, and only 18 % for the smaller branch R_2. R_2 and R_6 finish at the same time. Most to the processors (55 %) computing R_6 are transferred to compute c_{21}, R_7, and c_{11}. The remaining (27 %) compute c_{12} and c_{22}. Clearly this schedule cannot be derived by any heuristic like Naive or Greedy.

The number of iterations for all cases is less than 350, except for $Latt_8$, which takes 648 iterations. The run times for our substantially sub-optimal implementation is less than 3 minutes on a SPARC-10 in all cases, except for $Latt_8$, which takes around 12 minutes. However, $Latt_8$ is a very large task graph, with 17 matrix products, and 33 additions. So many operators typically result from extensive loop unrolling/ software pipelining employed at the macro node level, in a matrix expression compiler. In addition the matrix additions are very small in size compared to the multiplies (1.6 %). It is encouraging to note that the algorithm converges even for this large graph, with very different task sizes. The algorithm can hence be effectively used to determine the optimal node parallelism (number of processors allocated to node) and node sequencing, in a matrix arithmetic compiler.

7 Conclusion

In this paper we have presented a fast algorithm for the determining optimal task parallelism and sequencing (generalized scheduling), for arbitrary dags, considerably extending the previous results in [1]. We assume that the speedup function of each task was p^α, for the same α. The optimal schedule is shown to be the solution of a coupled nonlinear system of equations, analogous to the KCL and KVL equations of electrical circuit theory. Our algorithms use gradient based techniques to solve this system. The runtimes are a low order polynomial ($O(E^2 + EN + I(N + E))$ per iteration), in the size of the task graph. The algorithm was tested on a wide variety of dags representative of typical matrix computation. It takes only a few minutes on a SPARC-10 for all but the largest graphs. It is also robust enough to handle dags with widely varying task sizes. The optimal solutions found are upto 57 % faster than those derived using alternative heuristic approaches.

Our techniques are directly applicable to compilation of matrix computations. Indeed we have shown that the assumption of p^α speedup is reasonable for matrix operations, and have written a matrix expression compiler [2], incorporating the ideas presented here. Results on the MIT Alewife machine (not included) have shown the superiority of this scheduling technique over commonly used alternatives.

References

[1] G.N.Srinivasa Prasanna and Bruce R. Musicus. Generalised Multiprocessor Scheduling Using Optimal Control. In *Third Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 216–228, July 1991.

[2] G.N.S. Prasanna, A. Agarwal, and B. R. Musicus. Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory. *IEEE TPDS*, July 1994.

[3] Coffman E.F., Jr., editor. *Computer and Job Shop Scheduling Theory.* John Wiley and Sons, N.Y., 1976.

[4] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipies The Art of Scientific Computing*, chapter 10. Cambridge University Press, 1986.

[5] A. Agarwal et al. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. In *Workshop on Scalable Shared Memory Multiprocessors.* Kluwer Academic Publishers, 1991. Also MIT/LCS Memo TM-454, 1991.

[6] K. P. Belkhale and P. Banerjee. Scheduling Algorithms for Parallelizable Tasks. Center for Reliable and High-Perf. Computing, CSL., Univ. of Illinois, Urbana IL-61801, 1993.

[7] M. Blazewicz, J. Drabowski and J. Welgarz. Scheduling multiprocessor tasks to minimise schedule length. *IEEE Transactions on Computers*, C-35(5):389–393, 1986.

[8] J. Du and J.Y.T Leung. Complexity of Scheduling Parallel Task Systems. *SIAM J. Discrete Math.*, 2(4):473–487, November 1989.

[9] C.C Han and K.J. Lin. Scheduling Parallelizable Jobs on Multiprocessors. In *IEEE Conf. on Real-Time Systems*, pages 59–67, 1989.

[10] V. Sarkar. Partitioning and Scheduling Programs for Multiprocessors. Technical Report CSL-TR-87-328, Ph.D Thesis, Computer Systems Lab., Stanford University, April 1987.