Improving Parallel Program Performance Using Critical Path Analysis

Andrew W Kwan, Lubomir Bic, and Daniel D. Gajski

Department of Information and Computer Science
University of California Irvine
Irvine, California 92717

Technical Report #89-05
January 1989

Improving Parallel Program Performance Using Critical Path Analysis

Andrew W Kwan, Lubomir Bic, and Daniel D. Gajski

Department of Information and Computer Science
University of California Irvine
Irvine, California 92717

abstract

A programming tool that performs analysis of critical paths for parallel programs has been developed. This tool determines the critical path for the program as scheduled onto a parallel computer with P processing elements, the critical path for the program expressed as a data flow graph (when maximal parallelism can be expressed), and the minimum number of processing elements (P_{opt}) needed to obtain maximum program speedup. Experiments were performed using several versions of a Gaussian elimination program to examine how speedup varied with changes in granularity and critical path length. These experiments showed that when the available number of processing elements $P < P_{opt}$, increasing granularity improved program speedup more than reducing (the data flow graph's) critical path length, whereas when $P \geq P_{opt}$, increasing granularity degraded program speedup while reducing critical path length improved program speedup.

Improving Parallel Program Performance Using Critical Path Analysis

Andrew W Kwan, Lubomir Bic, and Daniel D. Gajski

Department of Information and Computer Science
University of California Irvine
Irvine, California 92717

1. Introduction

There is currently much research in programming parallel computers. Many parallel programming environments utilize an approach where the programmer writes the program as a set of tasks with clearly defined communication and synchronization (e.g., MUPPET [MKL87], POKER [SnS86], and Polylith [PRG87]). However, to improve parallel program performance most programmers utilize a "trial and error" method on parameters under programmer control (e.g., granularity, scheduling, etc.). For instance, a programmer might first measure a program's performance using a profiler, analyze the information provided by the profiler, and then alter the program's granularity (either in the coarser or finer direction) depending on the amount of communications overhead. The programmer would then execute the program again, and if the desired performance was not yet realized, another attempt at improvement would be tried. Alternatively, the programmer could develop another algorithm.

This situation can be improved in a number of ways. First, the programmer should have tools that provide parallel program performance information. These tools should be able to provide, at the very least, some of the more routine analyses that a programmer would need to have done. Second, the programmer must rely upon intuition to make changes to improve performance. Parallel programming is a very intuitive craft, and less intuitive programmers will have more difficulty improving programs. Guidelines for program improvement would help to reduce the intuition needed and make parallel programming easier.

We have developed a programming tool that performs critical path analysis of parallel programs. This tool determines the critical path for the program as scheduled onto a parallel computer with P processing elements, the critical path for the program expressed as a data flow graph (when maximal parallelism can be

expressed), and the minimum number of processing elements (P_{opt}) needed to obtain maximum program speedup. When $P=P_{opt}$, the length of the critical path of the scheduled program is less than or equal to that of the data flow graph, and the scheduled program will execute as quickly as possible. The maximum speedup of the parallel program lies between $P_{opt}-1$ and P_{opt}, and the method of determining P_{opt} is an empirical method for determining the maximum speedup predicted by Amdahl's law. The knowledge of the maximum possible speedup is useful to the programmer, as it tells the programmer the best performance that can be expected of the program, and provides a yardstick by which the current performance can be measured against.

Experiments were performed using several versions of a Gaussian elimination program to examine how speedup varied with changes in granularity and critical path length. The results of these experiments showed that when the available number of processing elements $P < P_{opt}$, increasing granularity improved program speedup more than reducing (the data flow graph's) critical path length, whereas when $P \geq P_{opt}$, increasing granularity degraded program speedup while reducing critical path length improved program speedup. These results are used to provide guidelines for parallel program performance improvement.

2. Critical Path Analysis

Dynamic data flow analysis techniques can be used to analyze and improve parallel program performance. In static data flow analysis, the program source code is analyzed to provide information about the program without having to execute it. In dynamic data flow analysis, the results of program execution are combined with the information from static data flow analysis to provide information on program performance.

In this section, we define two critical paths for parallel program analysis: the data flow graph critical path, and the scheduled program critical path. When a program is represented as a data flow graph, the critical path through the graph represents the quickest possible program execution when the maximum

possible parallelism is available. A scheduler takes the data flow graph and assigns its tasks and communications to a computer with a fixed number of processing elements and fixed communication mechanisms. This scheduled program also has a critical path, which tells how quickly the program actually executes. Comparison of the two critical paths can yield information useful for improvement of parallel program performance.

2.1 Program Graphs and Critical Paths

A parallel program can be written as a set of tasks, where each task is composed of a set of instructions to be executed in a sequential manner on an assigned processing element. Each task has clearly defined input and output parameters. Tasks cannot commence execution until all input parameters are available, thereby serving as the synchronization mechanism. Output parameters are sent out by the task after task completion. By executing the program, information about the execution time of each task and the time required to read or write each input or output parameter can be found (on a message passing computer, this corresponds to transmission time, whereas on a shared memory computer this corresponds to memory access time).

A data flow graph can be constructed from this information, and provides a representation of the program. Each node of the graph corresponds to a task, and the weight of a node is assigned the execution time for the task. Each edge leaving a node corresponds to an output parameter of a source node, each edge entering a node corresponds to an input parameter of the destination node. The weight of the edge is the time required to transmit or access the parameter. This data flow graph will be a directed, acyclic graph.

This data flow graph can be thought of as an abstraction of the program, and we can define an abstract computer to execute program. In this abstract computer, there is one processing element available for each task. Each edge of the data flow graph has a corresponding communications channel between processing elements. When thought of in this manner, the data flow graph illustrates the maximum parallelism that can be found in the program. If such an abstract computer existed, the program could execute at the

quickest possible speed. However, since each edge is utilized, the data flow graph also has the maximum communications overhead possible for program execution.

A path from some node i to another node j in a graph is a collection of nodes and edges that start at node i and end at node j. The data flow graph critical path is defined to be the longest path through the data flow graph, that is, is a path through the graph that has the largest sum of node weights and edge weights. The length of the data flow graph critical path is defined to be the sum of the weights of the nodes and edge in the data flow graph critical path. This critical path corresponds to the execution time of the program when maximum parallelism is possible. However, the critical path also contains a cost for communications.

The data flow graph computational critical path is defined as the path through the graph with the largest sum of node weights only. This is useful because communications cost is not considered. Usually, programmers want to maximize computation and minimize overhead. The computational critical path shows the computation-intensive path through the program. By comparing the computational critical path with the critical path, the programmer can get a feel for the impact of the communication costs for the program.

A scheduler assigns tasks to processing elements for execution. In essence, the scheduler transforms the data flow graph (a representation for an abstract computer) into a scheduled program (a representation for a real computer). The scheduled program itself can also be represented as a directed, acyclic graph. The scheduled program graph has the same nodes and edges as the data flow graph, but there are edges added to provide sequencing among tasks assigned to the same processing element. These new sequencing edges each have a weight of zero, and can be considered equivalent to an input parameter, as a task cannot commence execution until all input parameters are available and the previous task assigned to the processing element (i.e., the source node of the sequencing edge) has completed execution. Communications between two tasks has no cost when both tasks are assigned to the same processing element, since the tasks can communicate with each other through local memory (otherwise there is a cost

associated). This results in a change of weight (to zero) for some communication edges. We similarly define a critical path and computational critical path for the scheduled program graph. Comparison of the critical path and the computational critical path of the scheduled program graph can also be used to analyze communication costs.

The scheduled program graph is a compromise between the data flow graph, the capabilities of the real computer, and a sequential computer. The data flow graph expresses the maximum parallelism of the parallel program, but also expresses the maximum communications overhead. The real parallel computer has a limit on the number of tasks that it can execute simultaneously, but can reduce communication costs by assigning tasks to the same processing element (allowing tasks assigned to the same processing element to communicate through local memory). The sequential computer cannot provide for any parallelism since it can only execute one task at a time, but has no communications overhead since all tasks execute on the same processor and can communicate through memory. By comparing the critical paths of the scheduled program graph and the data flow graph, we can gauge the performance of the scheduled program against that of the ideal performance.

In general, this comparison assumes that computation, and not communication, dominates program performance. As previously stated, the data flow graph expresses maximum parallelism and maximum communication cost, whereas the scheduled program graph expresses intermediate parallelism and intermediate communication cost. If communication costs were dominant, substantial reductions between scheduled program graph and data flow graph critical path length are possible since communication costs can be reduced by scheduling, and the tasks performed in each critical path may have very little correspondence. When computation costs are dominant, there will be some correspondence between tasks on the critical paths, and there is a basis for comparison. This assumption can be valid for many programs. For the Gaussian elimination program considered in this paper, communication costs are relatively uniform for each task and path of the data flow graph, and computation time for each task is greater than the communication time. The computational critical path provides a check on communication costs — if there

is little resemblance between the critical path and the computational critical path (for either the scheduled program graph or the data flow graph), then communication, and not computation, is driving program performance.

2.2 The HYPERTOOL Method

A parallel programming aid called HYPERTOOL [WuG88] has been previously developed by our research group. HYPERTOOL relieves the programmer of scheduling, communication and synchronization insertion, and mapping of tasks onto processing elements. Using HYPERTOOL, a programmer develops an algorithm and expresses it (using a subset of the C programming language) as a set of serial and parallel procedures called by a main program. Each procedure performs some task, and has clearly defined input and output parameters. The main program consists of procedure calls.

The program can be executed, tested, and debugged on a sequential computer. The program is then analyzed by HYPERTOOL. HYPERTOOL utilizes dynamic data flow analysis to construct a data flow graph of the program. HYPERTOOL's scheduler then automatically (statically) schedules the tasks onto processing elements, inserts the appropriate communication and synchronization primitives, and maps the processing elements onto a hypercube. The program is then executed on a hypercube simulator.[1] The output of the simulator provides an event trace and statistics on processing element utilization and communications.

Using HYPERTOOL, program development has taken much less time compared to manual coding of programs. Comparison of execution time of programs developed using HYPERTOOL versus manually coded programs have demonstrated up to 300% improvement. Furthermore, there has been no observed degradation.

The critical path analysis tool interfaces with HYPERTOOL. It utilizes the data flow graph produced during HYPERTOOL's analysis to determine critical path and computational critical path of the data flow graph. These paths are found by performing a breadth-first search on the graph, finding the longest path to

each node as the graph is traversed, and retaining the longest path.[2] The tool then converts the data flow graph representation into a timed Petri net. The programmer can then view the Petri net, and watch an animated execution of the Petri net.[3]

After HYPERTOOL's scheduler produces the scheduled program graph and schedules the program, the critical path analysis tool determines the scheduled program graph's critical path and computational critical path. The tool then compares the length of the data flow graph's critical path against that of the scheduled program's. Based upon the results of the comparison, the tool searches for a the minimum number of processing elements needed to produce a scheduled program whose critical path length is less than that of the data flow graph.

2.3 Amdahl's Law

Amdahl's Law [Gus88] provides a theoretical limit to parallel program performance based upon the amount of sequential code that exists in the program. The equation for the parallel program speedup S can be written as

$$S(P) = \frac{T_{seq}}{T_{par}(P)}$$

where P is the number of processing elements used, T_{seq} is the execution time of the sequential program, and $T_{par}(P)$ is the execution time of the parallel program on a computer with P processing elements.

Let x be the fractional amount of code that can be executed in parallel in the sequential program. The amount of sequential code is 1–x. We then get

$$T_{par}(P) = T_{seq} \left(1 - x + \frac{x}{P} \right)$$

and

$$S(P) = \frac{P}{1 + x\,(P - 1)}$$

Let the efficiency ε, a measure of average utilization of the processing elements, be expressed as

$$\varepsilon = \frac{S(P)}{P}$$

Figure 1 provides a plot of ε versus x for various values of P. It shows that small amounts of parallel code can limit performance greatly, and especially so when large numbers of processing elements are used. On the other hand, it also shows that small improvements in the amount of parallel code can yield major performance improvements.

Amdahl's Law can provide useful insight into why parallel program performance may be poor. However, in practice there are some problems. First, it is usually difficult to measure the amount of sequential and parallel code in a program. To measure this, a programmer would have to time each instruction of the program, then classify each instruction as either serial or parallel, and then add up the total time spent by instructions in either classification. Second, a programmer typically has a fixed number of processing elements available, and so will be stuck with a particular efficiency curve from Amdahl's Law. Amdahl's Law does not provide any insight into how to improve the program. It only says that the programmer must reduce the amount of serial code.

2.4 Determining the maximum possible speedup

Comparison of the critical path of the scheduled program with the critical path of the data flow graph can provide a method for determining maximum possible speedup. With the exception of communication costs, the data flow graph critical path length is the quickest possible speed that the program can execute. On the other hand, consider a program that is scheduled onto a real computer (with P processing elements). If the critical path of the scheduled program graph is longer than that of the data flow graph, then the computer did not have enough processing elements available to avoid lengthening the scheduled program graph's critical path, i.e., there was not enough parallelism available on the real computer. If the scheduled

program's critical path length is less than or equal to the data flow graph's critical path length, then there is enough parallelism available on the real computer, and the program can execute as quickly as possible. To make maximum use of the real computer, one would want to know for what value of P can the program be scheduled such that the scheduled program graph's critical path length is less than the data flow graph's (i.e., for what minimum number of processing elements will the data flow graph's critical path constrain program execution). Let P_{opt} denote this minimum number of processing elements.

P_{opt} can be found as follows. We establish a lower and upper bound on P_{opt}, select a number of processing elements P' halfway between the lower and upper bound, and schedule the program onto a computer with P' processing elements. We analyze this newly scheduled program to find its critical path length, and compare it to the data flow graph's critical path length. If it is less, then we set P' to be the new upper bound, and save the value of P' as current value for P_{opt}. If it is more, then we set P' to be the new lower bound. We then iterate the process continually until the lower and upper bounds converge (this method is similar to the strategy used in binary search). The last value stored as the current value for P_{opt} will be the desired value.

The upper bound initially is set to be the maximum breadth (for some depth) of the data flow graph. This represents the maximum number of tasks that would execute simultaneously, and thus the maximum number of processing elements that would realistically be needed for maximum parallelism. The lower bound is initially set to be the sum of the weights of all the nodes (of the data flow graph) minus the critical path length of the data flow graph, divided by the length of the data flow graph, plus one. This is a conservative estimate, but will always be lower than any possible value of P_{opt}.

P_{opt} represents the minimum number of PEs necessary to achieve maximum program speedup. In reality, the maximum speedup is some real number between P_{opt} and $P_{opt} - 1$. But the maximum speedup was predicted by Amdahl's Law, based upon the amount of parallel code in the program and the number of processing elements available. So, the method of determining P_{opt} is really an empirical method for

finding the value predicted by Amdahl's Law, without having to calculate the amount of parallel code in the program.

3. Improving Program Speedup Using P_{opt}

Experiments were performed to determine methods of improving program speedup. The program utilized was one that performed Gaussian elimination using partial pivoting.[4] This program was selected because of its structure: it has more than one type of procedure; it has to perform several steps in sequence, and therefore has a critical path; and it has a somewhat regular task structure, but the structure does not scale with the the number of processors (it only scales with the data size). In other words, it is an application typical for a multiple-instruction stream, multiple data stream computer.

The original program takes N equations with N unknowns and organizes the data into an N by N+1 matrix, and then reduces that matrix into an upper triangular matrix. It does so by performing two basic steps on each column (except the last), starting with the first column and ending with the N^{th} column. In the "FindMax" step, column k of the matrix (where $1 \le k \le N$) is searched for the maximum value contained in rows k through N. This step is finds the pivot value, which is used to reduce the column. In the next step, "UpdateMtx," each column k through N is updated based on the value found in the FindMax step for column k. Figure 2 illustrates the data flow graph for the program on an 4 by 4 matrix. Nodes labelled F_k perform the FindMax task for the k^{th} column of the matrix. Nodes labelled U_j perform the task of updating column j of the matrix ($k \le j \le N$), based upon the results of the particular F_k that the U_j is dependent upon. Edges indicate a data dependency between nodes (edges leading from one node to several other nodes do not indicate that the several nodes receive identical data). Figure 3 illustrates the scheduling (by Hypertool) of the data flow graph of Figure 2 onto 2 processing elements.

The Gaussian elimination program was modified to reduce its critical path length and to increase its granularity. The original program and its modified versions were executed (using the same data set) over

several numbers of processing elements, and their speeds compared.

The original program, "g", assigned one column of data to each UpdateMtx task. In program "ng" (new Gaussian elimination), certain UpdateMtx tasks and FindMax tasks were merged into a new task UF to remove communication of a data item and eliminate some redundant work. The overall effect was to reduce the critical path length of the data flow graph. Figure 4 shows the new data flow graph. In program "g–ig2" (Gaussian elimination — increased granularity to 2 columns), some UpdateMtx tasks (from the original program) were merged to update two columns (vice one), eliminating some redundant work. Additionally, the scheduler was able to recognize the additional input and output parameters required for the larger grained update tasks, and schedule them so that overall message passing was lowered. However, the critical path was lengthened by the increased granularity. The data flow graph for "g–ig2" is shown in Figure 5. In program "g–ig4" (Gaussian elimination — increased granularity to 4 columns), some update tasks were merged to update four columns. More redundant work and communications were eliminated over that of "g–ig2", but the critical path was lengthened more, too.

Figures 6 and 7 show the data obtained for 8 and 16 equation problems, respectively. The data listed include the sequential execution time (T_{seq}), the parallel execution time (T_{par}), the speedup (S), the efficiency (ε), the average processing element utilization (u), and the number of (message-passing) communication instructions executed. In Figures 8 and 9, the speedup of "ng", "g–ig2", and "g–ig4" relative to that of "g" (relative speedup is calculated as the execution speed of "g" on P processing elements divided by the execution speed of the other program on P processing elements) is plotted over numbers of processing elements for the 8 and 16 equation problems, respectively.

Previous critical path analysis had shown that the value for P_{opt} was 4 PEs for the 8 equations problem, and 6 for the 16 equation problem. The data show a clear trend: when the number of processing elements used (P) was greater than or equal to P_{opt}, increasing program granularity reduced program performance. This would be due to the lengthening of the critical path caused by the increased granularity,

essentially reducing the amount of parallel code, and therefore reducing speedup, even though some communications and redundant work were eliminated. However, reducing the critical path length increased performance.

When P is less than Popt, the result is almost opposite. Although reducing the critical path length increased program speedup, increasing granularity increased program speedup even more (this effect is more noticeable in Figure 9 than in Figure 8). Also interestingly, the data indicate that when P is much less than P_{opt}, the effect of increased granularity becomes even greater, but when P is slightly less than P_{opt}, best program speedup is achieved by increasing granularity only slightly. Granularity that is too large is indicated by reduced program speedup and efficiency (as indicated by the data in Figure 9 for "g–ig4" running on 4 PEs). These numbers are difficult to determine in practice (as previously mentioned when discussing Amdahl's Law), but efficiency is found to correlate well with average processing element utilization (u), which is the average of each processing element's actual running time divided by the overall program execution time. Average processor utilization can be found relatively easily, so granularity that is too large is indicated by low average processor utilization (when $P < P_{opt}$).

Figures 2 and 3 can help illustrate the reasons for these trends. The scheduled program critical path (in Figure 3) is slightly longer than that of the data flow graph (in Figure 2), because there were not enough processing elements available. If an additional processing element was available, this problem would be alleviated. For the original program, with larger data sizes and inadequate processing elements, UpdateMtx tasks that were not in the data flow graph critical path will be a part of the scheduled program critical path. When modifications are made to reduce the data flow graph critical path, scheduled program performance improves slightly, but the main problem is the lack of processing elements. When modifications are made to increase granularity, the UpdateMtx tasks that impacted the scheduled program critical path are coalesced and become smaller, impacting the scheduled program critical path less. This effect becomes greater as the lack of processing elements becomes greater. However, when the available number of processing elements is large enough so that the data flow graph and scheduled program critical paths are essentially the same, the

only way to improve program performance is to reduce the critical path. Increasing granularity also increased the critical path length, and only served to degrade performance.

4. Programming Guidelines for Improving Parallel Program Speedup

The experimental results suggest some guidelines that can be used to improve parallel program speedup. These guidelines can be utilized for programs that have a critical path length independent of the number of processing elements used to execute the program, and a computer with a fixed number of processing elements P has been chosen to execute the program on. [Gus88] indicates that there are a classes of problems whose amount of parallel code (and therefore, critical path length) will and will not vary with the number of processing elements used. These guidelines are:

$P < P_{opt}$:

- high u: increase granularity
- low u: decrease granularity

$P \geq P_{opt}$:

- decrease critical path length

5. Summary and Future Research

We have developed a tool that performs critical path analysis for parallel programs. These programs are written as a set of procedures (with clearly defined input and output parameters) and procedure calls. This method of writing parallel programs is widespread. For example, MUPPET [MKL87], POKER [SnS86], and Polylith [PRG87] programs use this method of programming, in addition to HYPERTOOL.

The tool has been applied to various versions of a Gaussian elimination program, and guidelines for performance improvement have been empirically derived. Although Gaussian elimination is just one example, many types of programs for MIMD computers exhibit the same characteristics as the example Gaussian elimination program. These characteristics include an amount of parallel program code dependent on data size and independent of the number of processors used, the presence of a critical path that does not vary with the number of processors used, more than one type of task used, and program computations

dominant over communications.

Present computing capacity precluded running larger data sizes and simulating larger computers. We expect to be able to do so in the future. Additionally, we will continue to search for other programs and program classes to test the applicability of these guidelines. We will also continue with the development of more and better tools for visualization and performance analysis.

Acknowledgement

This work has been supported, in part, by the National Science Foundation (grant CCR–8700738).

Notes

[1] The simulator utilized is the SIMON simulator [Fuj83], which was modified to simulate a hypercube by the University of Illinois Urbana-Champaign.

[2] A more efficient, but more complex method to find the longest path would utilize heaps and Dijkstra's shortest path algorithm [Tar83]. This method of finding the longest path works because the data flow graph is a directed, acyclic graph. One merely constructs a new graph with the same nodes and edges as the data flow graph, and assigns edge weights based upon the communication cost of the corresponding parameter and the execution time of the source and/or destination tasks. The edge weights are then negated, and the shortest path algorithm used.

[3] Petri net simulation and animation tools utilized were those included in the P-NUT system [Raz87].

[4] The source code for the Gaussian elimination program is available in [WuG88].

References

[Fuj83] R.M. Fujimoto. *SIMON: A Simulator of Multicomputer Networks.* Report UCB/USD 83/140, University of California Berkeley, 1983.

[Gus88] J. F. Gustafson. *Reevaluating Amdahl's Law.* Communications of the ACM, Vol. 31, No. 5 (May 1988), pages 532–533.

[MKL87] H. Mühlenbein, O. Kräemer, F. Limburger, M. Mevenkamp, and S. Streitz. *Design and Rationale for MUPPET: A Programming Environment for Message Based Multiprocessors.* Proceedings of the First International Conference on Supercomputing. Lecture Notes in Computer Science 297. Springer-Verlag, Berlin, 1988.

[PRG87] J. Purtilo, D. A. Reed, and D. C. Grunwald. *Environments for Prototyping Parallel Algorithms.* Proceedings of the 1987 International Conference on Parallel Processing. The Pennsylvania State University Press, University Park, PA, 1987.

[Raz87] R. R. Razouk. *A Guided Tour of P-NUT (Release 2.2).* Technical Report 86–25, Department of Information and Computer Science, University of California Irvine, Irvine, CA, January 1987.

[SnS86] L. Snyder and D. Socha. *POKER on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment.* Proceedings of the 1986 International Conference on Parallel Processing. IEEE Computer Society Press, Washington, D.D., 1986.

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

[WuG88] M. Y. Wu and D. D. Gajski. *A Programming Aid for Hypercube Architectures.* Journal of Supercomputing, Vol. 2, No. 3 (1988).

[WuG88a] M. Y. Wu and D. D. Gajski. *Computer-Aided Programming for Multiprocessor Systems.* Technical Report 88–19, Department of Information and Computer Science, University of California Irvine, Irvine, CA, June 1988.

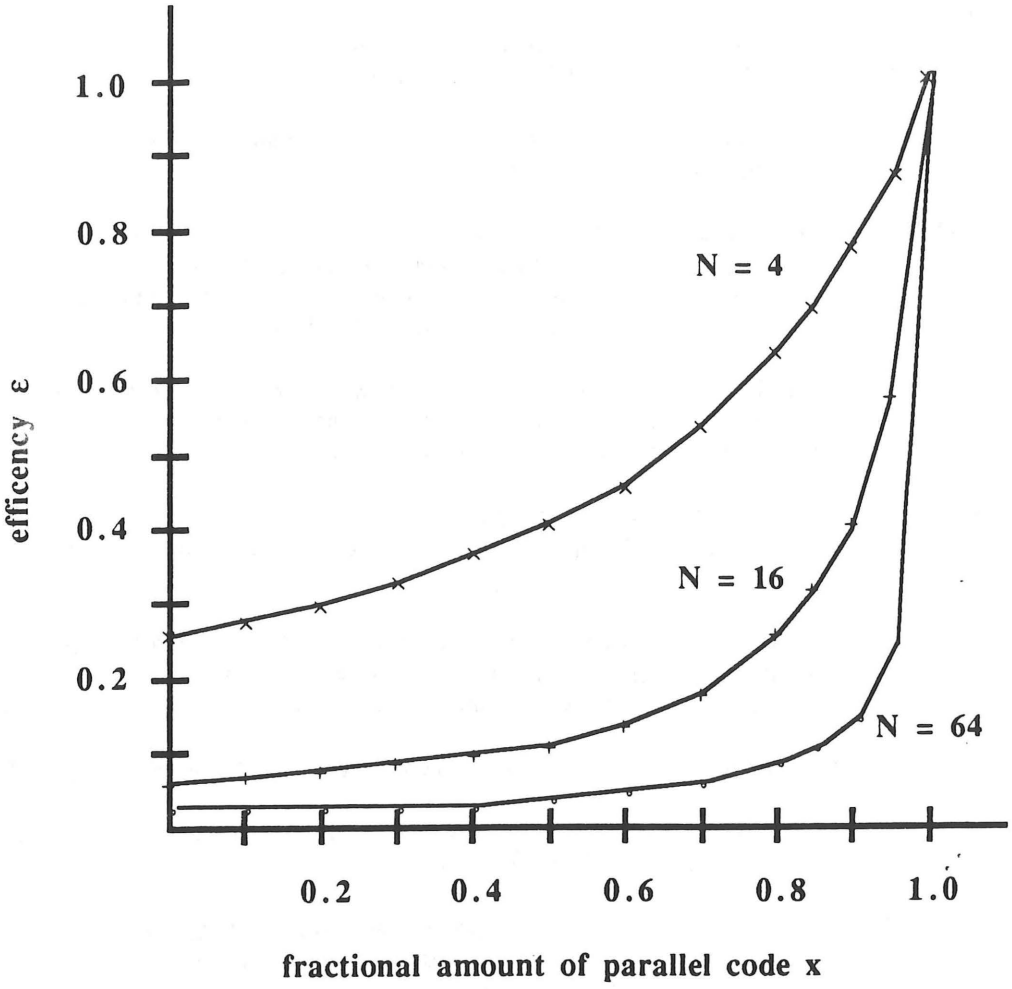
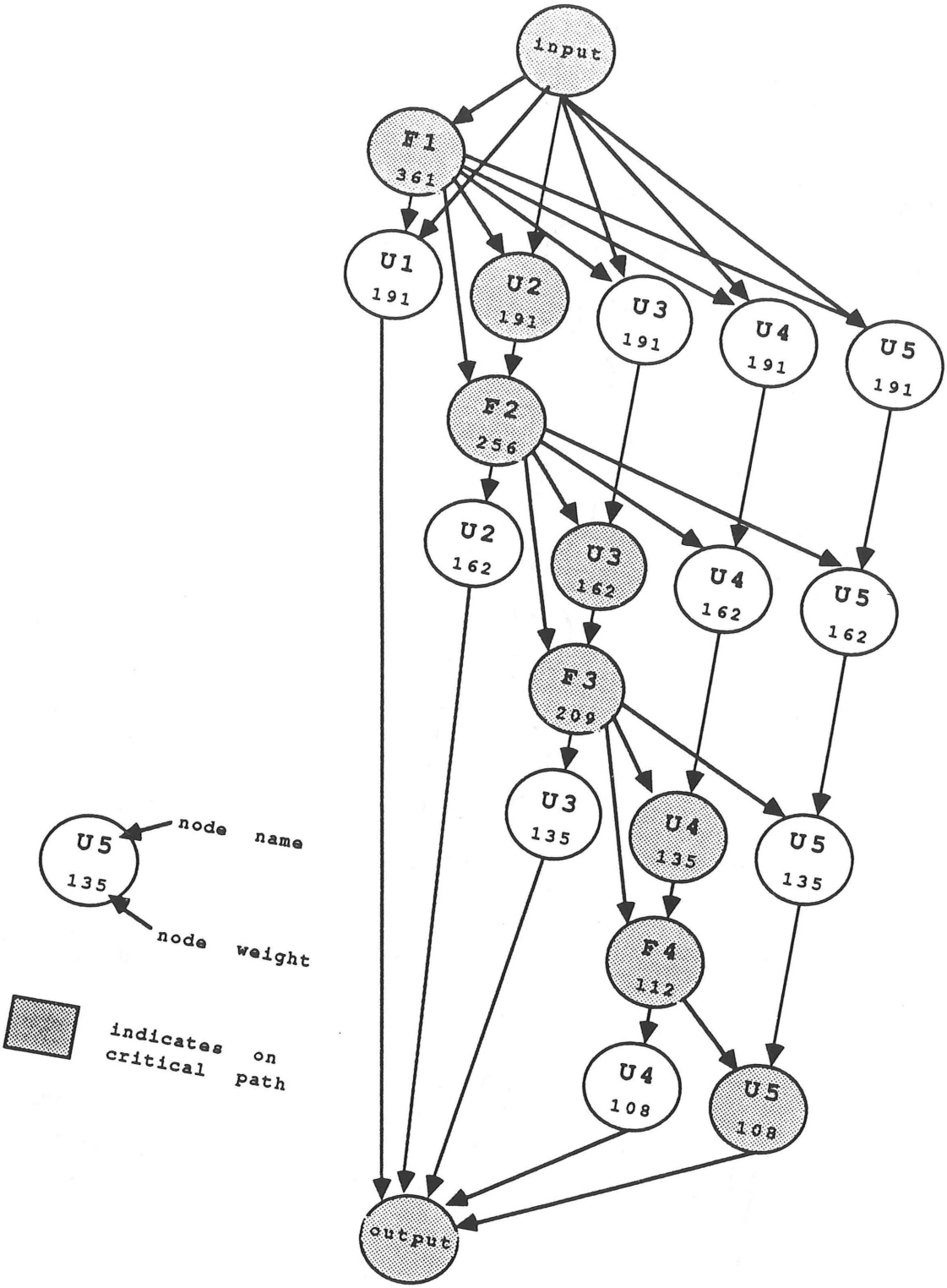
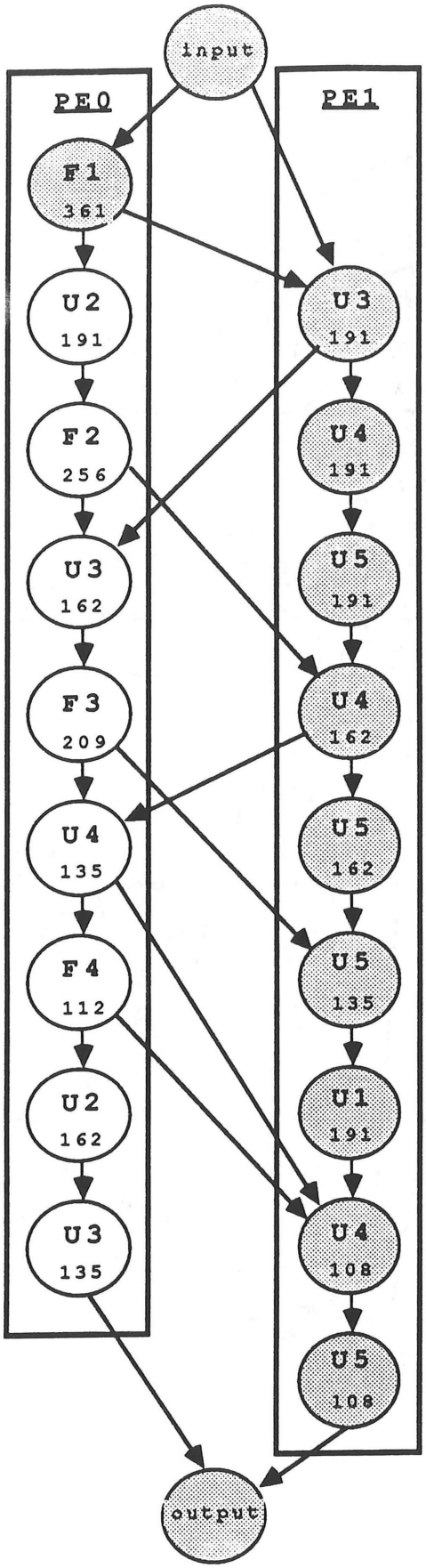

1.0
0.8
0.6
0.4
0.2
efficency ε
N = 4
N = 16
N = 64
0.2
0.4
0.6
0.8
1.0
fractional amount of parallel code x

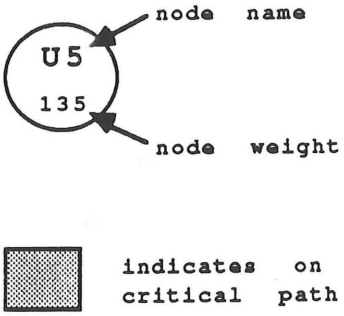
Figure 1



Data flow graph for "g" (4 equation problem).

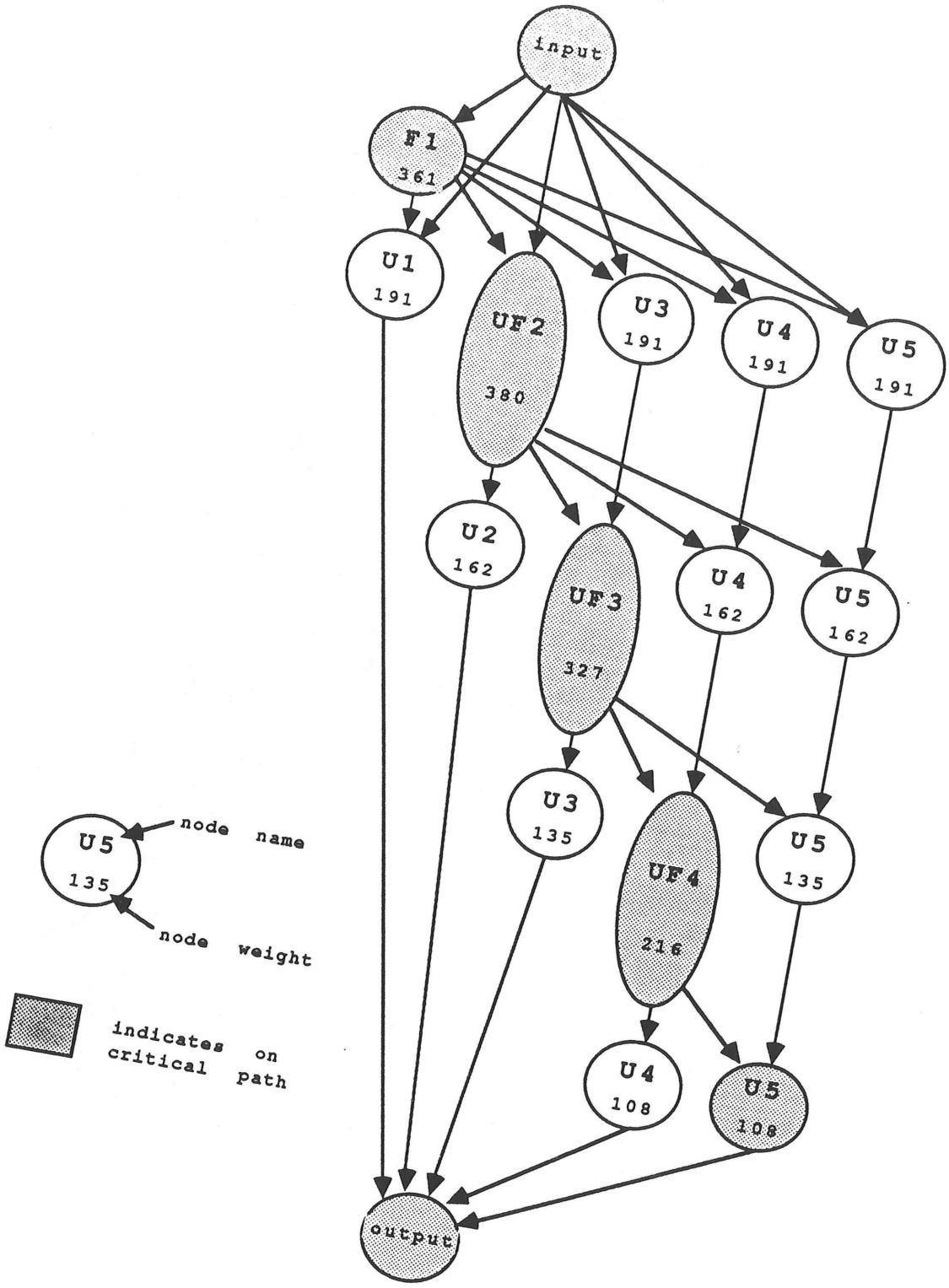
Figure 2





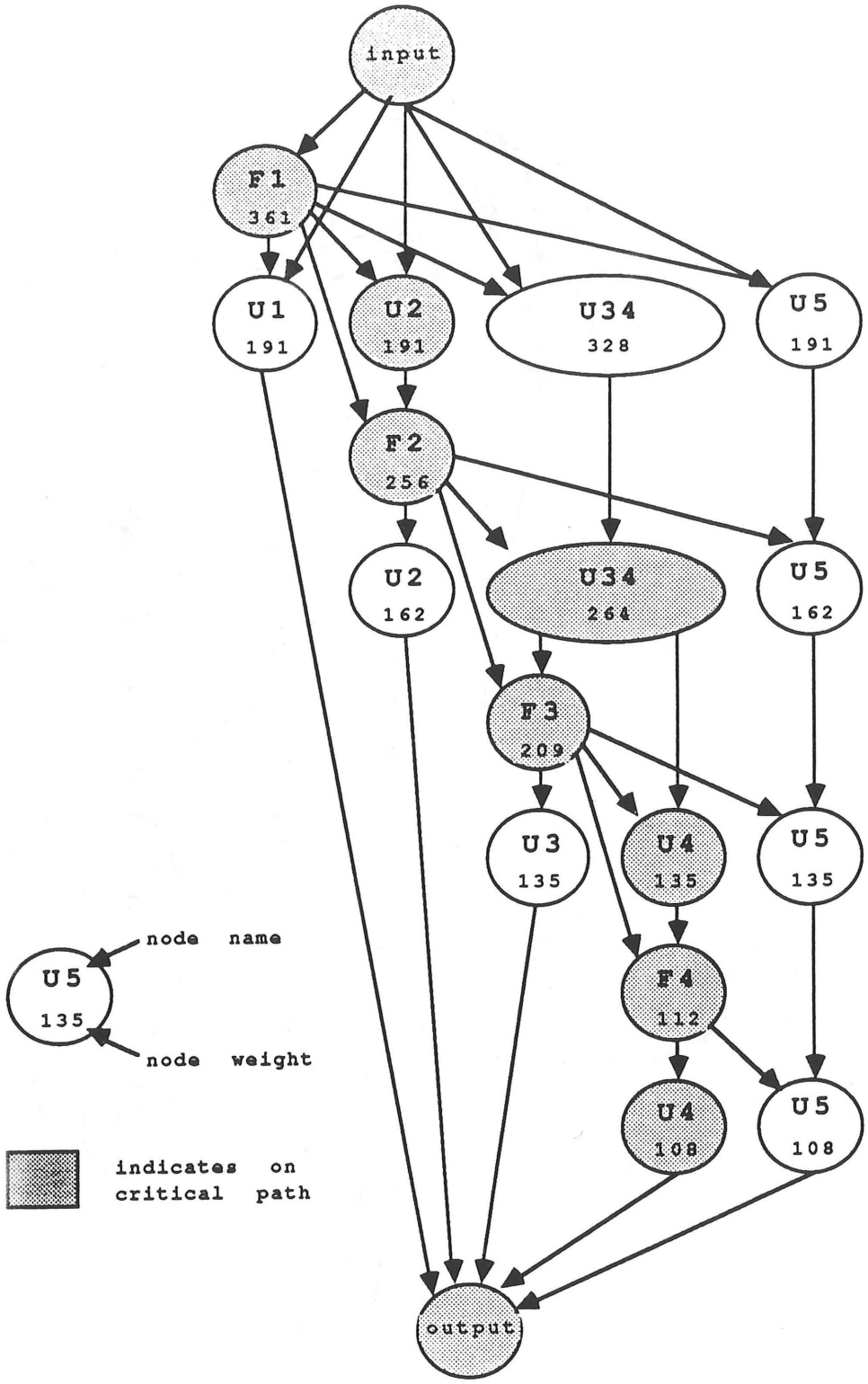
Scheduled program graph for "g" (4 equation problem, 2 processing elements).

Figure 3



Data flow graph for "ng" (4 equation problem).

Figure 4



Data flow graph for "g-ig2" (4 equation problem).

Figure 5

Data Summary: 8 equation problem

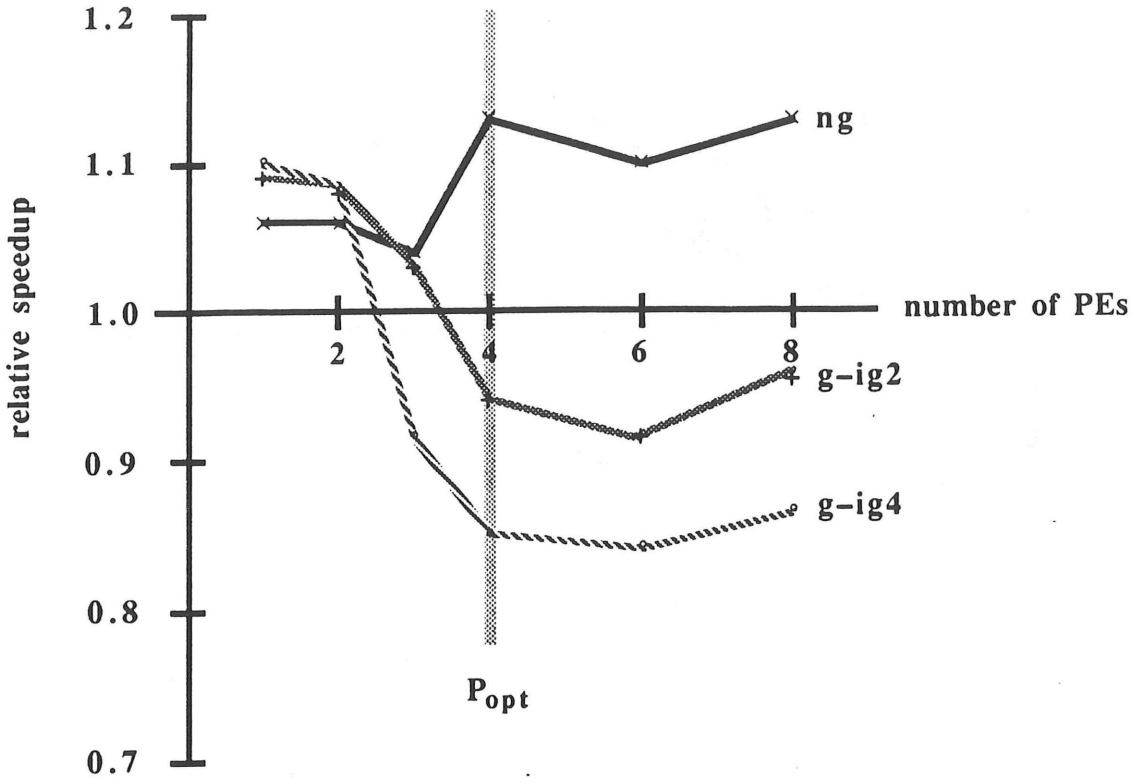
		g	ng	g-ig2	g-ig4
P=2 PEs	T_{seq} (μsec)	19,499	18,383	17,909	17,797
	T_{par} (μsec)	9,642	9,088	8,926	8,967
	S	2.02	2.02	2.01	1.98
	ε	1.01	1.01	1.00	0.992
	u	0.918	0.936	0.936	0.919
	comm	63	70	72	63
P=3	T_{seq} (μsec)	19,291	18,383	17,909	17,797
	T_{par} (μsec)	6,912	6,659	6,740	7,511
	S	2.79	2.76	2.66	2.37
	ε	0.930	0.920	0.886	0.790
	u	0.859	0.876	0.838	0.742
	comm	87	107	89	81
P=4	T_{seq} (μsec)	19,499	18,383	17,909	17,797
	T_{par} (μsec)	6,406	5,667	6,822	7,525
	S	3.04	3.24	2.63	2.37
	ε	0.761	0.811	0.656	0.591
	u	0.724	0.779	0.627	0.565
	comm	127	117	101	101
P=6	T_{seq} (μsec)	19,291	18,383	17,909	17,797
	T_{par} (μsec)	6,310	5,738	6,836	7,553
	S	3.06	3.20	2.62	2.36
	ε	0.510	0.534	0.437	0.393
	u	0.492	0.525	0.421	0.378
	comm	147	147	111	109
P=8	T_{seq} (μsec)	19,499	18,383	17,909	17,797
	T_{par} (μsec)	6,588	5,808	6,836	7,553
	S	2.96	3.17	2.62	2.36
	ε	0.370	0.396	0.327	0.295
	u	0.360	0.392	0.317	0.283
	comm	157	157	111	109

Figure 6

Data Summary: 16 equation problem

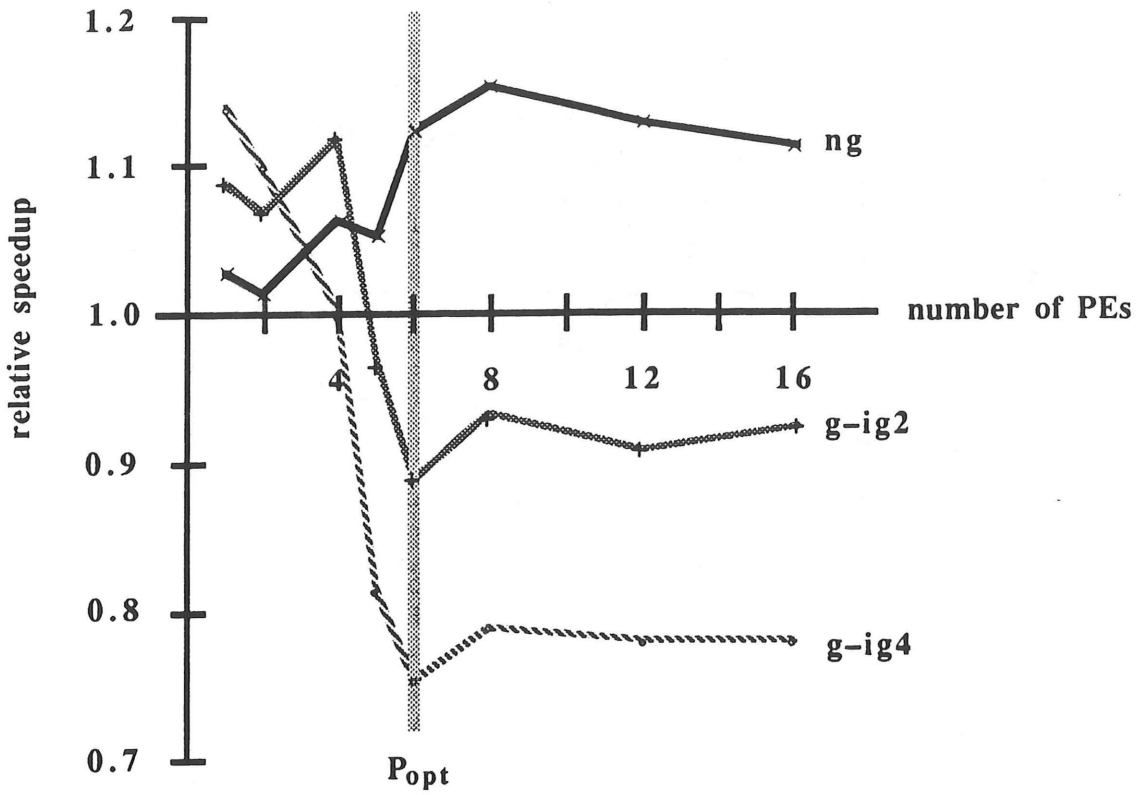
		g	ng	g-ig2	g-ig4
P=2 PEs	T_{seq} (μsec)	114,607	111,315	105,185	100,797
	T_{par} (μsec)	55,611	54,571	51,923	50,069
	S	2.06	2.04	2.03	2.01
	ε	1.03	1.02	1.01	1.01
	u	0.980	0.974	0.980	0.979
	comm	206	197	197	202
P=4	T_{seq} (μsec)	115,495	111,315	105,185	100,797
	T_{par} (μsec)	30,996	29,099	27,638	30,845
	S	3.73	3.84	3.81	3.27
	ε	0.931	0.959	0.951	0.817
	u	0.901	0.932	0.928	0.798
	comm	341	330	255	225
P=5	T_{seq} (μsec)	114,607	111,207	105,185	100,797
	T_{par} (μsec)	25,239	23,903	26,078	30,887
	S	4.54	4.65	4.03	3.26
	ε	0.908	0.930	0.807	0.653
	u	0.879	0.905	0.789	0.644
	comm	347	338	283	257
P=6	T_{seq} (μsec)	114,607	111,207	105,185	100,797
	T_{par} (μsec)	23,372	20,684	26,236	30,905
	S	4.90	5.38	4.01	3.26
	ε	0.817	0.896	0.668	0.544
	u	0.794	0.875	0.656	0.534
		381	373	313	277
P=8	T_{seq} (μsec)	115,495	111,315	105,185	100,797
	T_{par} (μsec)	24,558	21,084	26,414	30,929
	S	4.70	5.28	3.98	3.26
	ε	0.589	0.660	0.498	0.407
	u	0.578	0.652	0.491	0.401
	comm	483	467	341	285
P=12	T_{seq} (μsec)	114,607	111,207	105,185	100,797
	T_{par} (μsec)	24,076	21,396	26,428	30,929
	S	4.76	5.20	3.98	3.26
	ε	0.397	0.433	0.332	0.272
	u	0.393	0.432	0.327	0.267
	comm	545	545	351	285
P=16	T_{seq} (μsec)	114,607	111,207	105,185	100,797
	T_{par} (μsec)	24,272	21,592	26,428	30,929
	S	4.72	5.15	3.98	3.26
	ε	0.295	0.322	0.249	0.204
	u	0.294	0.322	0.246	0.200
	comm	573	573	351	285

Figure 7



Relative speedup of modified programs for 8 equations.

Figure 8



Relative speedup of modified programs for 16 equations.

Figure 9