Using Utilization Profiles in Allocation and Partitioning for Multiprocessor Systems

John D. Evans
Robert R. Kessler

UUCS-92-037

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

October 29,1992

Abstract

The problems of multiprocessor partitioning and program allocation are interdependent and critical to the performance of multiprocessor systems. Minimizing resource partitions for parallel programs on partitionable multiprocessors facilitates greater processor utilization and throughput. The processing resource requirements of parallel programs vary during program execution and are allocation dependent. Optimal resource utilization requires that resource requirements be modeled as variable over time. This paper investigates the use of program profiles in allocating programs and partitioning multiprocessor systems. An allocation method is discussed. The goals of this method are to (1) minimize program execution time, (2) minimize the total number of processors used, (3) characterize variation in processor requirements over the lifetime of a program, (4) to accurately predict the impact on run time of the number of processors available at any point in time and (5) to minimize fluctuations in processor requirements to facilitate efficient sharing of processors between partitions on a partitionable multiprocessor. An application to program partitioning is discussed that improves partition run times compared to other methods.

Using Utilization Profiles in Allocation and Partitioning for Multiprocessor Systems

John D. Evans and Robert R. Kessler

April 21,1992

Abstract

The problems of multiprocessor partitioning and program allocation are interdependent and critical to the performance of multiprocessor systems. Minimizing resource partitions for parallel programs on partitionable multiprocessors facilitates greater processor utilization and throughput. The processing resource requirements of parallel programs vary during program execution and are allocation dependent. Optimal resource utilization requires that resource requirements be modeled as variable over time. This paper investigates the use of program profiles in allocating programs and partitioning multiprocessor systems. An allocation method is discussed. The goals of this method are to (1) minimize program execution time, (2) minimize the total number of processors used, (3) characterize variation in processor requirements over the lifetime of a program, (4) to accurately predict the impact on run time of the number of processors available at any point in time and (5) to minimize fluctuations in processor requirements to facilitate efficient sharing of processors between partitions on a partitionable multiprocessor. An application to program partitioning is discussed that improves partition run times compared to other methods.

1 Introduction

Program Allocation[1] is the process of distributing work among the processors of a multiprocessor system. Effective program allocation is essential in achieving the goal of increased computation speed. **Multiprocessor partitioning** is the process of subdividing a multiprocessor system into separate subsystems (**partitions**) of processors. Partitionable multiprocessor systems have several advantages. Multiple programs or multiple subsystems within a single program may use the multiprocessor systems resources simultaneously and thereby serve an increased number of users and obtain increased system utilization and throughput. Multiprocessor partitioning and program allocation are interdependent. Multiprocessor partitioning determines the resources available for program allocation. Program allocation information is used in partitioning to estimate the run times for different sized partitions. The effective use of partitionable multiprocessor systems requires both effective partitioning and effective allocation. General problems of multiprocessor partitioning and program allocation have been shown to be NP-Hard problems[16, 11]. Because of this, efficient suboptimal approaches have been the primary avenue of progress.

In this paper we investigate the interaction between processor utilization patterns determined by allocation and the number of processors available to a partition determined by partitioning. We discuss

[1]The terminology of partitioning and allocation varies. The operational definitions used here are given.

the use of utilization profiles in allocating programs for parallel computation to facilitate multiprocessor partitioning. An allocation method is presented that addresses several basic goals:

1. Minimize program execution time.
2. Minimize the total number of processors used.
3. Characterize variation in processor requirements over the lifetime of a program.
4. Accurately predict the impact on run time of variation in the number of processors available at any point in program execution.
5. Minimize fluctuations in processor requirements to facilitate efficient sharing of processors between partitions on a partitionable multiprocessor.

An application of the allocation method is presented that optimizes partitions generated by existing multiprocessor partitioning algorithms and improves on the run times and efficiency produced by these methods.

Section 2 discusses background and related work. Section 3 discusses observed relationships between utilization profiles and program allocation. Section 4 presents a new allocation method using utilization profiles. In Section 5 we review simulation tests of the profiling/allocation method. Section 6 presents an application of the profiling/allocation method to optimize parallel multiprocessor partitions. Conclusions are presented in Section 7.

2 Background and Related Work

The work reported here makes use of several concepts that have wide application in parallel processing. The following working definitions are used here:

Tasks are single threaded sequences of instructions. Task execution can begin when all required input data is ready, and result data becomes ready for subsequent tasks when execution completes. Tasks may run to completion without interruption or synchronization.

Work is the amount of computation carried out by a task or program.

Task precedence graphs (or simply task graphs) are often used to represent data driven parallel programs [4, 1, 18] (see Figure 1). Graph nodes represent units of computational work (tasks) while arcs represent data dependencies between the tasks. Tasks are weighted to indicate the amount of **work** they represent.

Utilization: The number of processors in use at a point in time.

Utilization Profiles characterize the execution of a task graph. Figure 2(b) shows an allocation of the task graph from Figure 1 on four processors and Figure 2(a) shows the corresponding utilization profile. The utilization profile indicates the number of processors used at any point in time during execution. Profile shape is dependent on allocation and task graph characteristics. The profile representation used here is a list of **segment** pairs (U, W) where **U** is processor utilization per unit

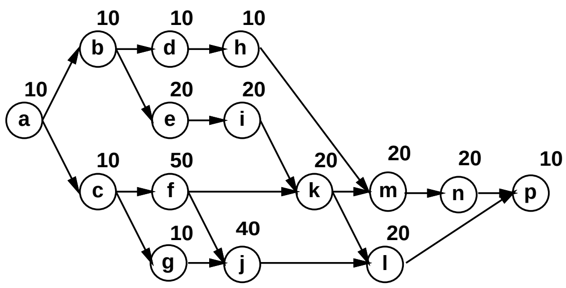

Figure 1: Precedence Graph

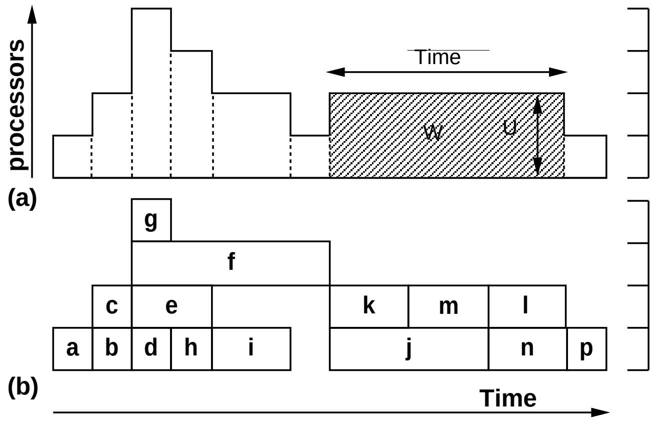

Figure 2: Utilization Profile and Gannt Chart

time and **W** is the work done during the segment. The relationship between **Time**, **W** and **U** is given by the following equation:

$$Time = \frac{W}{U} \tag{1}$$

Critical Path of a task graph is the longest precedence related sequence of tasks in the graph.

Average Parallelism [5] is the ratio of the total work of a task graph divided by the critical path length of the task graph.

$$Par_{avg} = \frac{W}{cpl} \tag{2}$$

Speedup is the graph execution time on a single processor divided by the execution time on the multiprocessor system. Equivalently, speedup equals the average profile segment utilization.

$$Su = \frac{Time_{seq}}{Time_{par}} = \frac{\sum_{i=1}^{maxseg} W_i}{\sum_{i=1}^{maxseg} Time_i} = U_{average} \tag{3}$$

Efficiency is the speedup divided by the number of processors used.

$$Efficiency = \frac{Su}{N} \tag{4}$$

2.1 Prediction and Allocation

Several researchers have approached the problem of performance prediction directly. The concept of **average parallelism** was used by Eager, Zahrojan and Lazowska[5] to analyze task graph performance and to determine performance bounds such as speedup and efficiency. Jaing, Bhuyan and Ghosal[10] and Jaing and Bhuyan[9] used **average parallelism** and the concept of **variation of parallelism** in performance analysis for multiprocessing task graphs. The **variation of parallelism** model evaluates a task graph by determining the degree of parallelism at successive stages (levels) and subsequently weighting the levels to determine piecewise execution times. The results are then combined to produce overall run times. This approach employs a queue based processor model and relies on the First Come First Served scheduling model to assign tasks to levels. Mak and Lundstrom[15] developed a somewhat more costly algorithm which employed similar processor and scheduling models. These methods have attempted to accurately predict task graph execution time at minimal computational cost, but rely on FCFS scheduling and have not addressed patterns of utilization directly.

The program allocation problem has been widely studied [11, 3, 8, 13, 6]. Graham[7] derived reasonable performance bounds for a class of allocation algorithms often referred to as list scheduling algorithms. List scheduling algorithms build schedules of tasks for each processor. A simple simulation of execution is carried out by tracking predicted task completion times[2]. During the simulation, the algorithms allocate tasks to processors by selecting an available task (one whose parents have completed) whenever there is an available processor. These algorithms are shown to produce results within a factor of 2 of optimal allocations. The first come first served heuristic (FCFS) leads to one of the simplest list scheduling algorithms where tasks are selected in first come first served order. The CPM allocation algorithm[11] prioritises tasks based on their **exit path lengths**[3] and has been shown to produce superior results (often within 5 percent of optimal). These algorithms address execution speed but largely ignore system utilization patterns.

Wilson and Gonzalez[20] have addressed the utilization aspect of allocation. They estimate the execution time, simply, based on the critical path length of the task graph. Their approach uses utilization profiles to characterize the utilization of processors over time. Their algorithm seeks to minimize the total number of processors used and if possible to reduce the maximum number of processors active at any time to equal the **average parallelism** of the graph. They accomplish this by shifting tasks at peak load times to execute later at lower load times whenever this can be done without violating precedence constraints. As an example, task **g** in Figure 2(b) could be delayed until after task **e** blunting the profile peak. When successful, this method produces allocations that approach 100 percent efficiency. The model, however, requires unit length execution times for all tasks and an adequate number of processors to achieve optimum run time.

2.2 Partitioning

A number of researchers have addressed the problems of partitioning multiprocessor systems into subsets (partitions) of processors to accommodate the allocation of multiple programs or program fragments. Partitioning problems in general are NP-Hard, however, polynomial time complexity algorithms are known for special cases [16]. Static approaches have been developed that restrict partitioning to a predetermined

[2]This method, therefore, incorporates time prediction.

[3]The exit path length of a task is the longest precedence related sequence of tasks from the task to the end of the graph.

size partition[14, 21] for each program. The required partition size is input to the partitioning algorithm. Static approaches have also been developed that allow the partitioning algorithm to select from a number of partition sizes for each program [17, 12, 16, 2, 19]. Partitioning algorithm input data indicates the run time of each program for each partition size in these algorithms.

Among these partitioning approaches are algorithms that consider independent programs only [12, 14, 2, 19, 21]. Alternately, precedence relationships between programs are considered in several approaches [17, 16]. Techniques similar to list scheduling allocation algorithms are often used to resolve precedence relationships. Using these techniques, a single partition of processors runs multiple programs one following the other. All of these approaches are appropriate for shared memory model multiprocessor systems[4].

Taken as a group, the common goal of these algorithms is to reduce total run time for the collection of partitions. The completion times of partitions, however, are not uniform. All of the partitioning algorithms may generate sets of partitions whose completion times vary by as much as 50 percent. The reassignment of processors after termination from partitions that terminate early to longer partitions is not considered.

These algorithms produce uniform partitions, each consisting of a fixed number of processors for the duration of the program. Processors within one partition may be unused for significant periods. While at the same time other partitions may be using all assigned processors and be capable of using more processors if they were available. The temporary reassignment of the unused processors to partitions that could use them is not considered.

2.3 Discussion

Previous research in performance prediction has seriously restricted the applicable allocation model and has not adequately addressed patterns of utilization. Allocation research has generally not addressed problems of utilization with the exception of [20] which severely restricts the problem domain.

In program allocation, increasing the number of processors increases speedup, but it also reduces efficiency. The concept of "number of processors," directly corresponds to the concept of partition size used in the partitioning algorithms discussed above. All the partitioning algorithms reviewed produce allocations based on uniform partition size. The efficiency and utilization pattern of individual partitions is not considered, although it is a potentially significant factor.

The goal of this paper is to discuss the potential for partitioning multiprocessor systems in a way that takes advantage of the potential for sharing underutilized processors between partitions. The pattern of processor utilization for a given allocation and multiprocessor configuration may vary significantly, however, with slight changes in system parameters. Because of this, sharing of processors between partitions may also significantly alter the pattern of processor utilization. We present algorithms that conceptually combine task shifting [20] with level analysis [10, 9]. Our allocation method enjoys the performance benefits of the CPM heuristic while minimizing the number of processors used and stabilizing fluctuations in utilization.

[4] Several algorithms address distributed memory models, at least to some extent[14, 16]

We demonstrate the application of this method to optimize the uniform partitions produced by [12] in Section 6. Our optimization is done by temporarily reassigning processors from one partition to another when this can improve overall run time.

3 Observed Utilization Patterns

Wilson and Gonzalez noted that for a task graph with uniform length tasks executed in optimal (critical path length) time, there are many potential allocations corresponding to different profiles. They also noted that the total number of processors used could often be reduced and efficiency increased by shifting task/processor assignments and task ordering. We observe that similar task shifting is produced by known allocation algorithms under more general circumstances. In particular, this occurs when CPM or FCFS allocation is used for graphs without uniform task length and with a suboptimal number of processors. Limiting the number of processors causes task execution to be shifted (delayed) until processors become available. For example, CPM allocation using 3 processors blunts the utilization peak in Figure 2 as described previously.

Variation in utilization patterns is observed when the number of processors varies. Figure 3 shows the utilization profile for the execution of a task graph using 5, 4 and 3 processor systems allocated using CPM scheduling. The shaded area of each profile indicates the total work carried out over time. The total shaded area of each profile is the same. The completion times using 5 and 4 processors are the same. The processor utilization (and corresponding allocations) are very similar during intervals T0-T1 and T2-T3. Significant task shifting and different utilization occurs in interval T1-T2 due to the variation in the number of processors. The peak in Figure 3(a) is flatter and the valley is partially filled in Figure 3(b). A fluctuation in utilization has occurred with decrease from 5 to 4 processors but there is no overall slowdown. We call this kind of fluctuation **flattening** (i.e. the 4 processor profile is **flattened** in this region). In anomalous situations there may actually be speedup in **flattened** regions.

Reducing the number of processors further (in Figure 3(c)) extends the execution time. The run time in Figure 3(b) during interval T1-T1(a) is approximately equal to the work **W** of that interval divided by the number of processors (4) while in Figure 3(c) it is approximately $\frac{W}{3}$. This accounts for the time extension. During the interval T1a-T2a **flattening** occurs. The time extension in T1-T1a is consistent with that reported in [10] for the execution time of levels where FCFS scheduling is used. The **flattening** effect occurs in CPM scheduling because tasks with shorter exit path lengths are postponed to accommodate critical path tasks (similar to [20]). **Flattening** occurs much less using FCFS scheduling, however, where only entrance path length is considered.

The effect of **flattening** on the profiles illustrated is local. Only the interval T1-T2a is affected. The remaining intervals may be shifted in time (i.e. T2a-T3) but are otherwise unchanged.

The relationships between these profiles has been observed to be typical of programs allocated to shared memory multiprocessors using the CPM allocation heuristic when the number of processors varies between profiles. In general, regions may **flatten** to some extent determined by graph precedence relationships

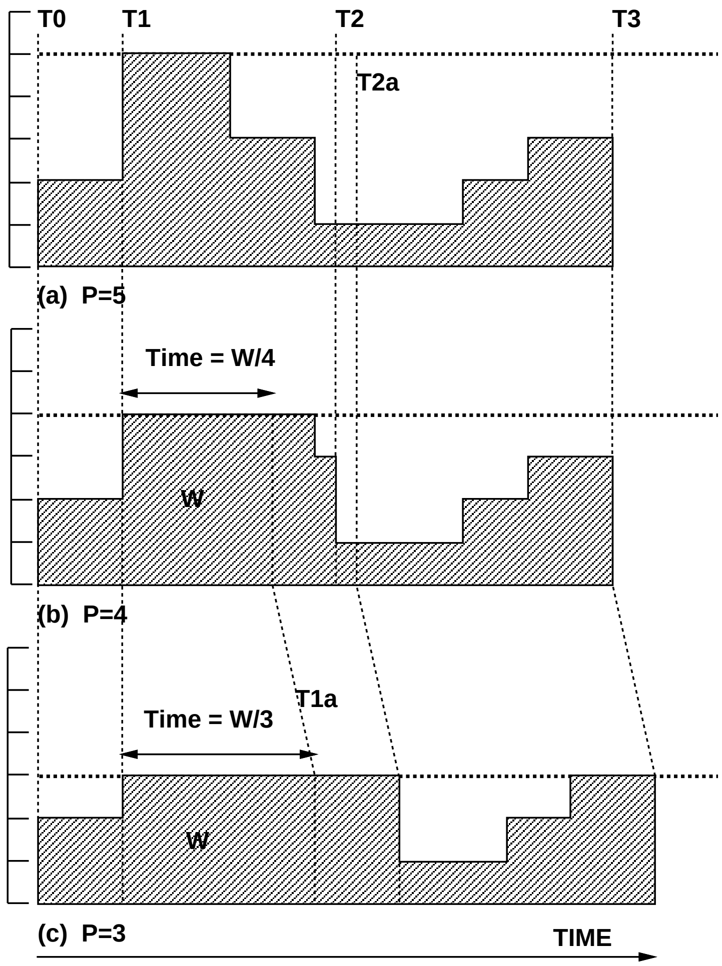

Figure 3: Processor Utilization Profiles

and beyond this the run time is extended. In profiles where all regions have been flattened, the run time extension of **flattened** segments may be approximated to a high degree of accuracy by dividing the flattened profile area by the number of processors (similar to [10, 9]). The simulation tests in Section 5 provide empirical validation of these observations.

4 Flattened Profile Generation and Allocation

In this section we discuss a program profiling/allocation method. Two algorithms are presented. First, a profile flattening algorithm that seeks to generate profiles with all segments flattened. Second, we present an allocation algorithm. The goal of this algorithm is to produce allocations that conform to the flattened profiles.

GEN_COMPOSITE generates flattened utilization profiles (see Figures 4 and 5). Input is a list of allocation profiles ordered by increasing number of processors. The algorithm iteratively compares pairs of profiles using procedure COMPARE. It begins with the fewest number of processors (2) and builds a flattened composite profile to be used for subsequent comparisons.

```
PROCEDURE: GEN_COMPOSITE
INPUT: L: A sequence of profiles ordered in
          ascending number of processors
FUNCTION: produce a composite profile
L1 = POP(L)
While L is not empty:
    L2 = POP(L)
    L1 = COMPARE(L1,L2)
Return L1
```

Figure 4: Generate Composite

```
PROCEDURE: COMPARE
INPUT:   LOWP,HIGHP: Utilization profiles
DATA:    SEG1,SEG2,END1,END2: Profile pointers
OUTPUT:   RES: result profile
FUNCTION:
    produce a composite profile by comparing
       LOWP with HIGHP
SEG1 = end segment of LOWP
SEG2 = end segment of HIGHP
1. While SEG1 is not at beginning of LOWP
1.a  While SEG1.nprocs == SEG2.nprocs
       set RES to match shorter of SEG1 and SEG2
       subtract shorter segment length from
          longer segment
       decrement shorter segment to next
          profile segment
1.b  If SEG1.nprocs > SEG2.nprocs then
          { flatten the peak in this interval }
       END1 = SEG1
       END2 = SEG2
       sweep SEG1 and SEG2 maintaining uniform
          time shift until
          work( SEG1, END1 ) == work( SEG2, END2 )
          set RES to match LOWP over
             interval (SEG1, END1)
1.c  else if SEG1.nprocs < SEG2.nprocs then
          { SEG2 is showing real speedup }
       sweep SEG1 and SEG2 maintaining uniform
          work progress { time shift will vary }
          until (SEG1.nprocs >= SEG2.nprocs)
          set RES to match HIGHP during this interval
Return RES
```

Figure 5: COMPARE

COMPARE compares pairs of profiles. It sweeps the profiles from back to front identifying regions which may be flattened and produces a composite profile with these regions flattened. Figure 6 illustrates this process for profiles (b) and (c) of Figure 3. In the profile with more total processors, **HIGHP**, the pattern of underutilization followed by overutilization is seen in intervals that may be flattened. An example of this is the interval Tb-Tc in Figure 6. The flattenable region extends from the beginning of underutilization until overutilization causes both profiles to sweep the same area. The new composite profile matches the profile with fewer total processors **LOWP** in these intervals. In the remaining intervals

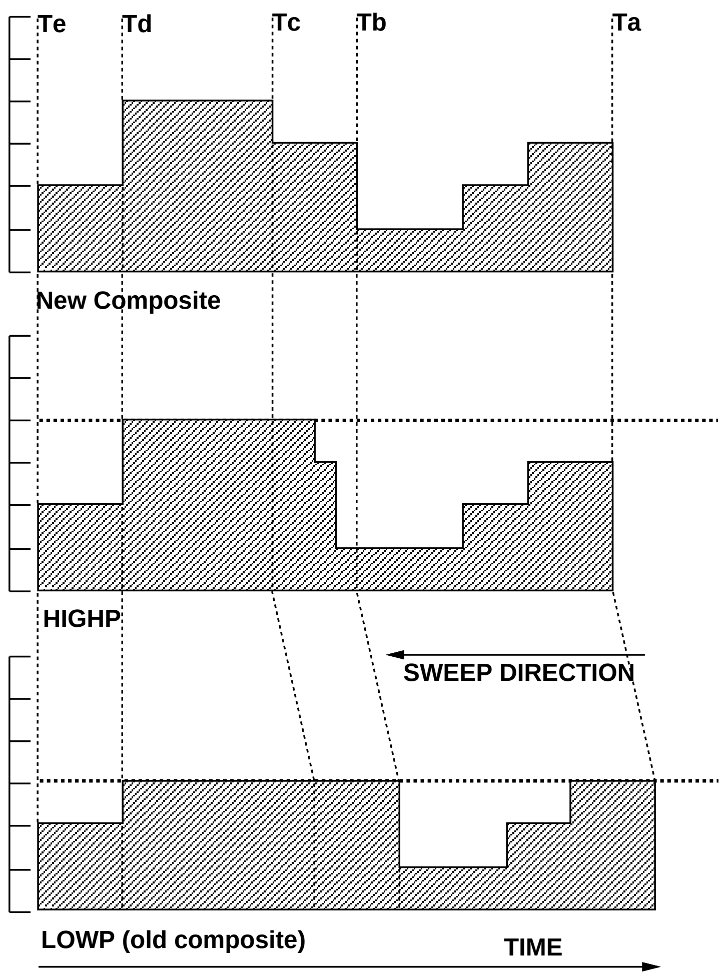

Figure 6: Profile Comparison

either profile **HIGHP** shows true speedup as in interval Tc-Td or both are the same as in Ta-Tb. The new composite profile matches profile **HIGHP** in these intervals.

4.1 Allocation Using Flattened Profiles

A modified CPM allocation algorithm has been developed that uses a flattened profile to regulate processor utilization. The modified algorithm produces allocations that conform to the profile. In addition to task execution times, the modified algorithm tracks the total work (**WORK**) completed during the simulated execution. When allocating a new task the profile is consulted using **WORK** to determine the profile segment(s) during which the task will execute. The function FEASIBLE determines whether the task can be allocated. FEASIBLE returns **FALSE** if executing the task will cause processor utilization to exceed $\lceil U \rceil$ for any segment. The algorithm is outlined in Figure 7.

$$FEASIBLE(TASK_i, procs\_needed) = \begin{cases} TRUE & \text{if the number of processors allowed by PRF for interval:} \\ & \text{(WORK, WORK + (procs\_needed * length\_of(}TASK_i\text{))) is} \\ & \text{greater than or equal to procs\_needed} \\ FALSE & \text{otherwise} \end{cases} \quad (5)$$

```
PROCEDURE: MODIFIED_CPM_ALLOCATE
INPUT:
    G: exit path length prioritized task graph
    PRF: composite profile generated by
        gen_composite
OUTPUT: A Schedule of tasks for each PE
FUNCTION: allocate task graph
Prioritize tasks by exit path length
assign task_0 to PE_0
insert completion event for task_0 into EQ
procs_used = 1
time = 0
WORK = 0
While( not empty( EVENT_Q ) )
1.a last_time = time
    time = next_event_time( EVENT_Q )
    WORK += procs_used * (last_time - time)
        { PROCESS TASK COMPLETIONS }
1.b while( next_event_time( EVENT_Q ) == time )
        E = pop( EVENT_Q )
        insert E.processor in PROCESSOR_Q
        procs_used --
        foreach child of E.task
            decrement arc_count of child
            if arc_count == 0 then insert child
                in TASK_Q
        { SCHEDULE TASKS }
1.c done = not empty( TASK_Q )
    while( not done )
        T = top(TASK_Q)
        done =
            not FEASIBLE( T, procs_used + 1 )) or
            empty( PROCESSOR_Q )
        if not done then
            P = pop(PROCESSOR_Q)
            assign T to P
            insert completion event for T in EVENT_Q
            procs_used ++
            pop(TASK_Q)
            done = not empty( TASK_Q )
```

Figure 7: ALLOCATION PROCEDURE

4.2 Performance Relationships in Flattened Profiles

The observations of Section 3 apply to profiles generated using the methods of this section. In particular, all segments of the composite profiles generated by GEN_COMPOSITE are **flattened** segments. Because of this, segment performance characteristics can be easily predicted. In addition, the effect of variations in the number of processors **P** available during segment execution can be predicted. For the purpose of dealing with variations in the number of processors, a third field **P** is added to each profile segment. **P** indicates the number of processors available to the program during the segment. We refer to the modified profiles as **partition profiles**. For a given segment **i** estimated performance characteristics are:

$$Processors\_used_i = \min(\lceil U_i \rceil, P_i) \tag{6}$$

$$Speedup_i = U_{actual,i} = \min(U_i, P_i) \tag{7}$$

$$Time_i = \frac{W_i}{U_{actual,i}} \tag{8}$$

$$Efficiency_i = \frac{U_{actual,i}}{P_i} \tag{9}$$

The estimated run time, work and efficiency for the entire program are:

$$Time_{total} = \sum_{i=1}^{maxseg} Time_i \tag{10}$$

$$W_{total} = \sum_{i=1}^{maxseg} W_i \tag{11}$$

$$Efficiency_{total} = \frac{\sum_{i=1}^{maxseg} Efficiency_i * Time_i}{Time_{total}} \tag{12}$$

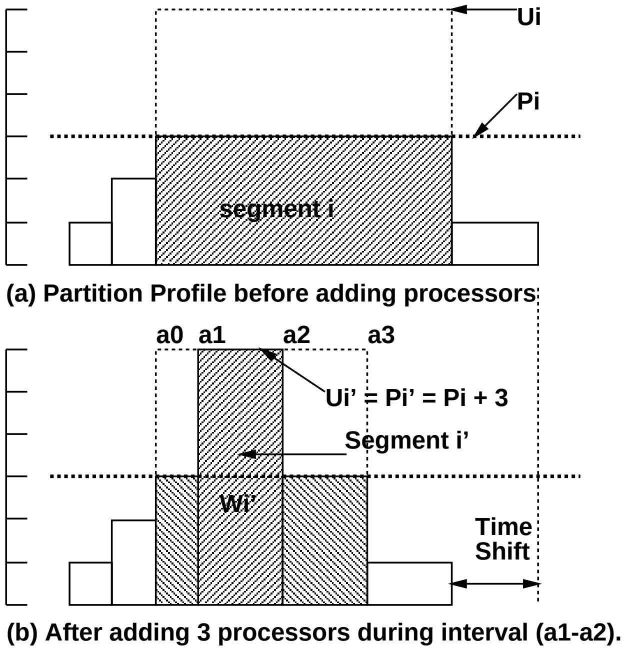

Figure 8: Adding Processors to a Segment

The effect of a change in the number of available processors **P** over a portion of a segment is estimated by subdividing the segment. Each subsegment has the same **U** value as the original and each has a portion of the work **W**. **P** values vary according to the change in number of available processors. Figure 8 shows the effect of adding 3 processors to a segment **i** for a period of time **a1 - a2**. When the segment is modified, preceding segments are unchanged. Subsequent segments are unchanged also except that they are shifted in time. For the modified subsegment i':

$$Time\_shift_{i'} = \frac{W_{i'}}{U_{actual,i'}} - \frac{W_{i'}}{U_{actual,i}} \tag{13}$$

5 Simulation Tests

Tests were conducted to determine the effectiveness of our allocation method. Over one thousand test graphs were constructed. For a given graph and multiprocessor configuration, P_{opt} is the minimum number of processors such that: $Time(P_{opt}) \leq 1.01 * Critical\_Path\_Length$. For each graph a set of allocations was produced with the number of processors $\mathbf{P}$ ranging from 2 to P_{opt}. Efficiency for an optimum run time allocation may vary from 0.0 to 1.0 depending on task graph characteristics. The graph set used, averaged approximately 66 percent efficiency (for CPM allocation) representing reasonably efficient task graphs.

The profiling/allocation algorithm was compared against the CPM allocation algorithm in several areas:

Run time: Run times of the allocated graphs produced by CPM using P_{opt} processors and by the profile driven allocator were compared. The average ratio of Profile driven allocation time to CPM allocation execution time was: 1.029

Resource Utilization: Average ratio of **resources allocated** to **resources used** was compared.

$$U\_ratio_{cpm} = \frac{P * Time_{total,cpm}}{W_{total}} \tag{14}$$

$$U\_ratio_{profiled} = \frac{\sum_{i=1}^{maxseg} processors\_used_i * Time_i}{W_{total}} \tag{15}$$

CPM	Profiled	CPM - Profile
1.545	1.126	0.419

Temporarily reassigning processors that are temporarily idle to partitions that could use them can improve resource utilization and decrease run time. The difference in resource utilization ratios represents the limit in resource utilization improvement that can be achieved by reassigning processors[5].

Run times were compared against predicted times for the profile/allocation method with $1 < P_i < P_{opt}$ processors. The prediction **Error** was calculated as:

$$Error = \left| \frac{simulated\ run\ time}{Time_{total}} - 1.0 \right| \tag{16}$$

[5] Very short reassignment intervals can rarely be used. Segments shorter than 0.5 percent of program length were merged for all tests.

When more than P_{opt} processors are available the run time is stable and within 1 percent of predicted time for P_{opt} processors. When one processor is used, predicted time matches actual time.

$P_{opt} - P$ Processors		
P	Test Cases	Avg. Error
1	1110	0.0082
2	1082	0.0097
3	805	0.0159
4	411	0.0249
5	50	0.0237
6	19	0.0251
7	5	0.0164

6 Application: Partition Optimization

In this section we demonstrate the use of utilization profiles to optimize the run time of multiprocessor partitions generated by the partitioning algorithms discussed in Section 2.2. All of these algorithms generate uniform sized partitions. Our algorithm optimizes partitions by temporarily reassigning unused processors from one partition to another that can use them.

Reassignment will improve system run time whenever a processor can be reassigned to the partition with the longest running time. The feasibility of processor reassignment is dependent on multiprocessor parameters and the duration of the reassignment. We refer to the minimum feasible reassignment interval as ϵ. In the limit as ϵ approaches zero, processor reassignment approaches the **processor sharing** discipline, however, in practice, ϵ less than the average task length is impractical.

6.1 Reassignment

Procedure REASSIGN optimizes a set of parallel partitions using partition profiles. REASSIGN is outlined in Figure 9. The algorithm uses an event queue to traverse the segments of all partitions simultaneously. Event times are the predicted times at segment boundaries. During the interval between each two events (segment boundaries) the algorithm assigns available processors to **needy** partitions that have higher U than P values. A greedy heuristic is used to prioritize the **needy** partitions based on longest completion time. The algorithm assigns the maximum possible number of available processors to the highest priority **needy** partition. Figure 10 illustrates reassignment for (**Partition 1:Segment i**) of a 2 partition system. For the given partitions the procedure will reassign two processors from **Partition 0** to **Partition 1** for the interval **(E.time, Top(Event_Q).time)**. The modified **Partition 1** is shown in Figure 10(b). Reassigning processors shifts the completion time for the partition segment and requires modifying **NEEDY_Q** and **EVENT_Q** entries and may require splitting the segment for the modified partition (as shown).

6.2 Allocation

The allocation method uses a set of communicating processes in simulated time. There is one allocation process for each partition and a central **processor_manager** process (see Figure 11). A global request

```
PROCEDURE: REASSIGN
INPUT: PART: set of partition profiles
       ε: minimum interval allowed
DATA:
    NEEDY_Q: priority queue of
        (index,segment_num,procs_needed)
        with priority = partition completion time
    EVENT_Q: priority queue of
        (partition_num, segment_num)
        with priority = segment completion time
FUNCTION:
1.a foreach partition i
        insert (i,0) in EVENT_Q
        if( U_{i,0} > P_{i,0} ) insert (i, 0, U_{i,0} - P_{i,0}) in NEEDY_Q
        if( U_{i,0} < P_{i,0} ) procs_avail += U_{i,0} - P_{i,0}
2.a while( not empty( EVENT_Q ) )
        E = POP( EVENT_Q )
        insert (E.i, E.segment+1) in EVENT_Q
        adjust NEEDY_Q entry and procs_avail for E
2.b     if( ( TIME( TOP( EVENT_Q )) - TIME( E ) ) > ε )
            while( not empty( NEEDY_Q ) and
                    (procs_avail > 0) )
                P = POP( NEEDY_Q )
                cnt = min( procs_avail, p.procs_needed )
                temporarily reassign cnt processors to PART(p.i)
                adjust PART(p.i), NEEDY_Q and
                    EVENT_Q as needed
                procs_avail -= cnt
```

Figure 9: REASSIGNMENT PROCEDURE

queue and global processor queue are used for communication[6]. The allocation processes use a modified version of the algorithm in Figure 7. The procedure is modified to request needed processors and relinquish unused processors as specified by the profile via the processor manager. The modification is accomplished by adding the program fragment in Figure 12 before position **1.a** in Figure 7.

6.3 Evaluation

The reassignment potential for a given set of partitions depends on task graph and multiprocessor parameters. In this section we present an analysis of the probability that the reassignment algorithm can improve performance and show experimental test results.

The potential for reassignment from (**Partition i:Segment a**) to (**Partition j:Segment b**) exists whenever $U_{i,a} < P_{i,a}$ and $U_{j,b} > P_{j,b}$. The probability $\mathbf{p_i}$ that no processors are available on a **k** processor partition **i** at a given point in time is equal to the average utilization of k^{th} processor of that partition. The probability that no processors may be reassigned to partition **j** of an **M** partition system is the product of probabilities of the remaining partitions. The probability q_j that at least one processor is available for

[6] Queue operation POP(Q) blocks when Q is empty.

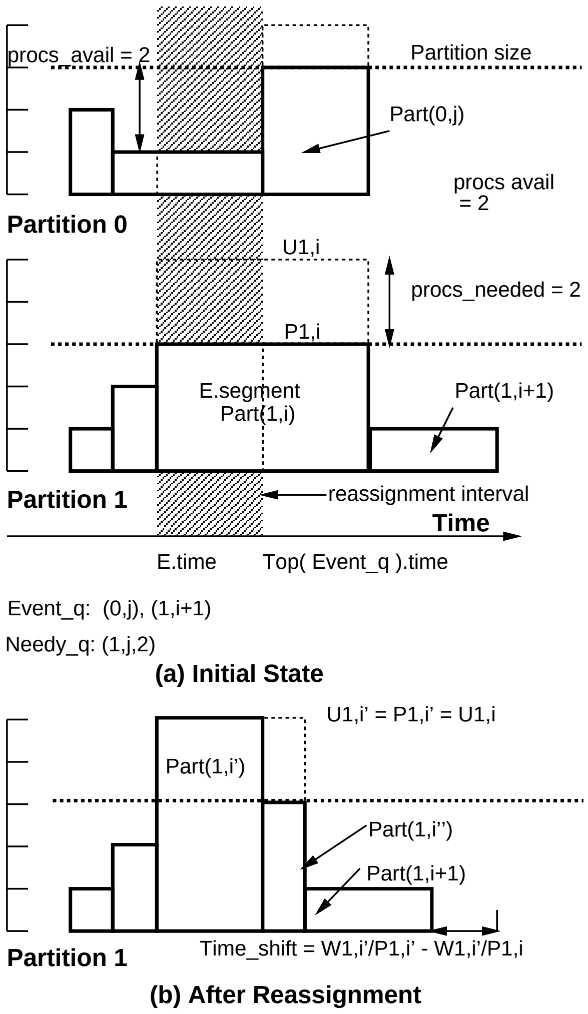

Figure 10: REASSIGN Procedure State Relationships

PROCEDURE: PROCESSOR_MANAGER
while TRUE
    request = POP(**global_request_q**)
    processor = POP(**global_processor_q**)
    insert **processor** in PROCESSOR_Q of **request.process**
    wake up **request.process**

Figure 11: Processor Manager

reassignment at a given point in time, then, is

$$q_j = 1 - \prod_{i=1}^{M} p_i, \ where\ i \neq j \tag{17}$$

when no partitions have terminated. When the shortest partition terminates T_{min} the probability becomes 1.

The expected percentage of the time that at least one processor is available to the longest partition **j**

```
sleep_until( next_event_time( EVENT_Q ) )
        { May wake up early with extra processors in queue }
procs_avail = procs_used + size_of( PROCESSOR_Q )
if( procs_avail > old_procs_avail )
    procs_requested − = (procs_avail - old_procs_avail)
        { relinquish processors }
while((procs_allowed( PRF ) < (procs_avail + procs_requested) )
        and (size_of( PROCESSOR_Q ) > 0))
    P = pop( PROCESSOR_Q )
    procs_avail −−
    insert P in global_processor_q
        { request processors }
while( procs_allowed( PRF ) > (procs_avail + procs_requested ))
    insert request in global_request_q
    procs_requested++
old_procs_avail = procs_avail
```

Figure 12: Allocation Procedure Modifications

is equal to the probability Q_j that one is available at any point.

$$Q_j = 1 - \frac{T_{min}}{Time_{total,j}} \left(\prod_{i=1}^{M} p_i \right), \ where\ i \neq j \tag{18}$$

Since p_i is at most 1, Q_j increases with increasing number of partitions and decreasing p_i values. Optimizing shorter partitions also decreases the average p_i value and increases Q_j.

6.3.1 Experimental Results

Simulation tests were used to evaluate the performance of the profile optimization method. Sets of 2 to 5 random graphs were used and initial multiprocessor partitions were generated using the algorithm of Krishnamurti and Ma [12]. The corresponding system speedups S_{km} and run times T_{km} were calculated. The REASSIGN procedure was then used to produce optimized partition profiles[7]. The optimized partition profiles were input to the modified allocation procedure which produced optimized allocations. The optimized system speedups S_{op} and run times T_{op} were calculated.

Determining the maximum possible system speedup S_{max} and minimum possible system run time T_{min} for an **N** processor system are NP-Hard problems. The amount of speedup and run time improvement possible by reassignment is dependent on the relative optimality of the initial partitions. In Figure 13, S'_{max} is an upper bound on the maximum possible speedup S_{max} .

$$S'_{max} \ = \ min \left(N, \frac{T_1}{T_{lim}} \right) \ \geq \ S_{max}, \tag{19}$$

where T_1 is the time for the entire system to run on a uniprocessor and T_{lim} is the maximum time for any one partition to run if given all N processors.

$$T_{lim} = \max_i (T_i(N)) \tag{20}$$

[7]using ϵ = average task size.

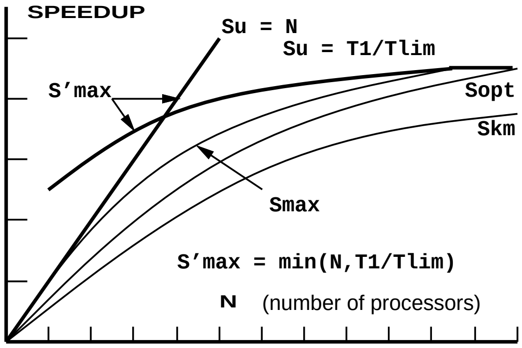

Figure 13: Speedup Graph

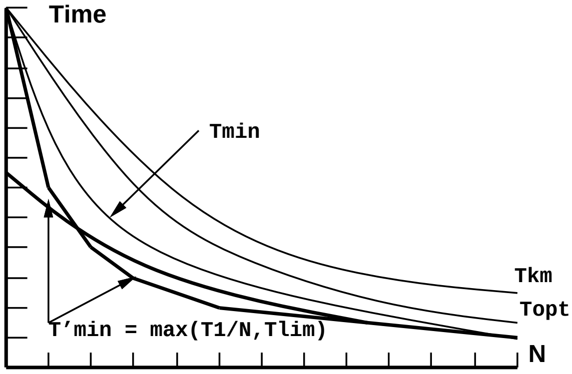

Figure 14: Runtime Graph

In Figure 14, T'_{min} is a lower bound on the minimum run time T_{min}.

$$T'_{min} \;=\; \max\left(\frac{T_1}{N}, T_{lim}\right) \;\leq\; T_{min} \tag{21}$$

Lower bounds on the achieved percentage of the maximum possible improvement in speedup and run time are:

$$S'_{imp} \;=\; \frac{S_{op} - S_{km}}{S'_{max} - S_{km}} \;\leq\; S_{imp} \tag{22}$$

$$T'_{imp} \;=\; \frac{T_{km} - T_{op}}{T_{km} - T'_{min}} \;\leq\; T_{imp} \tag{23}$$

Improvements were correlated against S'_{avail}, an upper bound on S_{avail}, the percentage of total possible speedup improvement still unachieved after initial partitioning.

$$S'_{avail} \;=\; 1 - \frac{S_{km}}{S'_{max}} \;\geq\; S_{avail} \tag{24}$$

$S'_{initial}$ a lower bound on the initial percentage of maximum possible speedup achieved

$$S'_{initial} = 1 - S'_{final} = \frac{S_{km}}{S'_{max}} \;\leq\; S_{initial} \tag{25}$$

$S'_{initial}$	S'_{avail}	Cases	T'_{imp}	S'_{imp}	S'_{final}
0.900	0.10	21	0.47	0.48	0.948
0.850	0.15	68	0.47	0.50	0.925
0.800	0.20	116	0.51	0.55	0.910
0.750	0.25	98	0.56	0.61	0.903
0.700	0.30	81	0.70	0.74	0.922
0.650	0.35	41	0.77	0.83	0.940
0.600	0.40	21	0.77	0.84	0.936
0.550	0.45	3	0.79	0.87	0.941
0.500	0.50	3	0.87	0.93	0.965

Figure 15: Performance Improvement

and S'_{final}, a lower bound on the final percentage of maximum possible speedup achieved S_{final}

$$S'_{final} \;=\; 1 - S'_{avail}(1 - S'_{imp}) \;\leq\; S_{final} \tag{26}$$

are also given.

Results are given in Figure 15. Experiments revealed that when the upper bound on possible improvement S'_{avail} was very low, S'_{imp} and T'_{imp} tended to be relatively lower. The 1 to 3 percent inaccuracy of the profile time estimation method becomes a significant factor in the low range, and makes the results unstable when S'_{avail} is below 0.05.

S'_{imp} tended to increase rapidly, however, as S_{avail} increased. The method is highly successful when the potential improvement is more significant. Because of this, combining partition optimization with the initial partitioning algorithm results in final speedups that average above 90 percent of optimal.

7 Conclusions

In this paper we have studied the relationship between program allocation and multiprocessor partitioning. We have presented a new approach to allocation using utilization profiles. This approach effectively minimizes program execution time and addresses several concerns of importance to multiprocessor partitioning that have not been well addressed previously:

1. Minimize the total number of processors used.
2. Characterize variation in processor requirements over the lifetime of a program.
3. Accurately predict the impact on run time of variation in the number of processors available at any point in program execution.
4. Minimize fluctuations in processor requirements to facilitate efficient dynamic reassignment of processors between partitions on a partitionable multiprocessor.

We have also presented algorithms for the application of the allocation method to the problem of optimizing parallel multiprocessor partitions that have shown significant improvement in performance over existing methods. Analysis of this method shows that the expected performance increases with increases in the number of partitions and with degradation in initial partition utilization. Test results using this method show average performance consistently above 90 percent of maximum possible speedup and show average improvement consistently increases with decreasing initial partition performance.

References

[1] Ackerman, W. B. Data Flow Languages. *Computer* (February 1982), 15–25.

[2] Belkhale, K. P., and Banerjee, P. Approximate Algorithms for the Partionable Independent Task Scheduling Problem. In *Proceedings of the 1990 International Conference on Parallel Processing* (1990), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. I–72 – I–75.

[3] Campbell, M. L. Static Allocation for a Data Flow Multiprocessor. In *Proceedings of the 1985 International Conference on Parallel Processing* (University Park, Pennsylvania, 1985), IEEE International Conference on Parallel Processing, The Pennsylvania State University Press, pp. 511–517.

[4] Davis, A. L., and Keller, R. M. Data Flow Program Graphs. *Computer* (February 1982), 26–41.

[5] Eager, K. L., Zahorjan, J., and Lazowska, E. D. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers C-38*, 3 (March 1989), 408–423.

[6] Evans, J. D., and Kessler, R. R. A Communication-Ordered Task Graph Allocation Algorithm. Submitted to: IEEE Transactions on Parallel and Distributed Systems, 1992.

[7] Graham, R. L. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics 17*, 2 (March 1969), 416–429.

[8] Ho, L. Y., and Irani, K. B. An Algorithm For Processor Allocation in a Dataflow Multiprocessing Environment. In *Proceedings of the 1983 International Conference on Parallel Processing* (1983), IEEE International Conference on Parallel Processing, The Pennsylvania State University Press, pp. 338–340.

[9] Jiang, H., and Bhuyan, L. N. Performance Analysis of Layered Task Graphs. In *Proceedings of the 1991 International Conference on Parallel Processing* (1991), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. III–275 – III–279.

[10] Jiang, H., Bhuyan, L. N., and Ghosal, D. Approximate Analysis of Multiprocessing Task Graphs. In *Proceedings of the 1990 International Conference on Parallel Processing* (1990), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. III–228 – III–235.

[11] Kaufman, M. T. An Almost-Optimal Algorithm for the Assembly Line Scheduling Problem. *IEEE Transactions on Computers C-23*, 11 (November 1974), 1169–1174.

[12] Krishnamurti, R., and Ma, E. The Processor Partitioning Problem in Special-Purpose Partitionable Systems. In *Proceedings of the 1988 International Conference on Parallel Processing* (1988), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. 434–443. Vol. 1.

[13] Lee, B., Hurson, A. R., and Feng, T. Y. A Vertically Layered Allocation Scheme for Data Flow Systems. *Journal of Parallel and Distributed Computing 11*, 4 (November 1991), 175–187.

[14] Li, K., and Cheng, K. H. Job Scheduling in Partitionable Mesh Connected Systems. In *Proceedings of the 1989 International Conference on Parallel Processing* (1989), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. II–65 – II–72.

[15] Mak, V. W., and Lundstrom, S. F. Predicting Performance of Parallel Computations. *IEEE Transactions on Parallel and Distributed Systems 1*, 3 (July 1990), 257–269.

[16] Narahari, B., and Choi, H.-A. Allocating Partitions to Task Precedence Graphs. In *Proceedings of the 1991 International Conference on Parallel Processing* (1991), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. I–621 – I–624.

[17] Polychronopoulos, C. D., and Banerjee, U. Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds. *IEEE Transactions on Computers C-36*, 4 (April 1987), 410–420.

[18] Preiss, B. R., and Hamacher, V. C. Semi-Static Dataflow. In *Proceedings of the 1988 International Conference on Parallel Processing* (1988), IEEE International Conference on Parallel Processing, The Pennsylvania State University Press, pp. 127–139.

[19] Turek, J., Wolf, J. L., Pattipati, K. R., and Yu, P. S. Scheduling Parallelizable Tasks: Putting it All on the Shelf. *Performance Evaluation Review 20*, 1 (June 1992), 225–236.

[20] Wilson, L. F., and Gonzalez, M. J. Manipulation of Parallel Algorithms to Improve Performance. In *Proceedings of the 1991 International Conference on Parallel Processing* (1991), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. I–119 – I–122.

[21] Zhu, Y., and Ahuja, M. Preemptive Job Scheduling on a Hypercube. In *Proceedings of the 1990 International Conference on Parallel Processing* (1990), International Conference on Parallel Processing, The Pennsylvania State University Press, pp. I–301 – I–304.