AN INTEGRATED SCHEDULING STRATEGY FOR MULTIPROCESSOR SYSTEMS

Mandar Joshi

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-24 July 1991

Table of Contents

List of Tables

List of Figures

Abstract

Allocating resources to minimize the execution time of a parallel program is an NP-complete problem. Neither of the two major strategies, static scheduling and dynamic scheduling, is a complete solution by itself. Static scheduling suffers from the unavailability of accurate information about the execution behavior of the program; while dynamic scheduling which use the run-time information may incur serious overheads unless effectively managed. The strategy presented here combines both static and dynamic scheduling. A schedule generated by a static scheduling algorithm is used as initial assignment. A simple dynamic scheduling algorithm is then applied to compensate for inaccuracies in the initial information as they become visible during the execution of the program Beginning with a good static schedule minimize the amount of work which must be done at run-time by the dynamic scheduler. The study presented here identifies the factors which affect the performance of scheduling algorithms and indicates that the integrated strategy can provide substantial improvement over either static or dynamic scheduling used separately.

Keywords : Multiprocessor scheduling, performance measurement.

Chapter 1

Introduction

Multiprocessor systems built from low cost, high performance microprocessors are a promising method for development of low cost, high performance parallel supercomputers. These multiprocessor systems can be used either to maximize the throughput of a given system by assigning different processors to different jobs or to maximize the speed up for a single job. Our focus is on attainment of the second goal. The task is to take a given parallel program and to allocate resources to the parallel program in such a way as to minimize the execution time of the program on a given multiprocessor architecture.

1.1 Problem Statement

A brief summary statement of the problem is as follows - given a parallel program

- partition it into a set of schedulable units of computation, and
- create a schedule for allocation of resources to these units of computation and the communication among them such that the total time of execution of the parallel program on the given architecture is minimized.

A parallel computation can be represented as a directed graph where each of the nodes represents a unit of computation and each edge represents

a dependency between two units of computation. Each node of the graph is called a schedulable unit of computation because it can, once its conditions for execution are met, be scheduled for execution on an appropriately chosen processor. Each edge represents a communication among nodes which for its execution, must be assigned to some communication resource.

The target architecture for execution of the parallel program can be represented by an undirected graph where each node represents a processor and each edge represents a communication link between two or more processors. We can, without loss of generality, consider each processor to also incorporate local memory. This model of target architectures spans all types of multiprocessor systems ranging from closely coupled systems of multiple processors sharing a common memory to distributed systems, which consist of a set of distinct computers connected by communication channels. The problem is to assign units of computation to processors and communications between units of computation to communication resources such that the total time to execute the computation on the given target architecture is minimized. The scheduling problem now becomes a mapping problem between the dependency graph representing the parallel computation and the graph representing the target architecture.

It has been shown[10] that the general graph mapping problem is equivalent to the graph isomorphism problem. Even more restricted versions of this problem are known to be NP-complete. Therefore, we typically must rely on heuristics which are known to be sufficiently accurate as well as computationally feasible.

1.2 Approach

Exact optimal scheduling algorithms typically require exact information concerning the execution time of each unit of computation, the amount of information exchanged and which paths through the dependency graph are executed. This information is not usually available, therefore, scheduling must be based upon estimates of these properties which may contain a little or a great deal of uncertainty. Furthermore, the computation graph may be a dynamic graph, i.e. it may dynamically add nodes or choose only a few paths in the dependency graph sepcification.

It is typically the case, though, that a substantial amount of information concerning the execution properties of the units of computation and the dependency graph in general, are available at the time the program is compiled and loaded. There has, therefore, been a considerable amount of effort applied to the development of static scheduling algorithms which utilize this information. These algorithms offer a substantial improvement over random assignments which do not utilize this information.

A good deal of time and energy has also been applied to the study of dynamic scheduling algorithms. In this family of scheduling algorithms the assignment of resources to tasks is modified at run time as the state of the system and the computation becomes more accurately resolved. These dynamic scheduling algorithms have also been found to have benefit in some circumstances.

This leads us to consider the possibility of integrating static and dynamic scheduling into a single system. A static schedule based upon the best information available at load time will often be a much more effective initial assignment of resources than a random assignment. Simple dynamic schedul-

ing, correcting inaccuracies due to the incomplete information available to the static scheduling algorithm, should lead to additional improvement.

The static algorithm used in this study is a technique called Linear Clustering [10]. The Linear Clustering algorithm is based upon a series of graph transformations. The end point of these transformations is a graph which is called the "Virtual Architecture Graph" or VAG. A VAG represents a processor configuration which, if realized, would be an optimal execution environment for the given program graph. The Virtual Architecture Graph is then mapped upon the target physical architecture. A simple dynamic scheduling algorithm in which processors request additional schedulable units of computation (SUC) when they exhaust their supply of SUCs is coupled with the static scheduler.

1.3 Results

The scheduling strategy described above was applied to programs generated with the CODE [5] parallel programming environment. The CODE system allows users to generate parallel programs and stores them in the form of a dependency graph. This stored dependency graph is then used to generate the actual code for different target architectures.

The static scheduling algorithm operates on the CODE graph and generates an initial schedule. The CODE parallel program generator uses that information while generating the program. The information required to support dynamic scheduling is also inserted into the executable code at this stage. The generated code is then compiled and run on the target architecture. The dynamic scheduling algorithm is invoked as needed during execution.

The results of this thesis are a determination of the factors which lead to this integrated scheduling algorithm producing an improvement over either

static or dynamic scheduling applied by itself.

1.4 Organization of the thesis

Chapter 2 gives an overview of the algorithms and methods available to be combined into an integrated static/dynamic scheduling system.

Chapter 3 describes the integrated strategy for combining static and dynamic scheduling in some detail.

Chapter 4 defines and describes the execution environment in which the integrated algorithm was tested. It also defines the experiments and gives the parameters driving the experiments, and the metrics measured in the experiments.

Chapter 5 draws conclusion from the performance studies and describes the possibilities for future work.

Chapter 2

Related work

Multiprocessor scheduling has been studied extensively for more than twenty years. Since the general multiprocessor scheduling problem is NP complete, most research has been focused upon finding heuristics which are fast but offer effective and satisfactory solutions in some domain of application. A fairly thorough survey of the literature previous to 1988 can be found in Kim [10]. The survey here lists the several approaches which were analyzed with respect to their applicability in a search for appropriate static and dynamic scheduling strategies for our integrated static/dynamic strategy.

2.1 Static Scheduling Strategies

Static scheduling algorithms are applied at compile time and/or load time to determine an "optimal" allocation of resources to minimize the execution of a given parallel program. A wide spectrum of cost functions have been used. Cost functions typically depend upon different parameters and use different metrics. There follows a brief description of types of strategies based upon the model used for representing the scheduling problem.

2.1.1 Graph Theoretic Approaches

Network flow methods [21] model the scheduling problem as finding a min-cut in a graph where the processors are synchronization nodes and the

other nodes are program modules. In communication graph based models the cost of communication is reduced by the use of graph embedding methods. Precedence graph methods model computations as data flow graphs, and determine the minimum amount of resources necessary to execute the data flow graph. Data flow graphs have also been extended to include communications costs between the modules. We have chosen an extended data flow graph model as the basis for our static scheduling algorithm. There are, however, a number of other possible methods.[18, 17, 15]

2.1.2 Simulated Annealing Methods

Simulated annealing algorithms begin with a random initial mapping and fine tune the solution for some specific cost function [20, 13]. These methods typically execute by pairwise permutation of assignments of units of computation to processors. The fundamental approach here is to begin with any initial assignment and try to find a better assignment.

2.1.3 Heuristic search

These methods involve a state space search. A search tree can be generated to represent partial assignment of modules to processors [1]. A cost function which calculates incurred costs along the path is used to determine whether the next node is to be expanded or discarded. The algorithm finds the best solution through an exhaustive search using a form of branch and bound.

2.1.4 Miscellaneous

Many other methods have been used including integer programming, iterative optimization of multiple subgoals, optimization of approximate mea-

sures of mapping, etc.[14, 23]

All of these methods depend upon the assumption that the execution profile of a program is known at the time the schedule is computed. It is not possible to determine accurate execution properties of arbitrary programs except in restricted cases.

It is also the case that the types of computation structures to which these methods can be applied is somewhat restricted. None of them apply to the most general case where the active paths in the dependency graph specification are not known until execution time. In addition, the complexity of application typically becomes much higher as the structure of the graph, is enriched from trees to directed acyclic graphs to general graphs.

We have chosen to use an extended data flow graph model developed by Kim [10] as the basis for our static scheduling. This algorithm, called Linear Clustering, is described in detail in the next chapter.

2.2 Dynamic Scheduling Strategies

Dynamic scheduling strategies can be classified as *load balancing* policies [2, 16] or *load sharing* policies [6, 11, 7]. Load balancing policies attempt to distribute the work evenly among the available processors while load sharing policies try to keep all processors busy. The justification for these strategies is that if all resources are kept fully utilized without introduction of extra work, then the execution time will be approximately minimized. There are a number of attributes which can be chosen in the construction of dynamic scheduling policies.

- Preemptive or Non-Preemptive schedules : With a non-preemptive policy a task is not allowed to move if it has already initiated its execution. A preemptive policy allows the rescheduling of an existing process where the conditions justify.

- Sender initiated or Receiver initiated : The reassignment of work can either be initiated by a processor when its load falls below a certain threshold or it can be initiated by a processor if its load increases beyond a certain threshold.

- Centralized or Distributed state : In a centralized state approach, work distribution is controlled from a central site. In a distributed approach, control over distribution of work is shared among all resources.

Dynamic strategies also work on partial information and introduce extra overhead at run time. It is necessary to choose a dynamic scheduling policy which introduces minimal overhead. We choose as our dynamic algorithm a very simple algorithm based upon empirical studies executed in the context of distributed operating systems.

Chapter 3

An Integrated Scheduling Strategy

Previous research has established that while both static scheduling strategies and dynamic scheduling strategies can generate improvement in execution time of a parallel program upon a multiprocessor architecture and that neither by itself is a complete solution. The strategy presented here integrates static and dynamic scheduling algorithms and determines the conditions under which this integration provides substantial advantage over either applied separately. A static schedule is applied to generate an initial assignment of resources to tasks. This assignment will be near optimal for a static graph to which the method applies and for which the virtual architecture graph can be realized and if all the execution properties are exactly known. A simple dynamic algorithm which has been shown to be effective in the distributed operating system environment is then applied to compensate for the inaccuracies in the information upon which the static schedule is based as they become visible through the execution of the program under the static schedule.

3.1 Justification for the Integrated Approach

The static scheduling algorithms assume that the execution times of the units of computation are fixed, and known *a priori* and that the use of communication resources is continuous and even throughout the execution of the program. These assumptions are rarely valid in practice after mapping to a real target architecture. The factors which affect these assumptions include

the following:

- Input data set : The execution time and the amount of communication among the units of computation are typically dependent upon the input data of the program. Execution properties may not be known for the specific input data for a particular execution.

- Resource contention : In particular, the use of communication channels is typically not uniform across the entire execution of a unit of computation. It is typically the case that communications take place at the beginning and end of the execution of a unit of computation. In addition, it is commonly the case that the Virtual Architecture Graph for a program cannot be realized directly without conflict on the real target architecture. These effects may increase communication times by rendering the assumptions under which the static schedule is generated to be invalid.

- System-dependent functions : System functions such as the spawning and termination of processes may also affect the execution time of the units of computation executing on a specific processor. These system functions are typically not incorporated in the computation graph but may take significant execution time and engender substantial communications.

These problems can be partially overcome by collecting information on runs of the program over a spectrum of input data and execution environments, and incorporating this information into the computation graph of the algorithm. It is, however, often not a practical means of improving static schedules. The

complexity of parameter structures and the amount of information which must be incorporated into the computation graph and into the algorithm renders use of extensive measurements impractical.

Dynamic scheduling strategies can accommodate this dynamically occurring uncertainty in program and system state. There are, however, a number of overheads associated with dynamic scheduling. Dynamic algorithms must gather state information and must execute scheduling algorithms at run time. In addition, execution of the decisions rendered by the dynamic scheduling algorithms may require substantial resources both in communication time and processing time.

The poorer the starting point for a dynamic scheduling algorithm the more often it must be run and the more complex and expensive implementing its decisions becomes.

Therefore, it seems intuitive that beginning a computation with the best possible static schedule which can be determined on the basis of existing information and accommodating for the deficiencies in resource usage information with dynamic scheduling offers some, and perhaps substantial advantage, for execution of complex parallel programs.

3.2 The Static Scheduling Model

The static scheduling model chosen here is based upon a specific model of computation in which the computation and the architecture are both modeled as graphs.

3.2.1 The Model of Computation

The model of computation is a triple (G_c, f_c^{comp}, f_c^{comm}), where

- G_c is a graph which specifies the computation. G_c is a directed graph and composed of the following:

  1. A node set N_c : Each node in N_c represents a schedulable unit of computation (SUC). Each SUC consists of a set of input dependencies, a set of output dependencies and a specification of the computation bound to the node. Each SUC requires all of its input dependencies to be satisfied before its completion. It is also required to satisfy all of its output dependencies before its completion. The computation bound to the node can be specified as arithmetic expressions, or a collection of statements or procedures in a high level language. It may also be realized as a collection of smaller SUCs.

  2. An Edge set E_c : Each edge in E_c, e = (n_i,n_j), $n_i, n_j \epsilon N_c$, is directed from SUC n_i to SUC n_j. Each dependency e, represents a dependency relation between SUCs n_i and n_j. Each dependency relation is realized by synchronization and data communication. The data communication is based on an asynchronous send/synchronous receive communication model.

- f_c^{comp} is a function which maps every SUC in N_c into a non-negative floating point number which is the expected execution time of the SUC.

- f_c^{comm} is a function which maps each edge e = (n_i, n_j) in Ec into a non-negative floating point number which is the expected time required for transfer of information from n_i to n_j.

3.2.2 Model of Target Architecture

The target architecture model is also a triple (G_a, f_a^{comp}, f_a^{comm}), where

- G_a is an undirected graph defined as follows:
  1. A set of processor nodes N_a, in which each processor node represents a processor with its local memory.
  2. An edge set E_a, where all edges are undirected. Each edge e = (n_i, n_j) represents a communication link between two processor modules n_i and n_j.
- f_a^{comp} maps each processor module in N_a into a pair of positive floating point numbers which denotes the computing power of the node relative to other nodes and the current local memory size.
- f_a^{comm} maps each edge (n_i, n_j) in E_a into a positive floating point number which represents the bandwidth of the communication link from processor node n_i to processor node n_j.

This model can be used to model all types of architecture. e.g., in shared memory architectures, the common global memory can be modeled as a dummy processor node which is fully connected with all other processor nodes.

3.2.3 Linear Clustering Algorithm

The static scheduling algorithm employed in this research is the linear clustering algorithm developed by Kim [10]. Multiprocessor scheduling is developed as a series of mappings from a computation graph to a Virtual Architecture Graph. The Virtual Architecture Graph explicitly displays all of the

parallelism which exists in the original computation graph and incorporates in its topology the communications among this maximally parallel set of units of computation. This virtual architecture graph must then be mapped to a physical target architecture graph.

A number of restrictions must be added to the original model of computation before the linear clustering algorithm can be applied. These restrictions include:

- Sets of edges entering and leaving a node may not be joined by an "OR" condition. This prevents the occurrence of nondeterminism in the computation graph.
- The computation graph may not have any back edges. This requires all loops in the original computation graph to be unrolled so that the computation graph is cycle free. An algorithm is given in [10] to unroll loops in procedural programs.
- The computation graphs must be static.
- The computation graph must have one root node and one sink node. This restriction can be met by adding additional nodes and connecting them appropriately to graphs which do not meet this requirement.

These restrictions may require selection of a particular path for a schedule from among the family of paths possible in the computation graphs. We begin by introducing the notion of nodes in the computation graph which are sequentially strongly dependent upon one another. A node N is sequentially strongly dependent node on node M, if N cannot execute until all output dependencies of M are satisfied, i.e. node N cannot begin its execution until

node M has completed its execution. A Linear Cluster is a connected subgraph of the computation graph. It is a sequence of nodes which are sequentially strongly dependent. The underlying idea of linear clustering is that nodes which are sequentially strongly dependent on each other can be assigned to the same processor without loss of parallelism. A path $(n_1,e_1,n_2,e_2,\ldots,e_{l-1},n_l)$ in G_c is called the *longest path* if it minimizes the following function:

$$\sum_{i=1}^{l-1}(T_{comp_i} + (T_{comm_i} + K_i \sum T_{comm^i_{adj}})) \tag{3.1}$$

where T_{comp_i} is the computation time for node n_i, T_{comm_i} is the communication time of n_i with its node n_{adj}, and $T_{comm^i_{adj}}$ is the communication time of n_{adj} with its neighbors other than n_i. K_i is scaling factor which is dependent on the target architecture. The linear clustering algorithm works by successively finding the longest path in G_c and cutting the subgraph defined by this longest path from G_c and replacing it by a single node.

Each of the longest paths is a linear cluster. The new computation graph generated after linear clustering can be further reduced by merging linear clusters that cannot be run in parallel. Two linear clusters, L_1 and L_2 can be merged only if L_2 may start only after L_1 has finished or L_2 can run only when L_1 is idle. This cluster merging further contributes to balancing the work-load and yields further reduction in interprocessor communications overhead. This new computation graph is the Virtual Architecture Graph which will be mapped to the target architecture graph. This Virtual Architecture Graph, if realized as an optimal multiprocessor system, will provide a processor for each linear cluster so that the mutually independent scheduled units of computation can be run in parallel as long as is necessary. The real target architecture may not have a sufficient number of processors available. In this case the clusters are assigned to processors using a secondary measure such as load balancing

based upon the available knowledge of execution time for the clusters. The real target architecture may, in addition, not incorporate a conflict free, direct communication channel between each processor. This may lead to contention and thus render invalid the optimality of the static schedule.

The total execution time of the computation cannot be less than the length of the critical path regardless of the amount of hardware resources available. Therefore, if the virtual architecture graph can be realized on the real target architecture, Linear Clustering makes it possible to achieve the minimum possible execution time for the given computation. Clusters which cannot be run in parallel are assigned to the same processors. This minimizes interprocess communication overhead and resources are used to their full potential for this computation. Gerasoulis and Yang [8] have shown that, for any nonlinear clustering of a coarse grain directed acyclic graph, there exists a Linear Clustering with equal or less execution time. Thus, Linear Clustering provides optimal mapping for the restricted class of computations to which it applies.

3.3 The Dynamic Algorithm

Selection of a dynamic scheduling algorithm must incorporate several different factors. These factors include:

- The amount of overhead : The amount of overhead is dependent upon the nature of the algorithm and the nature of the target system upon which the dynamic algorithm is implemented. It depends upon the frequency of execution of the algorithm, the costs to execute the decision cycle of the algorithm and upon the cost of implementing a given scheduling decision.

- Penalties for poor decisions : It is still the case that complete information and execution behavior is not available to the run-time system. Rather one can tell only that the assumptions of the static algorithm have been violated. Therefore, poor decisions will occasionally be made. The penalty for poor decisions can lead to an excessive cost for execution.

- Potential instability : Dynamic algorithms may be sensitive to system behavior. Decisions to move a process may be taken frequently, thus, at the worst case, the system may consume most of its resources by making poor decisions and may result in processor thrashing in an attempt to correct the poor decisions.

Chapter 4

Experimental Study

4.1 Experimental Setup

To evaluate the effectiveness of our strategy, some experimental programs were generated in the CODE parallel programming system and were run on Symult S2010 multiprocessor system. Our experiments were significantly influenced by the experimental setup we had available. The following sections describe the experimental setup.

4.1.1 CODE programming system

CODE (Computation Oriented Display Environment) is a graphical parallel programming system for writing parallel programs independent of target architecture and higher level programming language[5]. CODE is based on the Unified Computation Graph (UCG) model of parallel computation, where a parallel program is viewed as an extended directed graph [4, 19]. Each program graph consists of the following:

- Computation units : Each node in a UCG is composed of a functionality and a firing rule. A computation unit's functionality is the transformation of input dependencies to output dependencies. It can be specified in several sequential high level languages. A firing rule specifies the state of input dependencies which may enable this unit to execute. Each node may be a simple unit of computation or a subgraph.

- Dependency relations : Dependency relations compose computation units into the program structure and are represented as arcs in the program graph.

Since our model is simpler and more restrictive than the CODE model, it can be viewed as a subset of the CODE model.

A CODE program can be transformed into an executable program for any particular target architecture. This is accomplished by a "backend" program. The CODE backend for the Symult was modified to facilitate generation of programs according to static schedules. The modified backend program generates a schedule using the CODE graph and maps it onto the target architecture. This allows us to maintain architectural independence. Executable processes are generated according to the schedule. These processes can be executed on the target architecture after compilation.

4.1.2 Symult S2010 System

Symult S2010 system is an MIMD (Multiple Instruction Multiple Data) multiprocessor system composed of processors which communicate through a high-speed GigaLink network [22]. The description which follows describes the particular system used for running the experiments. This system consists of 24 nodes. Each node consists of Motorola MC68020 microprocessor operating at 25 MHz, a Motorola 68881 floating-point unit, RAM and an automatic message routing device. Typical performance of each node is 4.0 MIPS. 16 of the 24 processor system have 1 MB RAM, and rest of the nodes have 3 MB memory. The operating system is loaded into and executes in the portion of this memory. Individual processes with associated data, stacks and messages may utilize the remaining memory on each processor. The Symult system can

be accessed through a front-end Sun workstation and runs a Reactive Kernel on each of the nodes. The nodes are connected in a 6x4 mesh configuration. User processes can access files on a disk connected to any S2010 node or from the file system of the front-end.

4.2 Choice of Test Set

The properties of programs which can potentially benefit from the integration of static and dynamic scheduling were identified. Programs containing these characteristics in parametric form were prepared and used as the basis for experimental validation of the goodness of the integrated static/dynamic approach to multiprocessor scheduling.

The goal of our experimental study was to determine a few parameters from the parameters which affect the performance of our scheduling strategy. Some of the parameters can be determined by a straightforward analysis, some were determined after experimentation. e.g., Static or dynamic scheduling is ineffective for graphs which have a large number of nodes at the same level, and have small clusters with small differences in their individual execution times. In such cases, even random allocation gives sufficiently good results. Figure 4.1 shows such an example.

The workload factors which determine effectivenss of static and dynamic scheduling strategies include : certainty in the estimates of execution time and communication volume for the units of computation and the variation in execution time among the units of computation. A large variation in execution time renders static scheduling invalid. Dynamic scheduling can be used profitably only when there is a substantial variation among the execution times of the units of computation. These factors were the parameters varied in

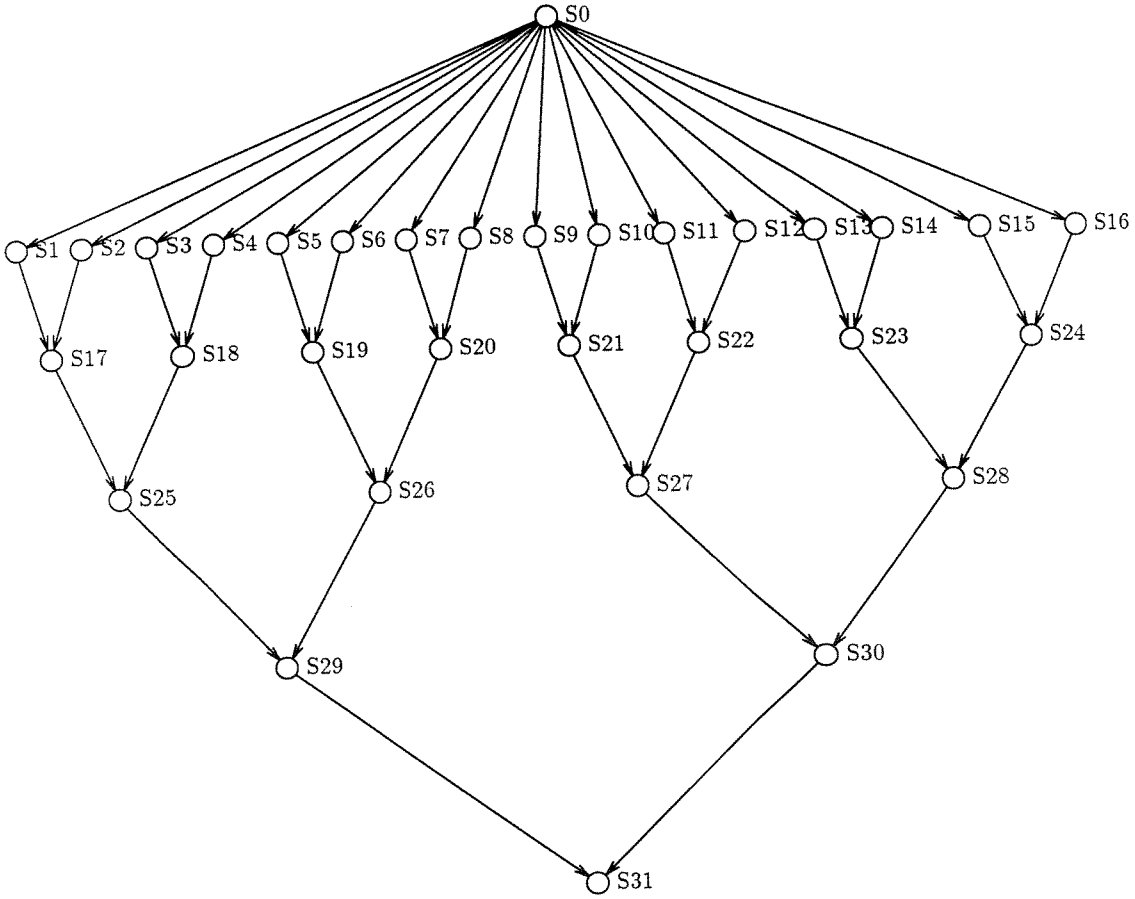

Figure 4.1: CODE graph for prime number generation

the studies executed for this thesis. The percentage of nodes whose execution was taken to be uncertain was varied from 10% to 100%. The range of variation in execution time was also varied from 10% to 100%.

There were some additional constraints added because of the target architecture. Since the processes created on Symult system are "heavyweight" processes and consume a lot of memory, very few processes can be spawned on a node at a time, although the operating system allows up to sixteen processes. Because of this reason, only the eight nodes with 3 MB memory were used for our experiments. Even then number of processes that can be spawned at a given time are restricted. To get around this problem, server processes on each node were introduced. These processes kept track of executing processes and corresponding messages in addition to performing dynamic scheduling. In the S2010 system, the individual clocks of the processors are not synchronized. To measure the execution time of the entire program, a simple addition of execution times of individual units is not sufficient. A host process was introduced to synchronize the execution of individual processes and measure the timing. The results of timing measurements were compared with random allocation of SUCs and static allocation of SUCs.

4.3 Results

Three synthetic random graphs were used as our test set. These graphs are derived from the real applications such as weather forecasting[15] and a molecular dynamics code[10]. The code inside each of the nodes is parameterized to vary execution times. All graphs were run on Symult S2010 system using random scheduling, static scheduling and dynamic scheduling. The execution times of the units were varied to determine its effect on the

execution times. Static scheduling always outperformed random scheduling at least by a factor of 2. This circumstance can be attributed to the large overhead associated with spawning a process on a Symult node. Hence, our integrated strategy was compared with static scheduing only.

The speedup obtained using the strategy is shown simply a ratio of execution time using static scheduling only and execution time using static and dynamic scheduling.

4.3.1 Graph G41

This graph is a slightly modified version of a synthetic program used by Kim[10] (Fig 4.2). This graph contains a large number of clusters with variation in execution times of individual clusters. The following table contains the clustering information and Figure 4.3 shows the VAG generated by static scheduling algorithm.

Figure 4.4 shows that the speedup for the integrated strategy is generally increasing as percentage of uncertainty is increased. The speedup obtained depends on the nodes chosen to vary the execution time and whether the variation is positive or negative. This can result in fluctuations as seen in Figure 4.4. For smaller variations, the uncertainty is not enough to offset the overhead of dynamic scheduling, which results in negative speedup.

4.3.2 Graph G82

Graph G82 contains two copies of G41. Assignment of nodes to clusters is similar to the assignment for G41. Figure 4.5 shows VAG of G82. Even though this graph contains more number of clusters, the speedup obtained is not significantly larger than speedup obtained with G41. The main reason

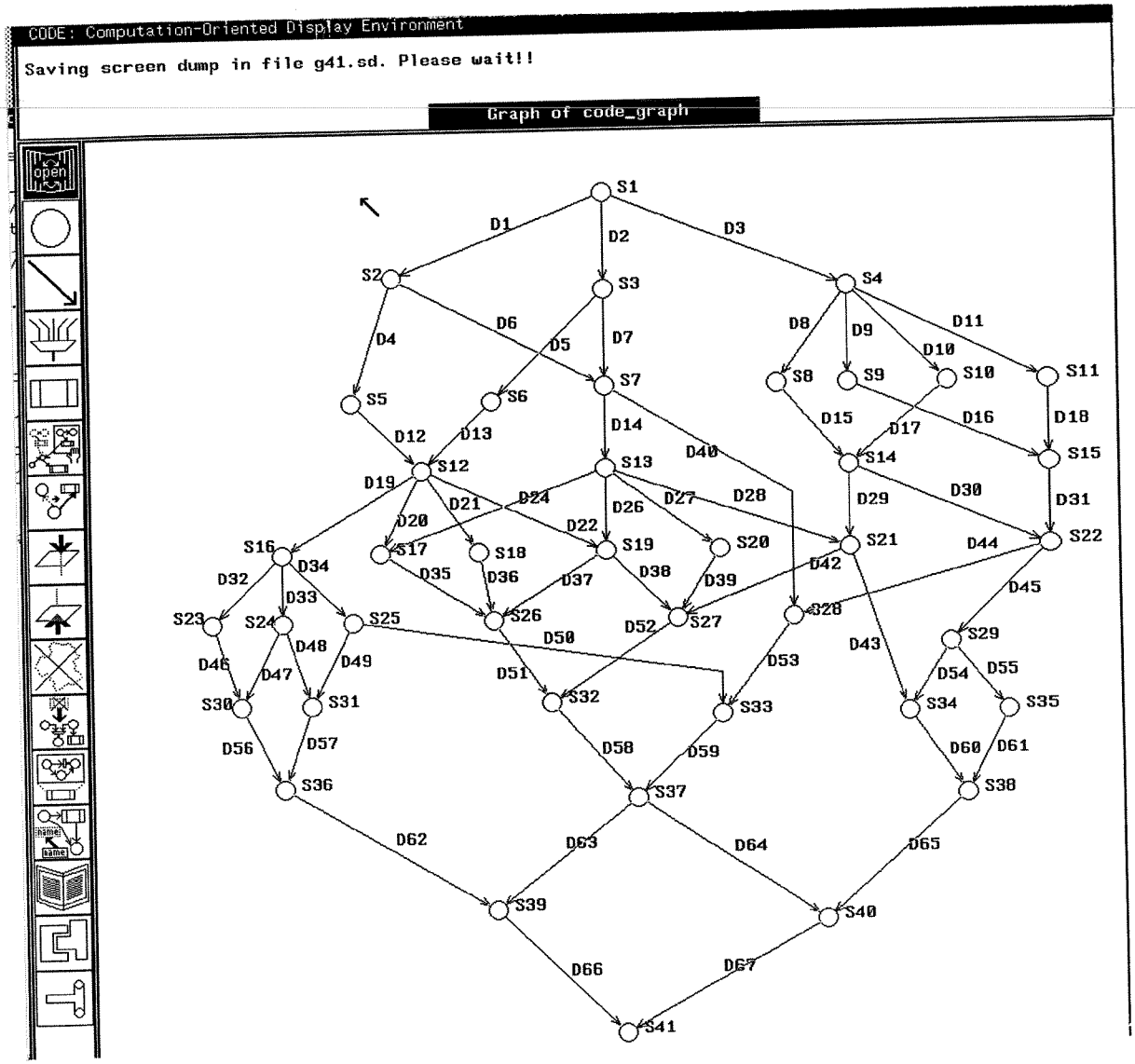

Figure 4.2: CODE graph representation of G41

Cluster	Nodes	Cluster	Nodes
C0	1,4,11,15,22,29,35,38,40,41	C7	23
C1	28,33,37,39	C8	7,13,20
C2	3,6,12,16,25,31,36	C9	18
C3	34	C10	17
C4	10,14,21,27,32	C11	9
C5	24,30	C12	8
C6	19,26	C13	2,5

Table 4.1: Assignment of nodes to clusters in G41

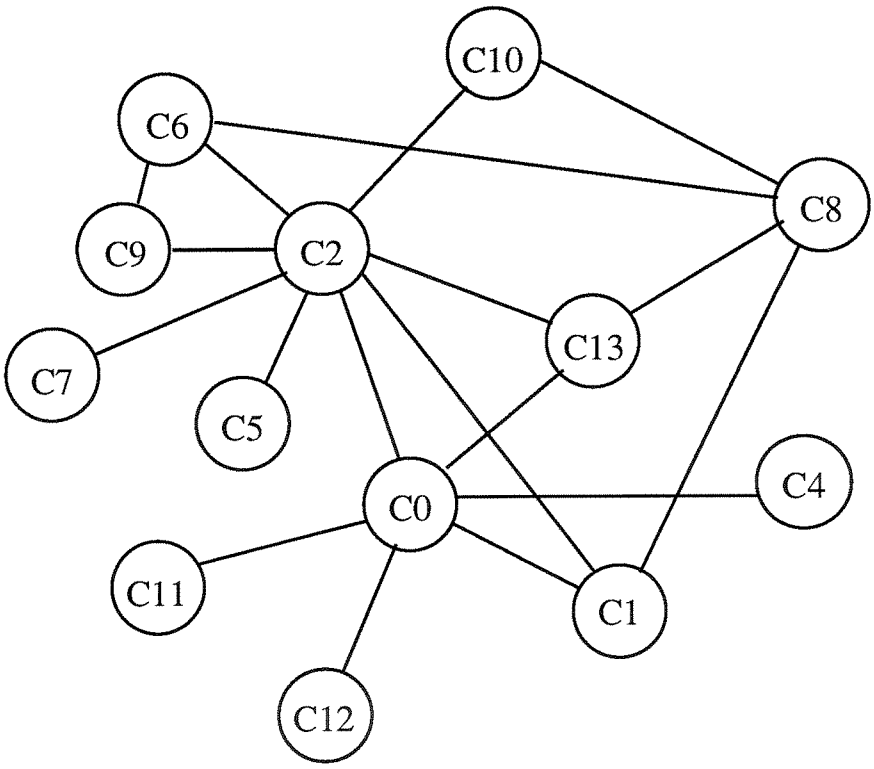

Figure 4.3: VAG of G41 after Linear Clustering

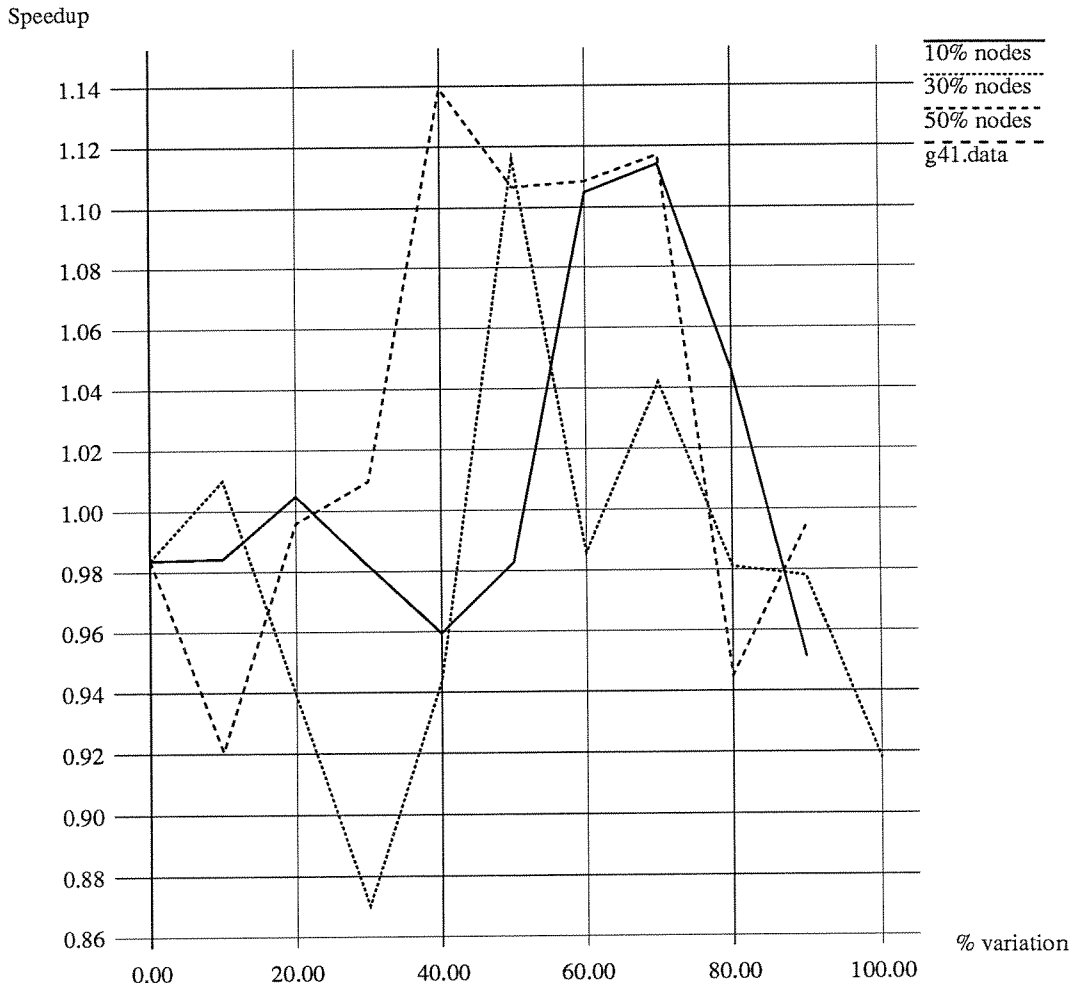

Figure 4.4: Performance results for G41

Cluster	Nodes	Cluster	Nodes
C0	1,10,19,26,30,31,32	C7	5,13
C1	17,24,29	C8	4,12
C2	7,15,23,28	C9	25
C3	14,21,27	C10	22
C4	2,11,20	C11	8
C5	9,18	C12	6
C6	16	C13	3

Table 4.2: Assignment of nodes to clusters in G32

for this is the choice of dynamic scheduling algorithm. The currently implemented dynamic scheduling algorithm, tries to use idle processors only after they run out of work. The idle period of processors spent waiting for some event, may not utilized. The speedups for graph G82 are generally similar to those obtained for graph G41.

4.3.3 Graph G32

This graph is a representation of a weather forecasting program[15]. Although the number of clusters is high in the graph, all the clusters are small and variation in the clusters is very small. The speedups for this graph are in a smaller range than those obtained for graphs G41 and G82.

4.4 Conclusions

The experimental results indicate that following parameters are important for effectiveness of the integrated scheduling strategy:

- Properties of the program graph : The following properties in program graphs were found more suitable for our strategy:

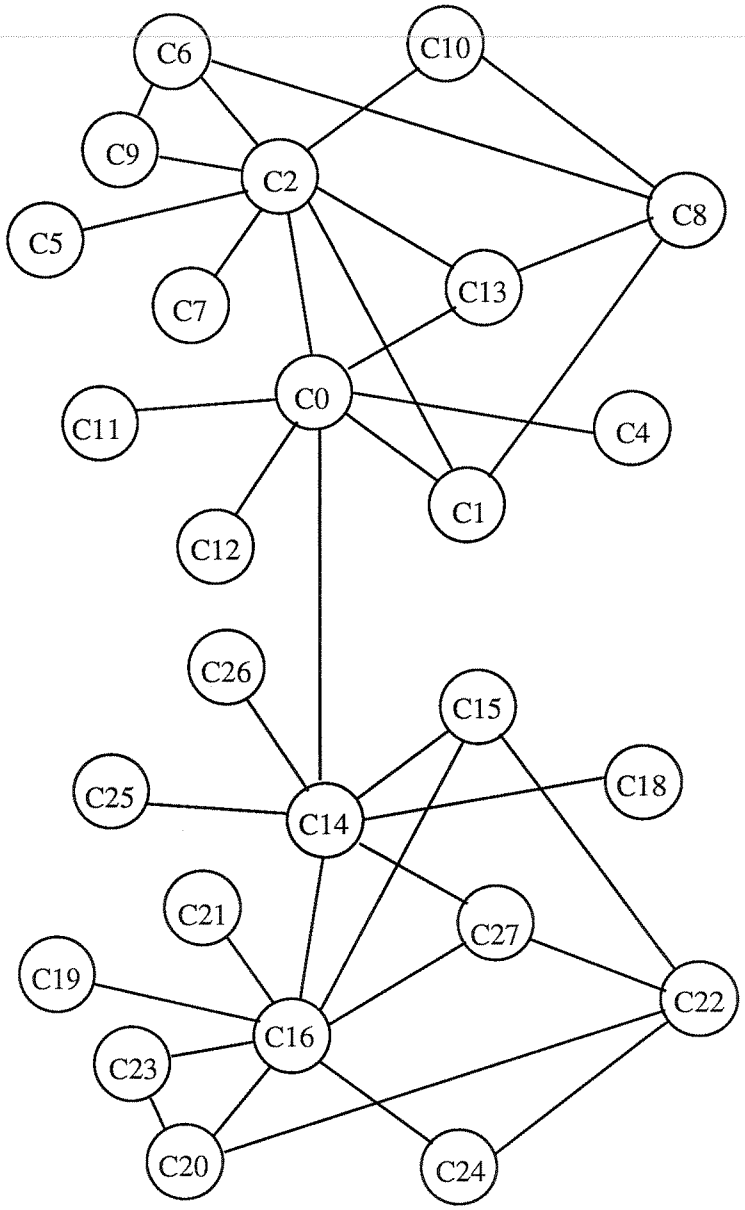

Figure 4.5: VAG of G82 after Linear clustering

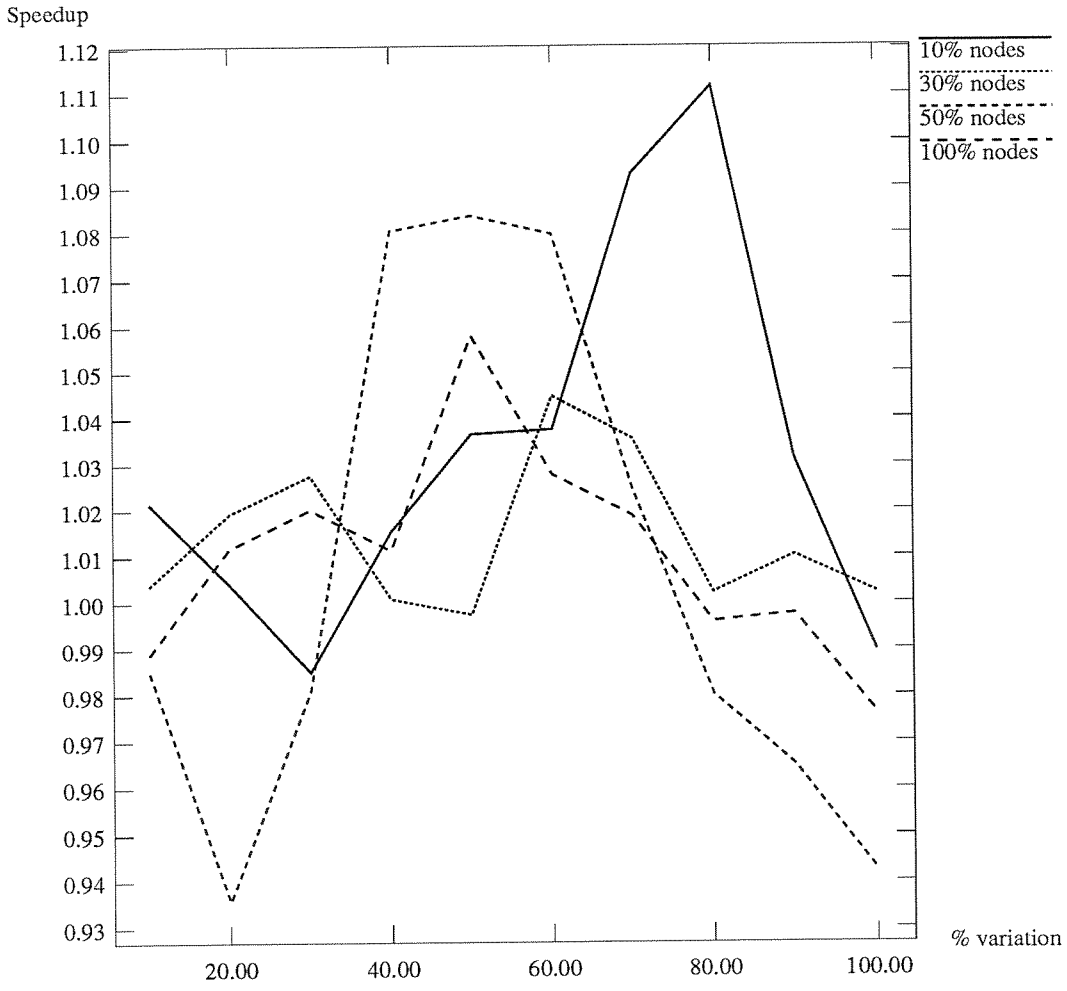

Figure 4.6: Performance results for G82

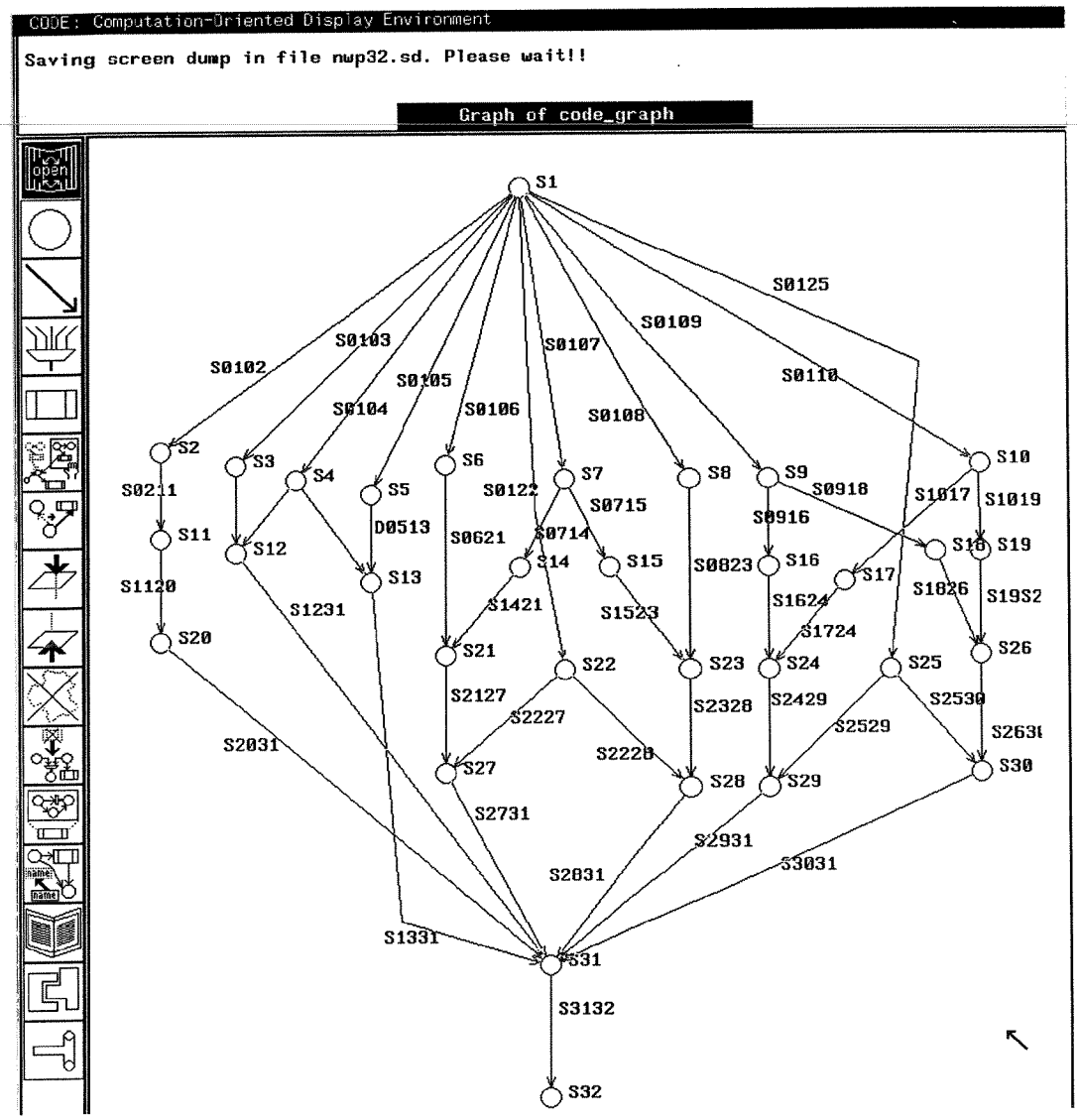

Figure 4.7: CODE graph representation of G32

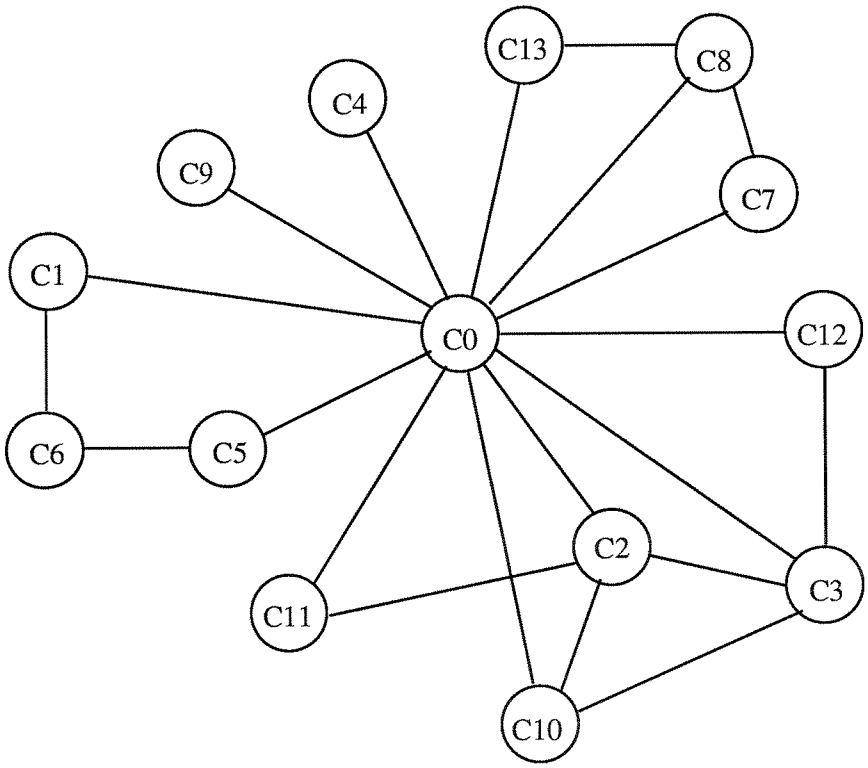

Figure 4.8: VAG of G32 after Linear Clustering

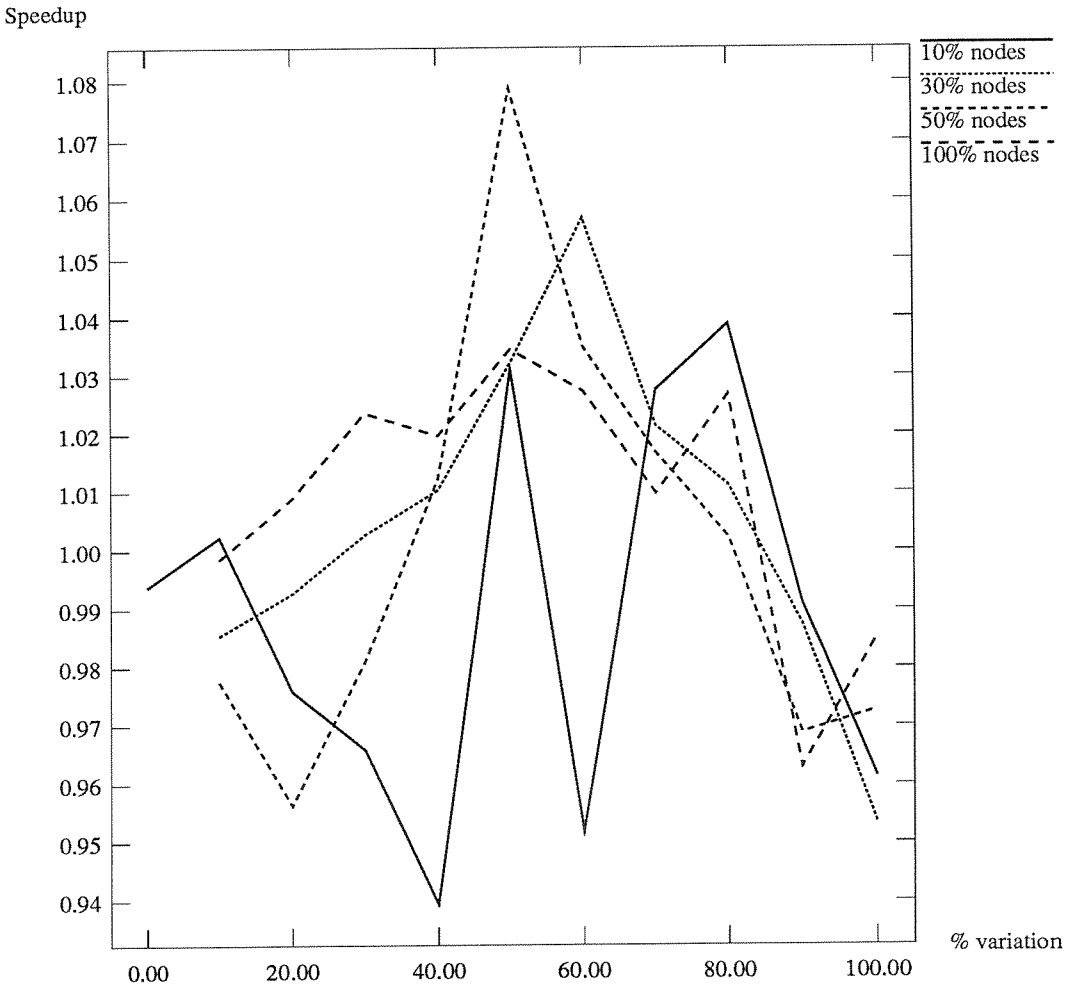

Figure 4.9: Performance results for G32

1. Large number of clusters : The number of clusters should be more than number of processors.
2. The clusters should have a wide variation in terms of their execution times.

- Overhead of moving a process : If the overhead of moving a process reltive to execution time of the unit is small the integrated scheduling strategy performs better.
- Uncertainty in predicted execution times : The integrated scheduling strategy can effectively handle moderate uncertainity in execution times. If the uncertainity is too low or too high, overhead of moving a process becomes dominant. In which case, the integrated scheduling startegy performs worse than static scheduling.

The currently used dynamic scheduling algorithm does not make use of idle periods when a process is waiting for an event. Use of such idle periods should lead to even better performance.

Chapter 5

Conclusions and Future research

This study has demonstrated that there exists a substantial spectrum of the parameter space of multiprocessor scheduling, where an integrated adaptive scheduling algorithm based upon a reasonable static scheduling system and a simple adaptive method can lead to substantial improvement over either applied separately.

The critical parameters are the degree of uncertainty in the execution characteristics of the program, and the relative granularity and number of units of computation.

The future work which remains to be accomplished includes the following:

- Study of the impact of uncertainty upon dynamic scheduling algorithms.
- Extension of the studies to include larger graphs on larger architectures.

BIBLIOGRAPHY

[1] Jacob Bahren and Edith C. Halbert. Roses: An efficient scheduler for precedence-constrained tasks on concurrent multiprocessors. In *Hypercube Multiprocessors 1986*, pages 123–147. Society for Industrial and Applied Mathematics, 1986.

[2] K. M. Baumgartner, R. M. King, and B. W. Wah. Implementation of Gammon: An efficient load balancing strategy for local computer system. In *International Conference on Parallel Processing*, pages II–77–II–80, 1989.

[3] S. H. Bokhari. On mapping problem. *IEEE Transactions on Computers*, C–30(3):207–214, March 1981.

[4] J. C. Browne. Foundation and programming of parallel computations: A unified approach. In *International Conference on Parallel Processing*, pages 624–631, 1985.

[5] J. C. Browne, M. Azam, S. M. Sobek, R. N. Rao, and C. L. Lin. *Programming with CODE: A Computation Oriented Display Environment.* The University of Texas at Austin, 1988.

[6] R. Bryant and R. A. Finkel. A stable distributed scheduling algorithm. In *International Conference on Distributed Computing Systems*, pages 314–323, 1981.

[7] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load

sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, SE–12, No. 5:662–675, May 1986.

[8] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. Technical Report LCSR-TR-153, Rutgers University, 1989.

[9] Mario J. Gonzalez. Deterministic processor scheduling. *Computing Surveys*, 9-3:173–204, September 1977.

[10] S. J. Kim. A general approach to multiprocessor scheduling. Technical Report UT–CS–TR–88–04, The University of Texas at Austin, 1988.

[11] P. Krueger and R. A. Finkel. An adaptive load balancing algorithm for a multicomputer. Technical Report CS–TR–539, University of Wisconsin, Madison, 1984.

[12] D. L.Eager, E. D. Lazowska, and John Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems*, pages 63–72. ACM SIGMETRICS, 1988.

[13] Soo-Young Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computers*, C-36:433–441, April 1987.

[14] Virginia M. Lo. Task assignment to minimize completion time. In *International conference on Distributed Computing Systems*, pages 329–336, 1985.

[15] David F. Martin and Gerald Estrin. Experiments on models of computation and systems. *IEEE Transactions on Electronic Computers*, EC–16(1):59–69, February 1967.

[16] L. M. Ni and K. Hwang. Optimal load balancing strategy for a multiprocessor system. In *Proceedings of International conference on Parallel processing*, pages 352–357, 1981.

[17] Ponnuswamy Sadyappan and Fikret Ercal. Nearest-neighbor mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, C36-12:1408–1424, December 1987.

[18] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. *Journal of ACM*, pages 17–26, 1986.

[19] S. M. Sobek. *A Constructive Unified Model of Parallel computation.* PhD thesis, The University of Texas at Austin, 1989.

[20] Craig Steele. Placement of communicating processes on multiprocessor networks. Technical Report 5184:TR:85, California Institute of Technology, 1985.

[21] Harold S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software engineering*, SE–3(1):85–93, January 1977.

[22] Symult Systems Corporation. *Symult Series 2010 Programmer's manual*, 1989.

[23] Jerry C. Yan. Post-game analysis– a heuristic resource management framework for concurrent systems. Technical Report CSL-TR-88-374, Stanford University, 1988.