NASA
Technical
Paper
3323

June 1993

Performance Analysis of a Large-Grain Dataflow Scheduling Paradigm

Steven D. Young
and Robert W. Wills

NASA

NASA Technical Paper 3323

1993

Performance Analysis of a Large-Grain Dataflow Scheduling Paradigm

Steven D. Young
and Robert W. Wills
Langley Research Center
Hampton, Virginia

National Aeronautics and Space Administration
Office of Management
Scientific and Technical Information Program

Abstract

This paper describes and analyzes a paradigm for scheduling computations on a network of multiprocessors using large-grain dataflow scheduling at run time. The computations to be scheduled must follow a static flow graph, while the schedule itself will be dynamic (i.e., determined at run time). Many applications characterized by static flow exist, and they include real-time control and digital signal processing. With the advent of computer-aided software engineering (CASE) tools for capturing software designs in dataflow-like structures, macro-dataflow scheduling becomes increasingly attractive, if not necessary. For parallel implementations, using the macro-dataflow method allows the scheduling to be insulated from the application designer and enables the maximum utilization of available resources. Further, by allowing multitasking, processor utilizations can approach 100 percent while they maintain maximum speedup. Extensive simulation studies are performed on 4-, 8-, and 16-processor architectures that reflect the effects of communication delays, scheduling delays, algorithm class, and multitasking on performance and speedup gains.

1. Introduction

1.1. Background

Dataflow methods have been criticized because of extraordinary scheduling and communication overhead, difficulties in specifying (or converting to) low-level implementations, memory management obstacles, synchronization issues, and other aspects of hardware and software complexity (refs. 1 to 7). These problems can be significant drawbacks when the dataflow is implemented at the instruction level (refs. 4, 5, 8, and 9). By spreading the overhead of the dataflow over computations on the order of hundreds or thousands of instructions or more, the dataflow complexity (both scheduling and communication) is no longer a limiting factor for performance. We refer to dataflow at this level as macro-dataflow or large-grain dataflow (refs. 1, 3, 4, 6, 7, 10, and 11). In addition to the benefit of reducing the effect of overhead, macro-dataflow provides a more natural means to describe applications aimed at both parallel and serial implementations. The use of computer-aided software engineering (CASE) tools (refs. 12 to 14) for software design has become popular as a mechanism for designing software systems according to a structured method. This method entails creating functional decompositions in the form of hierarchical levels of dataflow diagrams. Hence, as designers begin to use these tools exclusively, it becomes desirable for multiprocessing platforms to effectively and efficiently distribute the application onto the available resources. Because of the hierarchical dataflow diagram structure, scheduling using macro-dataflow techniques becomes straightforward.

One drawback of dataflow-based systems in the past has been the problem of converting software designs and implementations to a dataflow structure. In fact, tools have been developed to automatically generate dataflow diagrams from source code (refs. 7, 10, and 15). For example, in reference 7, extensions to C++ permit automatic generation of the dataflow graphs at run time (with additional overhead). Therefore, the question remains, will the designer, compiler, or run-time environment create a more effective functional decomposition that can be used to exploit parallelism? We assume that if the programmer provides the breakdown as a natural consequence of the design method, we should expect a more efficient use of resources (refs. 1 and 3).

The primary idea of dataflow scheduling (not computation) is to put all jobs that are ready to be scheduled in an execution queue (EQ), and when a processor becomes available to execute these jobs, they are removed from the queue and sent to the available processor. As processors finish jobs, they notify the scheduler that they are available to perform subsequent work. Of course, as with any queuing system, a queuing discipline must be chosen to determine the order in which jobs in the queue are dispersed (e.g., whether the order is "first-come, first-served," "last-come, first-served," or priority based).

Most micro-dataflow EQ's consist of an instruction and some context information (i.e., a tag). Tags allow dynamic schedules. A macro-dataflow scheduler consists of a very similar structure, but a subroutine location and the size are included as opposed

to a single instruction. This method assumes that all processors have a local copy of the application. If local memory size is a problem, other implementations can be used, but because this is a high-level analysis, we will leave the details of the implementation to subsequent papers. This scheduling paradigm makes the amount of work for control overhead negligible when compared with the amount of work controlled. References 11, 16, and 17 describe an example of an implementation of such a paradigm using marked Petri-nets.

Another drawback of either a micro- or a macro-dataflow is that enough parallelization must be present in a computation to keep the EQ from becoming empty so that the processors remain busy at all times. For a single user on a multiprocessor network, it is a difficult task to keep all the processors busy because most computations involve a significant serial fraction (i.e., a percentage of the computations that must be done serially). Introducing multitasking to the macro-dataflow environment provides completely independent threads of work that may be done in parallel. A macro-dataflow system that runs several tasks may be viewed as a system running a standard dataflow task, which has several times the ability to be parallelized. Therefore, by adding tasks to the system, we can increase the percentage of the total work that can be parallelized (i.e., the parallel fraction). This addition of tasks makes it less likely that the EQ will become empty.

Further, another problem can occur using multitasking when some tasks have more parallelization than do others. If one task has a larger number of processes which can be scheduled in parallel, these processes tend to dominate the EQ, while tasks that have a smaller number of processes which can be parallelized are left waiting in a long line to be scheduled. To avoid this scenario, the scheduler must allocate processors to tasks with a probability that is at least proportional to the amount of inherent parallelism in the tasks to be executed.

One final point is that the statistic of 100-percent processor utilization can be misleading. Although a processor may have work to do at all times, some parts of this workload will involve waiting for input/output (I/O) operations to be completed. One solution to this problem is to schedule more than one process to each processor (i.e., called multiprogramming). The processor then can perform a context switch, during an I/O operation, to a subsequent job. When the I/O operation is completed, the processor switches back to the original job. Although multiprogramming adds a small amount of overhead at the local processor (i.e., switch time), it is apparent that the effective utilization of the processor will increase.

1.2. Problem Statement

The purpose of this paper is threefold: (1) to conduct a performance analysis of the macro-dataflow scheduling method just described using common workload structures as benchmarks, (2) to show how multitasking can be used to increase processor utilization, and (3) to describe how CASE tools naturally lend themselves to this type of scheduling.

Specifically, this paper discusses three suites of experiments. The first set uses the fork-join problem to determine the effects of problem size on achievable speedup and processor utilization. Speedup is defined as the ratio of the latency on a single processor system to the latency on a multiprocessor system. Processor utilization is the fraction of time that a processor is busy doing useful work. The second set of experiments also uses the fork-join problem to reveal the effects of adding a multitasking capability to the scheduler. Finally, the third set of experiments reveals the effect of various scheduling and communication overheads on the performance of the scheduler for three classes of problems: the fork-join, the binary tree, and the diamond-shaped graph.

Section 2 describes the approach used to achieve the prescribed goals. Section 3 describes in detail the experiments performed. Section 4 presents the simulation results obtained and the analysis of them. Finally, section 5 presents the conclusions that can be drawn.

2. Approach

2.1. Macro-Dataflow Scheduler Model

From a computational view, macro-dataflow is purely data driven (ref. 7). However, viewed as a scheduling mechanism, macro-dataflow is driven both by the data and by the processor availability. By forcing processor availability to drive the schedule, we can optimize both speedup and processor utilization.

For this analysis, the model that we evaluate consists of two "first-come, first-served" queues that are responsible for distributing work across an ideal network with no communication delays. (However, in section 4.4, we will show the effect of adding communication delays.) The first queue contains those jobs (processes) which are ready to be executed, and the second queue contains identifiers representing those processors that are available to do work. Both queues are updated as the processors complete the jobs and become idle. As long as neither queue becomes empty, the network operates at 100-percent

efficiency. As described earlier, this approach models a multitasking capability and reduces the probability of having an empty queue.

For a more accurate model, we will study the effects of communication and scheduling delays on system performance. This analysis will aid us to evaluate alternate methods of scheduling work, routing messages, and implementing queues.

As stated in section 1.1, I/O needs to be addressed for further utilization improvement. Several independent jobs could be given to a single processor and, in effect, multiprogrammed to keep all processors operating at full speed. The model for I/O and the techniques to multiprogram tasks have not yet been created.

A high-level system design and simulation environment was used to effectively analyze the scheduling paradigm. This environment was provided by the Architecture Design and Assessment System (ADAS) toolset (ref. 18). This toolset enables a high-level description of both the architecture and the application workloads, the discrete-event simulation of system activity, and the acquisition of performance-related data to facilitate the analysis.

2.2. Algorithm Description

To evaluate a wide range of applications, a concise method of workload description was necessary. Because work must be partitioned at the subroutine (process) level for a macro-dataflow scheduling paradigm, a simple dataflow diagram suffices (e.g., as shown in fig. 1). This diagram represents the processes (P0, ..., P6) to be executed and the order in which they must be executed. Other diagrams can be generated to represent specific problems (e.g., fork-join or binary tree). These dataflow diagrams then can be converted to an ASCII representation with the following format (in which -1 is a terminator):

```
Number-of-tasks:  1
Number-of-processes:  7
P0-duration:  0.574
P0-sends-to:  1 2 3 -1
P1-duration:  0.983
P1-sends-to:  4 -1
P2-duration:  0.317
P2-sends-to:  4 -1
P3-duration:  4.583
P3-sends-to:  5 -1
P4-duration:  2.441
P4-sends-to:  6 -1
P5-duration:  7.092
P5-sends-to:  6 -1
P6-duration:  0.139
P6-sends-to:  -1
```

This conversion is done for the set of tasks which comprises a single workload. The resulting ASCII file contains both precedence information and complexity approximations for each process within each task. The code that is responsible for simulating the activity of the system reads this file at startup to become aware of the work that it must perform.

The workload description (either the dataflow diagrams or the ASCII file representation) can be generated from the source code itself or as mentioned in section 1, from a CASE tool specification of the application. For simulation purposes, approximate process durations can be calculated from either a single-processor implementation or the instruction counts along with the benchmark instruction times. Because this is a high-level analysis of the paradigm, precision (or accuracy) is not a real issue at this point. In fact, by varying process durations, we can show the sensitivity of the overall performance to different granularities.

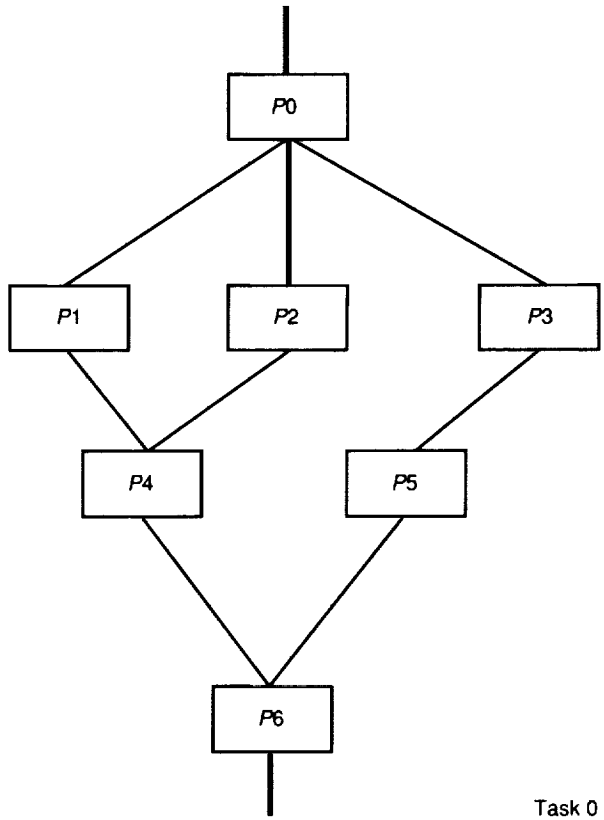


Figure 1. Sample dataflow diagram for single task.

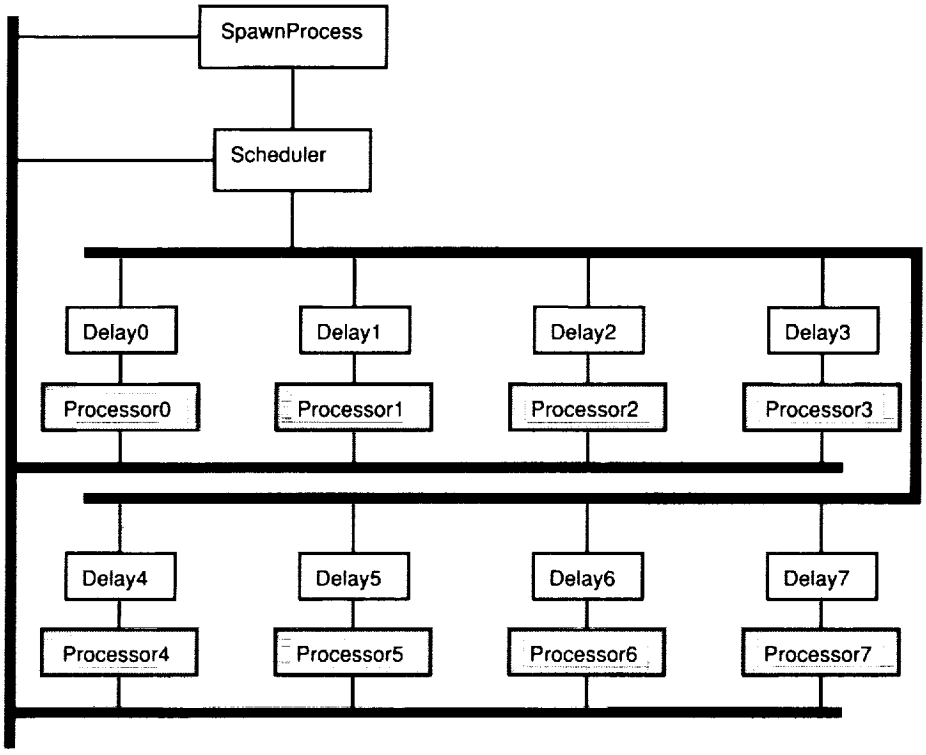


Figure 2. ADAS model of macro-dataflow scheduler (with eight processors).

2.3. ADAS Architecture Model

As stated, the queue operation will be simulated using the ADAS toolset. This toolset was first developed at the Research Triangle Institute (RTI) and is now marketed by Cadre Technologies, Incorporated. The version used for this analysis runs on a Sun work station. The ADAS toolset allows a system designer to evaluate alternative architectures with respect to system performance and behavior.

An ADAS model consists of nodes, arcs, and tokens. Tokens are abstract entities (which can contain data) which traverse through a dataflow diagram and stimulate activity during the simulation. As tokens pass through nodes in the dataflow diagram, some function is performed, and the token is either absorbed or sent out along one of the arcs emanating from the node.

This ADAS toolset requires a dataflow graph model that represents the operation of the macro-dataflow scheduler and its interaction with the multiprocessor that it controls (fig. 2). To allow behavioral simulation of an ADAS model, a functional description of each node in the graph is required. This description can be written in either the C or Ada languages to represent how data (tokens) propagate throughout the model. Once this code is written and its behavior has been verified, simulation of the model can be performed. The following section describes the functionality of each node in the model.

2.3.1. SpawnProcess node. At initialization, the code associated with the SpawnProcess node reads the ASCII file that represents the work to be done and sends the initial process of every task to the scheduler. These tokens contain a task identifier, a process identifier, and the process duration. After initialization, this node waits for input from the processors to determine when a process has been completed. Once a process is completed, this node sends any subsequent processes that only depend on that process to the scheduler. This scenario continues until all tasks have been completed, at which time no outputs are generated and the simulation terminates.

2.3.2. Scheduler node. The scheduler node contains two queues. The JobList queue contains processes, sent from the SpawnProcess node, which are ready to be scheduled. The PEready queue contains the processor identifiers of all processors that are idle at the current time. (Initially, all

processors are idle.) As long as both queues are not empty, a process is removed from the JobList queue and sent to the processor that is identified at the top of the PEready queue.

2.3.3. Delay nodes. The delay nodes simulate the communication delay associated with transmitting a process description to a processor. These nodes simply hold the processes for an amount of time that is proportional to the size of the process and the communication bandwidth. By varying the communication bandwidth attribute, we can determine the effect of alternate interprocessor connection topologies and routing methods on performance.

2.3.4. Processor nodes. The processor nodes simulate the delay associated with actually executing the process. These nodes hold the processor for an amount of time that is proportional to the duration of the process which is specified in the token. The duration value has been calculated for a specific processor. To observe the effect of selecting different processors, we can scale this number by a factor that represents the change in speed of the alternate processor. After completing the execution of the process, the SpawnProcess node is notified. Also, the PEready queue in the scheduler node is updated to reflect a new idle processor.

2.4. Simulation and Analysis

After describing the functionality of the nodes using the C language, the CSIM facility of the ADAS toolset can be used to simulate the execution of the scheduler-multiprocessor model. Input variables for each simulation are both workload related (e.g., type, size, granularity, structure, and iteration count) and architecture related (e.g., scheduling overhead, interprocessor communication overhead, and processor throughput). During and after the simulations, the following indices are recorded: task latencies, processor utilizations, network (e.g., bus) utilization, average queue sizes, and speedup gains.

During the analysis phase, the results of the simulation studies were plotted to show the effects of changes in input parameters as well as in workload type. The specific phenomena that we wish to observe include the effect of communication and scheduling delays on speedup, the maximum speedup achievable using this large-grain dataflow scheduling paradigm, the processor utilization as a function of algorithm size, and the benefit of adding multitasking and multiprogramming capabilities to the scheduler.

Experiments

For the experiments performed to date, three classes of workloads have been used: fork-join algorithms (i.e., problems), binary tree algorithms, and diamond-shaped algorithms (figs. 3 to 5). These applications are diverse in structure and, hence, place different stresses on the scheduling process. Also, they are widely accepted benchmarks for multiprocessing systems because they represent a large segment of computationally intensive problem sets.

The first experiments use the fork-join class of algorithms (e.g., fig. 3) to show the effects of problem size and multitasking. Process durations are chosen from a normal distribution. The fork and the join (top and bottom, respectively) process durations have a mean of 0.5 and a standard deviation of 20 percent; the durations of the processes between the fork and the join have a mean of 4.0 and a standard deviation of 25 percent. Communication and scheduling delays are set to zero. The workload is executed for 100 iterations, and the results are averaged.

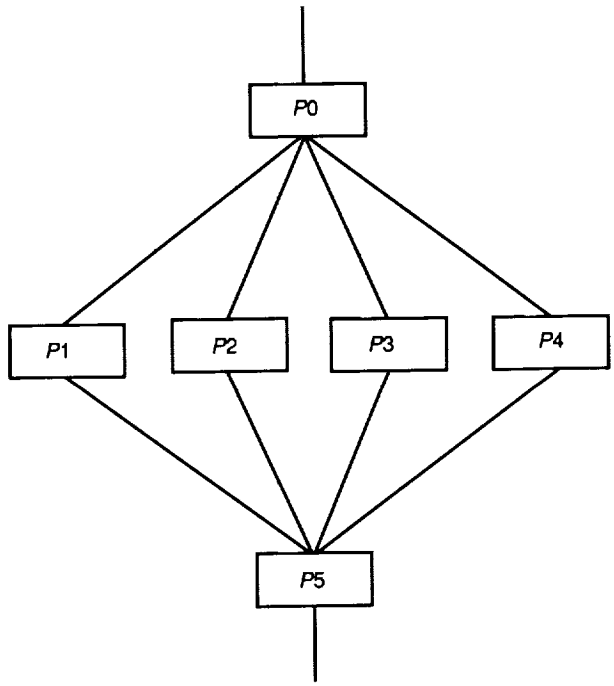


Figure 3. Fork-join algorithm example (with width of four processes).

The subsequent experiments use all three workload classes (fork-join, binary tree, and diamond-shaped) to show the effect of scheduling and communication overhead. Fork-join process durations are chosen in the same manner as just described. Process

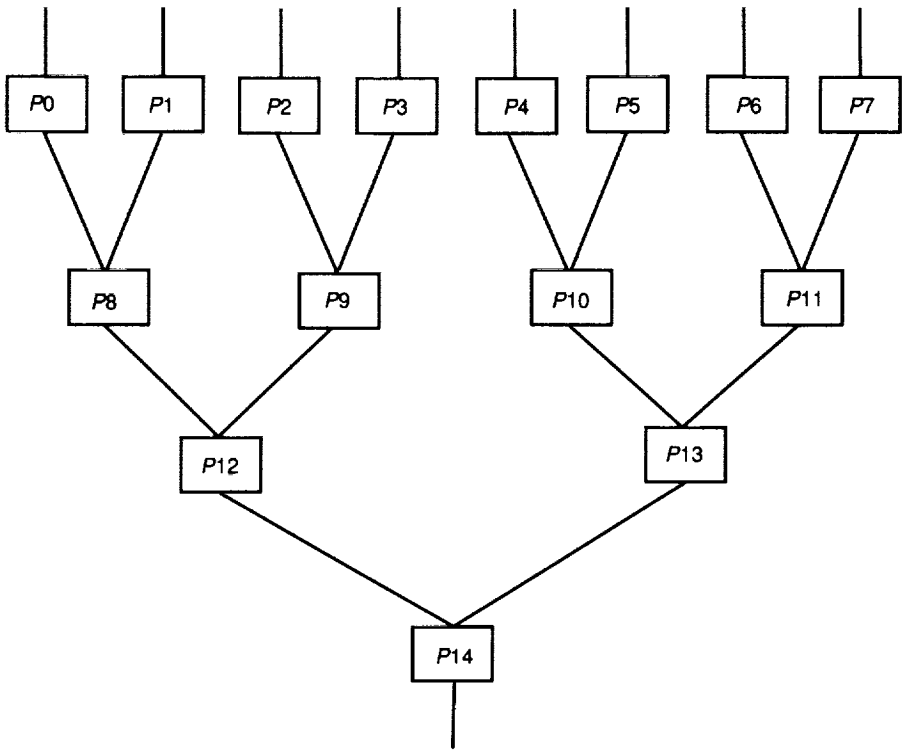


Figure 4. Binary tree algorithm example.

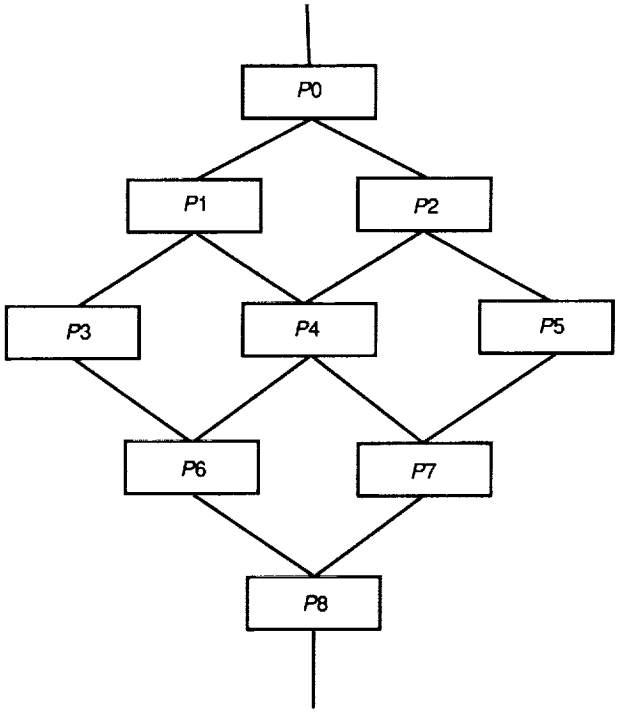


Figure 5. Diamond-shaped algorithm example.

durations for the binary tree and the diamond-shaped algorithms are chosen from a normal distribution with a mean of 1.0 and a standard deviation of 10 percent. Communication and scheduling overheads range from 0 to 50 percent of the duration of the process being communicated or scheduled. Again, the workloads are executed for 100 iterations, and the results are averaged.

4. Simulation, Results, and Analysis

4.1. Problem Size Effects for Fork-Join Class of Problems

The initial suite of experiments was aimed at studying the effect of problem size on speedup or task latency for a given multiprocessor architecture. For these experiments, we assume that no scheduling or communication overhead exists. Figure 6 shows the results of experiments that were run using the fork-join algorithm with increasing width (fan-out). The fork-join algorithm is used because it represents the most general class of problems encountered in parallel applications.

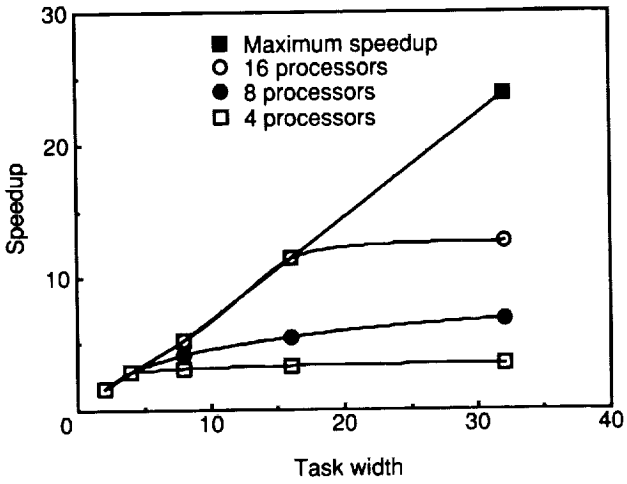


Figure 6. Speedup gains for fork-join algorithm.

The four curves in figure 6 represent speedup gains for three multiprocessor architectures as well as for the optimal case in which one has an infinite number of parallel processors. Maximum speedup is calculated using Amdahl's Law (ref. 19), which states that the most speedup achievable depends on the amount of inherent parallelism in the application. The maximum speedup $S_{\max}$ is calculated using

$$S_{\max} = \frac{T_s}{T_{cp}} \tag{1}$$

where T_s is the time to execute the work in serial (sequentially) and T_{cp} is the time to execute the critical path. The critical path in this sense is the path through the dataflow graph that takes the longest to execute. Note that there is another factor constraining speedup which is equal to the number of processing elements in the architecture (i.e., for a 16-processor system, we can never exceed a speedup of 16). We denote this maximum speedup $S_{p\max}$.

For the four-processor architecture, speedup remains close to 3.5 as the parallel fraction grows. (Remember, the parallel fraction is that percentage of the problem which can be done in parallel.) Now, we can introduce another measure E, which will capture the efficiency of the scheduling paradigm to utilize the available processing power. The term E is defined as

$$E = \frac{S}{S_{p\max}} \tag{2}$$

where S is the speedup observed during the simulation. For example, in figure 7, the scheduling efficiency E of a single fork-join task with a width of 32 processes will be 0.875, 0.854, and 0.762 for the 4-, 8-, and 16-processor architectures, respectively. Obviously, by increasing the amount of parallelism in the load, we can increase these efficiencies. However, the important observation here is that regardless of the load size, the efficiency of the scheduler will be high as long as the width of the fork is at least as large as the number of processors (fig. 7). Further, by implementing multitasking, we will show in the next section that even for small tasks, high single-task efficiency can be achieved.

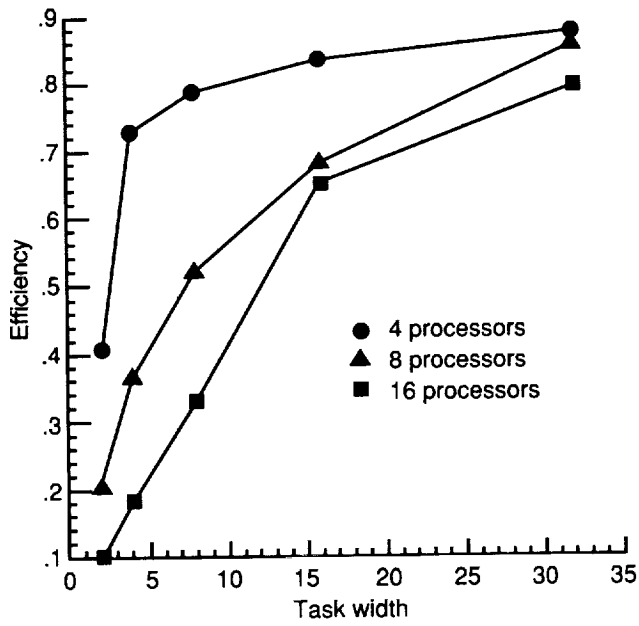


Figure 7. Efficiency of scheduler for fork-join algorithm.

4.2. Multitasking for Fork-Join Class of Problems

Next, we observe the effect of implementing multitasking into our scheduling paradigm. Again, we assume the ideal situation in which we have no schedule or communication overheads and we can generate speedup figures that are annotated with the multitasking data (figs. 8, 9, and 10). In addition, all tasks that make up a multitask job have equal width.

The data from the four-processor configuration (fig. 8) show us that we can maintain a near-constant speedup for a given number of tasks, regardless of how much inherent parallelism exists within each task, as long as the intratask parallelism is approximately equivalent across the tasks. For example, the curve representing the speedup for two tasks in figure 8 flattens after an intratask width of four processes. The speedup remains at or near 2.0. While the per-task efficiency is only 50 percent, the processors are doing twice as much work compared with

the single-task scheduling that has a 78-percent efficiency. This implementation is not necessarily good. Actually, by putting the two tasks together (concatenation) and running them as a single larger task, we could get the 78-percent efficiency ($S = 3.1$) because the speedup remains almost constant as the task complexity increases.

The real payoff for implementing a multitasking scheduler becomes evident when there is not sufficient parallelism in a single task to effectively use the processing power that is available. When this occurs, we can increase the parallel fraction by permitting multiple tasks to run concurrently. For example, figure 8 shows that when the intratask parallelism is low (with a width of two processes), we achieve almost the same speedup whether we execute one or two tasks ($S = 1.630$ and 1.615, respectively). This speedup is also very near the maximum speedup that is achievable ($S_{max} = 1.640$).

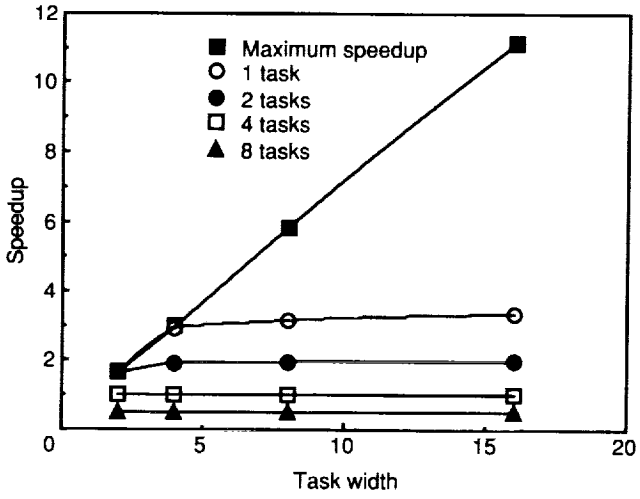


Figure 8. Multitasking speedup gains for fork-join algorithm (with four processors).

This behavior is even more apparent in the 8- and 16-processor systems (figs. 9 and 10). Notice in figure 9, when there is a small inherent parallelism (with a width of two processes), we achieve nearly the same speedup ($S = 1.63$, 1.57, and 1.46, respectively) whether we execute one, two, or four tasks. This again is near the maximum achievable speedup ($S_{max} = 1.64$). Further, with more processing power, intratask parallelism can increase and still sustain similar speedup. Figure 9 shows that even if the width is four processes, we can execute one or two tasks concurrently and maintain speedup ($S = 2.92$ and 2.64, respectively) near the maximum level ($S_{max} = 2.945$). Figure 10 further reveals this trend. With 16 processors, we can maintain speedup with a larger number of tasks as well as with tasks of increased complexity.

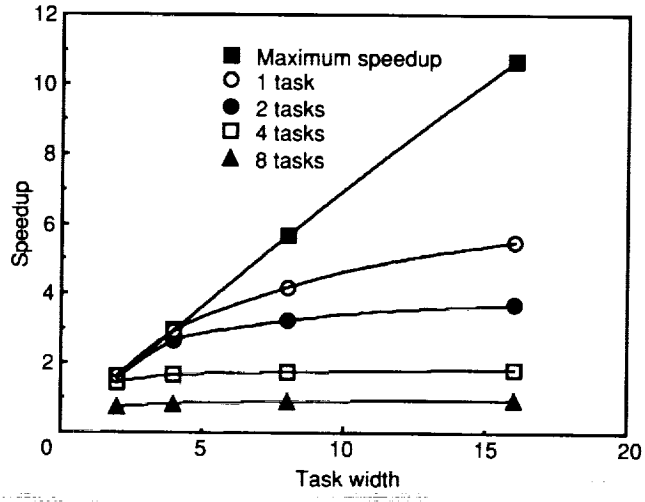


Figure 9. Multitasking speedup gains for fork-join algorithm (with eight processors).

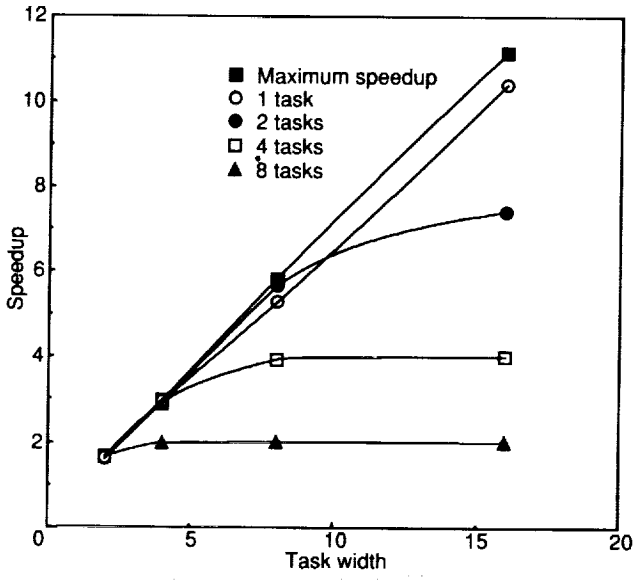


Figure 10. Multitasking speedup gains for fork-join algorithm (with 16 processors).

Another way to view the benefit of multitasking is to look at the processor utilizations with respect to the speedup gains and the amount of total work that can be done (figs. 11, 12, and 13).

In figure 11, notice that either one or two "narrow" (with a width of two processes) tasks can be completed without overutilizing the processors (40 and 80 percent, respectively), thereby maintaining the speedup gain shown in figure 8. However,

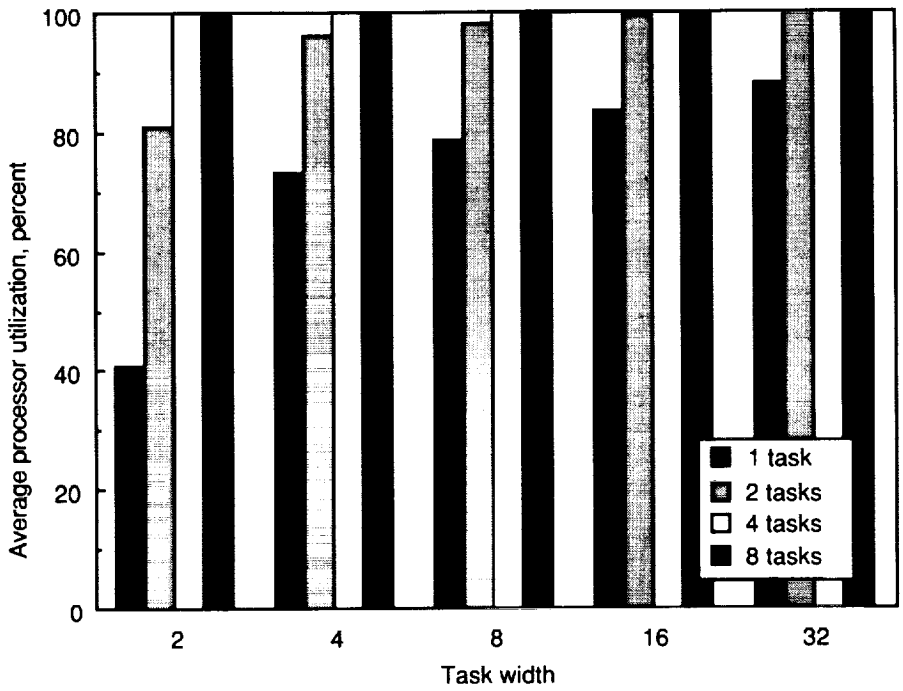


Figure 11. Average processor utilizations for fork-join algorithm (with four processors).

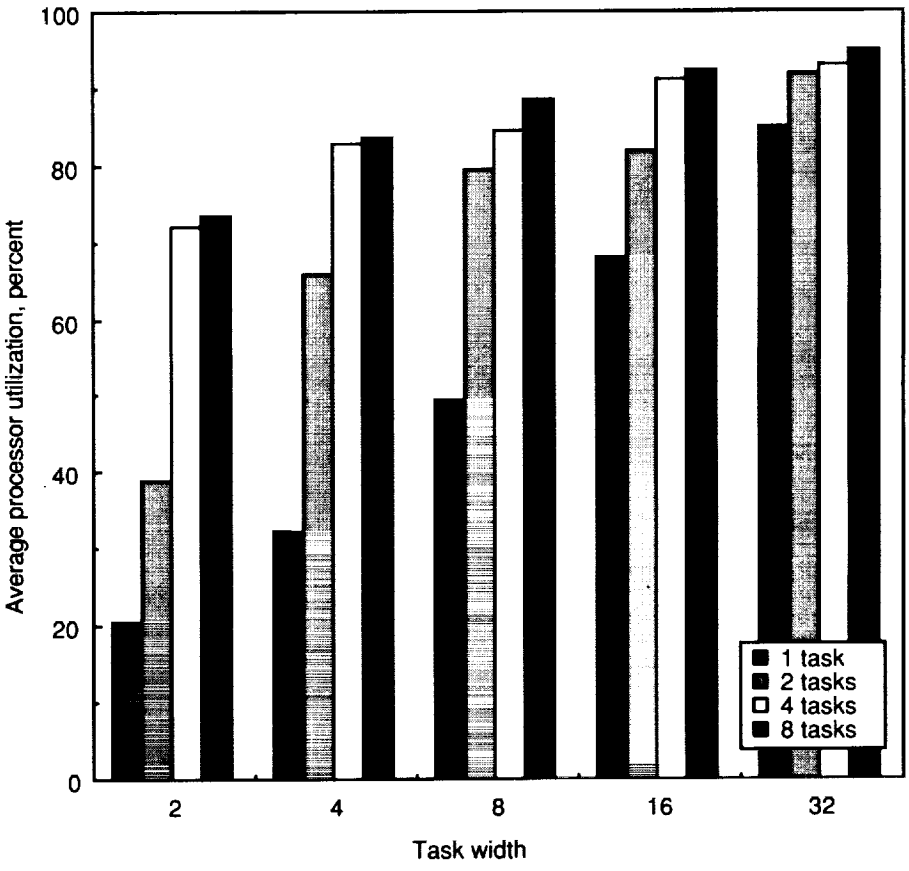


Figure 12. Average processor utilizations for fork-join algorithm (with eight processors).

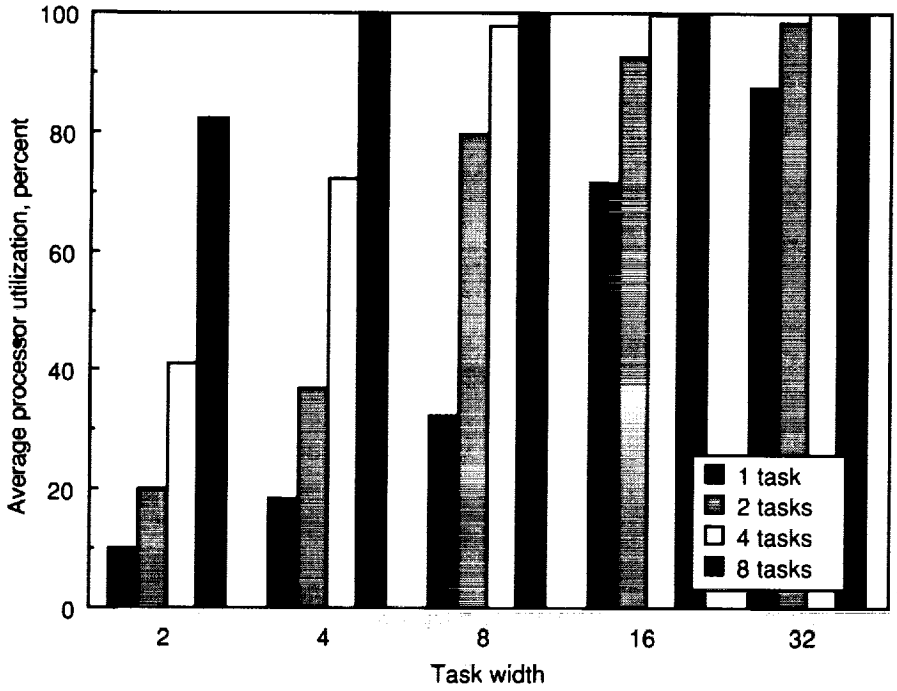


Figure 13. Average processor utilizations for fork-join algorithm (with 16 processors).

once the processors do become overutilized (with four or eight tasks), multitasking no longer helps on a four-processor system. Again, this observation supports the data in figure 8. We conjecture that for small systems, it is best to use multitasking only for tasks that have a small parallel fraction (width). For tasks with a large parallel fraction, the nature of the dataflow scheduler will optimize the achievable speedup if the tasks are done sequentially and not concurrently.

Ideally, through the use of multitasking, we would like to keep the levels (i.e., the number of entries) of both queues (i.e., JobList and PEready) close to equal. Maintaining near-equal levels in these two queues minimizes the amount of idle time for each processor. If the queue containing the ready jobs backs up, the processors become overutilized; whereas, if the queue containing the idle processor identifiers backs up, the processors are underutilized. This ideal level should be no more than the number of processors in the system.

Figures 12 and 13 reveal the effect of adding processing power (8- and 16-processor systems). Note, in these cases, "narrow" tasks can be executed concurrently to effectively utilize all processor bandwidth and maintain speedup (figs. 8 and 9). Also, the "fatness" of tasks can increase and maintain speedup as long as the utilization is not too high.

All this information supports Amdahl's Law that speedup is constrained by parallel fraction, and it also shows the effect of limited processing power on the attainable speedup. Perhaps the most revealing evidence of the relationship between these two determinants (the parallel fraction and the available resources) is shown in figures 14 and 15.

The solid lines in figure 14 represent workloads with different degrees of multitasking. In figure 15, each individual workload is shown along with its projections (the dashed lines) onto a two-dimensional grid. We can see that by introducing multitasking to an underutilized system, we can effectively increase the parallel fraction and, thus, do more work at the same level of speedup. Multitasking becomes inefficient (in terms of the effective speedup) only when the processing power constraint is exceeded (with a utilization of approximately 100 percent) and the speedup deteriorates. Of course, this situation assumes that the tasks are of near-equal complexities. If this is not the case, the scheduler could use priorities or weighting factors to preclude "fairness" toward allocating work to resources. From figure 15, we see that to get the most work done at the highest

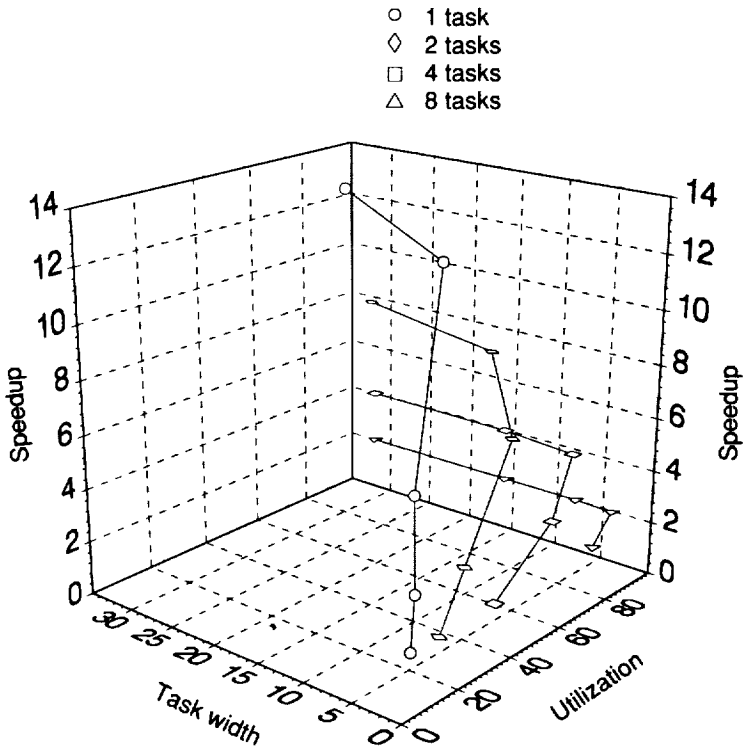


Figure 14. Benefits of multitasking (with 16 processors).

speedup, we should use a width of approximately 32, 16, 8, and 4 processes for 1, 2, 4, and 8 tasks, respectively. These guidelines allow us to avoid over-utilization while we maintain speedup.

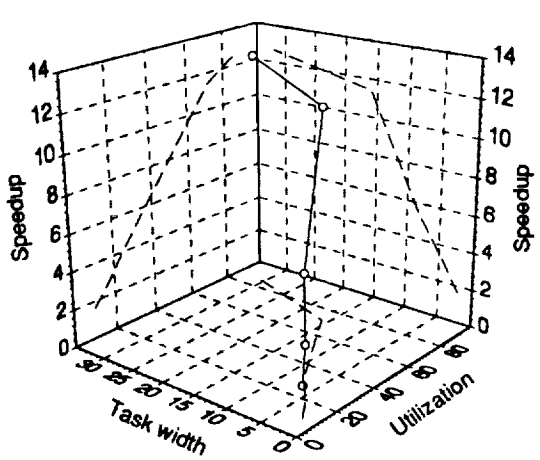
4.3. Scheduling Overhead Effects

This suite of experiments will attempt to show the effects of scheduling delays. We will use all three workload classes discussed in section 3 (e.g., fork-join, binary tree, and diamond-shaped problems). These simulations were all run on the 16-processor architecture model.

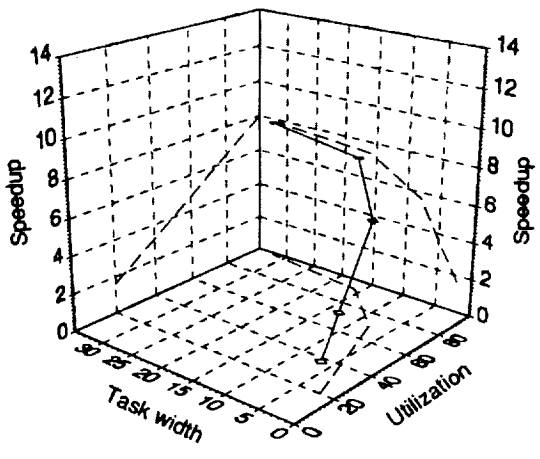
The data in figure 16 were taken while simulating a single task of each type. All three tasks have the same order of complexity (i.e., the fork-join, binary tree, and diamond-shaped problems have 258, 511, and 529 processes, respectively). The differences, with respect to the scheduler, are the amount of communication that must take place and the number of instantiations of the scheduling mechanism required.

Two interesting phenomena can be observed in figure 16. First, note the initial flatness of each curve. Both the binary tree and the diamond-shaped algorithms can maintain a near-constant speedup as long as the scheduling overhead is less than 8 percent; beyond that, the speedup drops off in an exponential decay. The fork-join application can incur up to 20 percent overhead while losing only about 6 percent of its speedup (13.7 down to 12.5) before it too begins to exponentially decay.

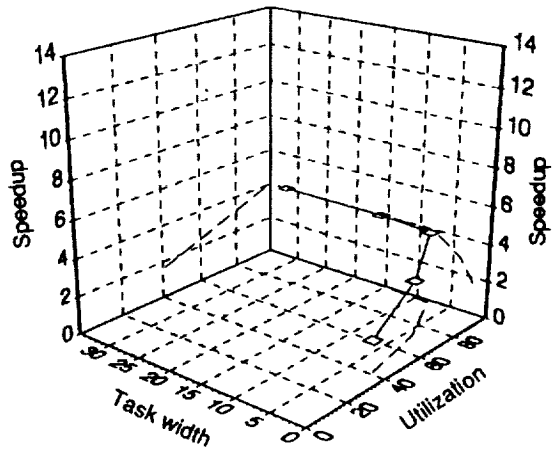
This near-constant behavior at small overheads is due to the number of scheduling events S_e that must occur during the execution of a specific algorithm. For the macro-dataflow scheduler described, S_e is equal to the number of levels in the dataflow representation of the workload. For asymmetrical dataflow graphs, S_e would be the number of processes in the longest path through the graph. The larger the S_e, the more work the scheduler must do and, hence, the more effect that the scheduling overhead will have on the speedup. The curves shown in figure 16, although having similar complexity (the number of processes), have different values of S_e. For example, S_e for the fork-join problem will always be just three; those for the binary tree problem will be $\log_2 n$ (where n is the number of processes at the top of the tree), and those for the diamond-shaped problems will be $2n - 1$ (where n is the number of processes at the center level of the diamond).



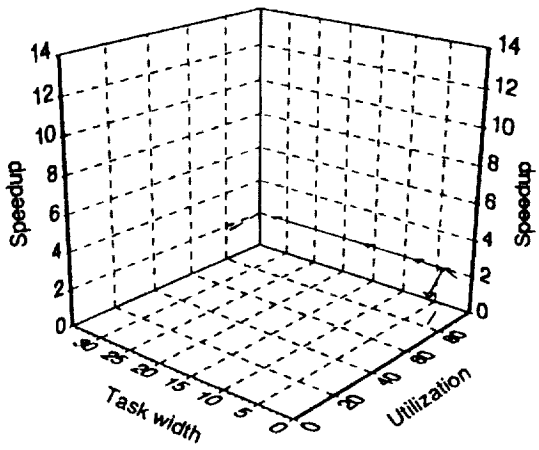
(a) 1-task processor.



(b) 2-task processor.



(c) 4-task processor.



(d) 8-task processor.

Figure 15. Two-dimensional projections of figure 14.

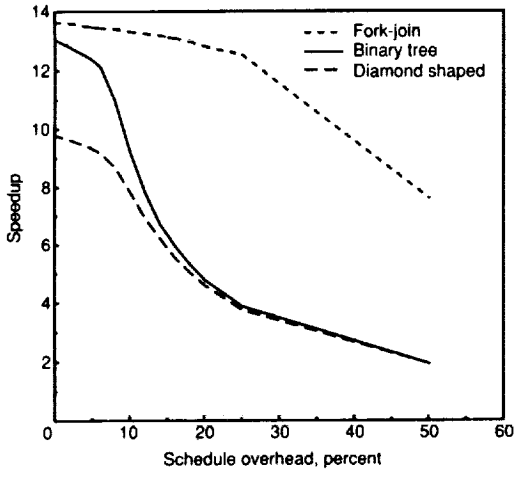


Figure 16. Speedup gains and scheduling overhead with 16 processors).

These data reveal that for certain classes of workloads, the effect of scheduling overhead on speedup can be minimal. Also, they show that scheduling overhead can have a serious effect (exponential decay) on speedup once it gets beyond a certain threshold. However, up to this threshold, speedup can be maintained near a constant.

With the simplistic nature of this macro-dataflow scheduling paradigm (i.e., a queue), we feel that the expected overhead would be low (less than 5 percent) which, according to these data, would allow us to maintain the speedup achievable with no overhead.

4.4. Communication Overhead Effects

We now look at the effects of interprocessor communication delays. Here, we use the same three workload classes that were used in section 4.3: a fork-join problem with 258 processes, a binary tree problem with 511 processes, and a diamond-shaped problem with 529 processes; all these workloads execute on the 16-processor architecture model. A much

more consistent behavior can be seen (fig. 17). For each class of workload, a period of near-constant speedup (with an overhead of less than 7 percent) exists, which is followed by an exponential decay in the speedup gain.

The differences in speedup here are due to the amount of communication which must be done by each class of algorithm. This amount is directly proportional to the number of edges in the dataflow diagram that represents the workload, as shown in the following table:

Workload	Edges
Fork-join	$2(n-2)$
Binary tree	$2(n-1)$
Diamond shaped . . .	$2(n-\sqrt{n})$

Here n is the number of processes in the workload. These data (fig. 17) show that for tasks with near-equivalent complexity (i.e., the number of processes), the fork-join problem will perform best ($S = 13.667$), followed closely by the binary tree problem ($S = 13.05$), and then the diamond-shaped problem ($S = 9.798$). Notice that these data reflect our conjecture about the number of edges because, for this case, the numbers of edges are 512, 510, and 1012 for the fork-join, binary tree, and diamond-shaped problems, respectively. In general, the fork-join problem will have $2(n-2)$ edges (the fewest), the binary tree problem will have $2(n-1)$ (just two more), and the diamond-shaped problem will have $2(n-\sqrt{n})$.

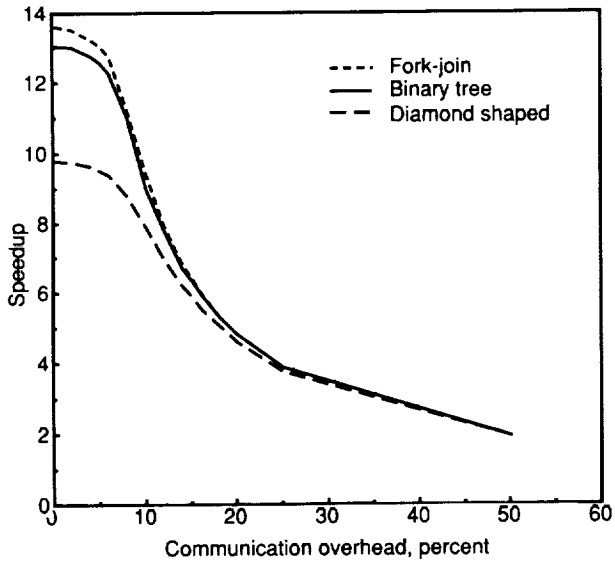


Figure 17. Speedup gains and communication overhead (with 16 processors).

Therefore, we believe that for values of n much greater than $\sqrt{n}$, the behavior of the three workloads will be similar. However, for small values of n, the effect will be much more significant.

It is apparent from this analysis that the communication overhead has a greater effect on performance than does the scheduling delay, which is due in part to the nature of the macro-dataflow paradigm for scheduling work (i.e., near optimal). This effect also occurs because the number of edges will be much larger than the number of scheduling events for most large problems. Therefore, although we have shown the efficiency of this scheduling paradigm, we are still faced with finding ways to implement fast reliable communication.

5. Conclusions

This paper presents a performance analysis of a macro-dataflow mechanism that can optimally schedule work onto a set of distributed processors using large-grain dataflow representations of the workload as input. With the emergence and acceptance of computer-aided software engineering (CASE) tools, whose underlying structure for software designs is functional decomposition into dataflow diagrams, generation of this input form becomes straightforward.

Performance analysis results, obtained via simulation, are presented which reveal the effects of problem size and class on speedup and processor utilizations. Also shown are the benefits of including a multitasking capability to increase the effective parallel fraction of the workload. By adding the multitasking capability, we show that more work can be done with the same speedup gain while increasing the processor utilizations.

Finally, we present quantitative data that show the effects of scheduling and communication overheads. In both cases, a period of constant speedup exists (as the overhead increases), which is followed by an exponential decay in the speedup. The rate of this decay depends on the parameters that determine the amount of scheduling (the number of scheduling events) or the communication (the number of edges) that must take place for a given workload. Note that one of the salient features of macro-dataflow is a reduced number of required communication and scheduler events.

Future work will include looking at the multiprogramming of individual processors to reduce the effect of large input/output delays, investigating the implementation issues, and further analyzing the performance of specific applications.

References

1. Lee, Edward A.; and Messerschmitt, David G.: Synchronous Data Flow. *Proc. IEEE,* vol. 75, no. 9, Sept. 1987, pp. 1235–1245.

2. Shaffer, Phillip L.; and Johnson, Timothy L.: Data Flow Analysis of Concurrency in a Turbojet Engine Control Program. *Seventh American Control Conference, Volume 3,* Inst. of Electrical and Electronics Engineers, Inc., 1988, pp. 1837–1845.

3. Jagannathan, R.; Downing, A. R.; Zaumen, W. T.; and Lee, R. K. S.: Dataflow-Based Methodology for Coarse-Grain Multiprocessing on a Network of Workstations. *Proceedings of the 1989 International Conference on Parallel Processing, Volume II,* Emily C. Olachy and Peter M. Kogge, eds., Pennsylvania State Univ. Press, 1989, pp. II-209–II-215.

4. Babb, Robert G., II; Storc, Lise; and Ragsdale, William C.: A Large-Grain Data Flow Scheduler for Parallel Processing on CYBERPLUS. *Proceedings of the 1986 International Conference on Parallel Processing,* Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, eds., IEEE Catalog No. 86CH2355-6, IEEE Computer Soc., 1986, pp. 845–848.

5. Arvind; Culler, David E.; and Maa, Gino K.: Assessing the Benefits of Fine-Grain Parallelism in Dataflow Programs. *Proceedings Supercomputing '88,* IEEE Catalog No. 88CH2617-9, IEEE Computer Soc., 1988, pp. 60–69.

6. Gokhale, Maya B.: Macro vs Micro Dataflow: A Programming Example. *Proceedings of the 1986 International Conference on Parallel Processing,* Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, eds., IEEE Catalog No. 86CH2355-6, IEEE Computer Soc., 1986, pp. 849–852.

7. Grimshaw, A.; and Lui, J.: *Mentat: An Object-Oriented Macro-Dataflow System.* NASA TM-101165, 1988.

8. Ghosal, Dipak; and Bhuyan, Laxmi N.: Performance Evaluation of a Dataflow Architecture. *IEEE Trans. Comput.,* vol. 39, no. 5, May 1990, pp. 615–625.

9. Louri, Ahmed: An Optical Data-Flow Computer. *Optical Information Processing Systems and Architectures,* Bahram Javidi, ed., *Volume 1151 of Proceedings of the SPIE—International Society of Photo-Optical Instrumentation Engineers,* 1990, pp. 47–58.

10. Grimshaw, Andrew S.; Silberman, Ami; and Liu, Jane W. S.: Real-Time Nentat Programming Language and Architecture. *Globecom '89—IEEE Global Telecommunications Conference & Exhibition, Volume 1,* IEEE Catalog No. 89CH2682-3, IEEE Communications Soc., 1989, pp. 4.6.1–4.6.7.

11. Mielke, Roland R.; Stoughton, John W.; and Som, Sukhamoy: *Modeling and Optimum Time Performance for Concurrent Processing.* NASA CR-4167, 1988.

12. *Teamwork/SA®, Teamwork/RT®—User's Guide, Release 4.0.* Cadre Technologies Inc., c.1991.

13. *The Software Through Pictures (StP) User's Manual.* Interactive Development Environments, 1991.

14. *ALS Case (Advanced Launch System Computer-Aided Software Engineering)—User's Manual.* Charles Stark Draper Lab., Inc.

15. Parhi, Keshab K.; and Messerschmitt, David G.: Fully-Static Rate-Optimal Scheduling of Iterative Data-Flow Programs Via Optimum Unfolding. *Proceedings of the 1989 International Conference on Parallel Processing, Volume I,* Kevin P. McAuliffe and Peter M. Kogge, eds., Pennsylvania State Univ. Press, 1989, pp. I-209–I-216.

16. Mielke, R.; Stoughton, J.; Som, S.; Obando, R.; Malekpour, M.; and Mandala, B.: *Algorithm to Architecture Mapping Model (ATAMM) Multicomputer Operating System Functional Specification.* NASA CR-4339, 1990.

17. Som, S.; Obando, R.; Mielke, R. R.; and Stoughton, J. W.: ATAMM: A Computational Model for Real-Time Data Flow Architectures. *Call for Papers, ISMM International Conference—Parallel and Distributed Computing, and Systems,* International Soc. for Mini and Microcomputers, 1990, pp. 241–245.

18. *ADAS—An Architecture Design and Assessment System for Electronic Systems Synthesis and Analysis—User's Manual, Version 2.5.* Cadre Technologies Inc., c.1988.

19. Hwang, Kai; and Briggs, Fayé A.: *Computer Architecture and Parallel Processing.* McGraw-Hill, Inc., c.1984.

*U.S. GOVERNMENT PRINTING OFFICE: 1993-728-150/60060

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1993	Technical Paper

4. TITLE AND SUBTITLE
Performance Analysis of a Large-Grain Dataflow Scheduling Paradigm

5. FUNDING NUMBERS
WU 509-10-04

6. AUTHOR(S)
Steven D. Young and Robert W. Wills

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17128

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TP-3323

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 62

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper describes and analyzes a paradigm for scheduling computations on a network of multiprocessors using large-grain dataflow scheduling at run time. The computations to be scheduled must follow a static flow graph, while the schedule itself will be dynamic (i.e., determined at run time). Many applications characterized by static flow exist, and they include real-time control and digital signal processing). With the advent of computer-aided software engineering (CASE) tools for capturing software designs in dataflow-like structures, macro-dataflow scheduling becomes increasingly attractive, if not necessary. For parallel implementations, using the macro-dataflow method allows the scheduling to be insulated from the application designer and enables the maximum utilization of available resources. Further, by allowing multitasking, processor utilizations can approach 100 percent while they maintain maximum speedup. Extensive simulation studies are performed on 4-, 8-, and 16-processor architectures that reflect the effects of communication delays, scheduling delays, algorithm class, and multitasking on performance and speedup gains.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Dataflow; Scheduling; Multiprocessors; Multitasking performance	15
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		

NSN 7540-01-280-5500

Standard Form 298(Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

NASA-Langley, 1993