A User Guide to the algorithm mapper: A system for modeling and evaluating parallel applications/architecture pairs

C. E. Houstis

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

John R. Rice
Purdue University, jrr@cs.purdue.edu

S. M. Samartizis

D.L. Alexandrakis

Report Number:
88-793
A USER GUIDE TO THE ALGORITHM MAPPER:
A SYSTEM FOR MODELING AND EVALUATING
PARALLEL APPLICATIONS/ARCHITECTURE PAIRS

C. E. Houstis
Elias N. Houstis
John R. Rice
S. M. Samartzis
D. L. Alexandrakis

CSD TR-793
August 1988
A USER GUIDE TO THE ALGORITHM Mapper:
A SYSTEM FOR MODELING AND EVALUATING PARALLEL
APPLICATIONS/ARCHITECTURE PAIRS

C.E. Houstis*, E.N. Houstis**, J.R. Rice**
S.M. Samartzis**, and D.L. Alexandrakis**

Computer Science Department
Purdue University
Technical Report CSD-TR 793
CAPO Report CER-88-29
August 1988

ABSTRACT

We present a methodology for evaluating the performance of application programs on distributed computing systems. An application $A$ is represented by an annotated graph $G(A)$ giving its requirements for processing, memory and communication, plus the precedence between computation modules. Machines are represented by a similar graph $G(M)$ and the methodology is to map $G(A)$ into $G(M)$. The mapping problem is subdivided into three steps (s1) a reduction of the parallelism of $G(A)$ to that of $G(M)$, (s2) scheduling the computational modules to (nearly) minimize communication costs, and (s3) actual layout of resulting graph into $G(M)$. The two technical problems addressed here are (1) using communication delay models to simplify $G(M)$ and the step (s3), (2) scheduling the modules (step (s2)). Our communication models apply well to "uniform" architectures, we explicitly consider the following four: single bus and common memory, single bus and distributed memory, multiple bus and distributed common memory, Banyan interconnection and distributed common memory.

* This research was supported by NSF grant DMC-8508064A1.
** This research was supported by ARO grant DAAG29-83-K-0026 and AFOSR grant 84-0385.
THE ALGORITHM MAPPER: A SYSTEM FOR MODELING AND EVALUATING PARALLEL APPLICATIONS/ARCHITECTURE PAIRS

C.E. Houstis, E.N. Houstis, J.R. Rice
D.L. Alexandrakis and S.M. Samartziis

1. INTRODUCTION

2. MODELING APPLICATION/ARCHITECTURE PAIRS
   2.1 Review of Existing Modeling Methodologies
   2.2 Comparison of Modeling Methodologies

3. METHODOLOGY FOR SHARED MEMORY ARCHITECTURES
   3.1 A Stochastic Data Flow Graph Representation
   3.2 A Deterministic Data Flow Representation
   3.3 Performance Analysis of the Parallel System Architecture
   3.4 Performance Measures of the Algorithm Mapper

4. THE ALGORITHM MAPPER RESOURCE ALLOCATION SYSTEM
   4.1 The Allocation Algorithm
      4.1.1 Parameters of the application
      4.1.2 Parameters of the architecture
      4.1.3 The allocation model
      4.1.4 The heuristic allocation algorithm
      4.1.5 The output of the allocation algorithm
      4.1.6 Analysis of the use of the time frame in the allocation algorithm
   4.2 The Algorithm Mapper Simulation System
      4.2.1 The algorithm mapper preprocessor
      4.2.2 The algorithm mapper graphics user interface
      4.2.3 Shared memory architecture models
   4.3 A Parallel Implementation of the Algorithm Mapper
   4.4 Time Complexity and Optimality of the Algorithm Mapper

5. EXAMPLE STUDIES USING THE ALGORITHM MAPPER
   5.1 Cholesky Decomposition
   5.2 PDE Collocation Application
   5.3 Robotic Elbow Manipulator Application
   5.4 Reduction of Parallelism in the Robot Application
   5.5 Performance Evaluation of Application/Architecture Pairs

6. SUMMARY

7. REFERENCES

8. APPENDICES
   I. Description of the Allocation Program
   II. Instructions for Using the Preprocessor
   III. Error Messages
1. INTRODUCTION

Parallel processor systems are efficiently utilized when the computations they are assigned can be performed in parallel and they are mapped in such a way as to maximize their speed up. Such systems can be interconnected in a variety of ways, which can be roughly classified as shared memory and non-shared memory. In shared memory systems, processors usually have their own local memory and communicate by contending for common resources, such as an interconnection network and shared memory. Shared memory can be either distributed among the processors in the form of shared memory modules, or it can be common. The interconnection network along with the shared memory can be regarded as the system's communication network. Interconnection networks are commonly from the class of multiple bus systems, ranging from the simplest configuration of a single bus up to a highest bandwidth configuration of a crossbar switch. The class of Banyan networks is also common. In the case of the multiple bus interconnection, the distance between the processors is clearly one, since the interconnection provides a direct connection from every processor to every other processor (through the common memory). In the Banyan case, the distance between processors can also be regarded as one, since on the average the routes between processors uses the same number of intermediate switches.

In non-shared memory architectures, a variety of interconnections exist. For example, processors can be placed at the nodes of a grid or at the nodes of a cube, etc. In these cases, the distance between processors is not necessarily one and it depends on the routes chosen between processors and the geometry of the architecture.

A number of methodologies exist in the literature which address the problem of mapping computations to parallel systems and they can be divided into two categories; (a) the methods that assume implicitly or explicitly that the distance between processors is one and (b) the methods that assume that this distance is different from one. Our methodology explicitly assumes the distance between processors is one and thus we review mainly such methodologies.

We first state the mapping problem. We consider an application A to be a computation with four properties: processing requirements, memory requirements, communication requirements and precedence (or synchronization) between the subcomputations. We visualize the computation broken into computational modules which are nodes of a precedence graph for the computations. We note the processing and memory requirements at each node of the graph. We note the communication requirements along each link or edge of the graph. This annotated graph is called G(A).

We consider a machine to have three components: processing elements, memory elements and communication paths (an interconnection network). Similarly, the machine can be
represented by an annotated graph $G(M)$.

In general, the mapping problem has three somewhat independent steps:

1. Schedule the computational modules so that the application runs efficiently.
2. Reduce the parallelism of the application to that of the machine.
3. Given Steps 1 and 2 are done, embed the application into the machine.

We apply our methodology to several applications, mainly numerical or real-time.

2. MODELING APPLICATION/ARCHITECTURE PAIRS

2.1 Review of Existing Modeling Methodologies

Most of the existing algorithms have addressed Step 1, [CHU 80], [CHU 87], [EFE 82], [HAES 80], [GTYL 76]. In general, the approaches used can be classified as graph theoretic, integer programming and heuristic methods.

In [CHU 80], a graph theoretic and an integer programming approach is used to solve the scheduling or allocation problem, which is defined as the assignment of $M$ modules to $N$ identical processors, so that interprocessor communication is minimized. The communication network connecting the processors is not described explicitly and the solution implicitly assumes a shared memory architecture.

In [CHU 87] a heuristic approach to a task allocation for distributed systems is presented. The issues discussed are very similar to what we have examined. The objective function in [CHU 87] is the minimization of the maximum processor loading. This has produced a number of differences in their analysis and solution of the problem. They are also not concerned about parallel module execution.

A heuristic approach to module allocation by minimizing interprocessor communication subject to a load balancing constraint is suggested in [EFE 82]. This approach clusters modules into processors and contains a mechanism to solve Step 2 of the mapping problem as follows: $N$ processors are assumed and the modules are clustered heuristically into possibly $N + \mathcal{K}$ clusters. A second heuristic, a module reassignment algorithm, is to divide the $K$ clusters among the $N$ processors when possible, by balancing their load.

In [HAES 80], a graph theoretic approach is used. $N$ modules in the application graph are initially associated with $N$ available processors and are regarded as resident modules. The algorithm then divides the rest of the graph into $N$ clusters, which are centered at the resident modules by minimizing communication between processors. In [GTYL 76], a two module clustering algorithms are used to search for “eligible” pairs of modules, eligible in the sense that when they are assigned to the same processor, the greatest possible interprocessor
communication is eliminated. This "fusion" continues until all eligible pairs are fused. The implicit assumption here is that the number of processors is equal to the number of clusters obtained. A shared memory architecture is also implicitly assumed, since the distance between processors is not considered.

In a non-shared memory architecture, the work in [BERM 84, 85], [BOKH 81], [WILL 84], provide heuristic algorithms for matching an application to a parallel architecture. In [BERM 84], a software system called Prep-P is described, it provides to the user a transparent interface of the parallel machine, a preprocessor for the parallel application and heuristics for mapping the application to a grid architecture. In [BOKH 81], the number of processors in the system are assumed equal to the number of modules in the application and using this as a starting point, one heuristically solves the problem of maximizing the number of pairs of communicating modules that fall onto pairs of directly connected processors. In [WILL 83], modules are assigned to heterogeneous processors using heuristics; queueing disciplines for the processors are decided based on these assignments. In all of the non-shared memory algorithms, communication delay depends to a great extend on the number of links (number of processors) in the path between the communicating processors. Thus, optimal routing between processors is an integral part of the mapping algorithms.

2.2 Comparison of Modeling Methodologies

For shared memory architectures, the approaches based on graph theoretic methods [CHU 80], [HAES 82] and integer programming methods [CHU 80], result in fairly complex algorithms which prohibit their use for applications with large graphs. The heuristic approaches [EFE 82], [GYLY 76], are more promising and simple to use.

In all of the above models, the interprocessor communication is measured in terms of the amount of data transferred among modules assigned to different processors. These models fail to incorporate the performance characteristics of the parallel system. In our model, data transfers are assigned a communication cost which includes the queuing delay incurred by communicating processors. This requires the performance analysis of the parallel systems architecture. The resulting performance measure is the queuing delay versus utilization of the communication network. Two different classes of interconnection networks have been examined, the Banyan and multiple bus. We have experimentally shown that queuing delay affects considerably the allocation decision, thus our approach incorporates the system’s architecture into the mapping heuristics. The use of performance models simplifies the system’s involvement into the problem. Moreover, it provides the means of examining the performance of application/architecture pairs. The third step in the mapping problem for shared memory architectures is simplified, since the distance between processors is one, and it is an arbitrary
assignment of the allocation of modules obtained in Step 2 to the systems processors, provided that memory requirements are satisfied.

In non-shared memory architectures, the communication cost is the system routing delay due to the various paths followed by information until it reaches its destination and this is accounted for in [BERM 84] and [WILL 83]. The approach used in [BERM 84] is useful, but computationally very complex when one considers general and arbitrarily large application graphs. In [WILL 83], the approach is simple, but one does not have enough information about how close it is to the optimum.

The approach we propose for shared memory architectures is computationally simple, applies to large and general graphs and it can easily be extended to non-shared memory architectures. Moreover, we have examined a few small problems (Cholesky decomposition) where we know the optimum schedule and we see that our algorithm produces this optimum schedule. The extension of this work for non-shared memory architectures is underway.

3. METHODOLOGY FOR SHARED MEMORY ARCHITECTURES

We have given three steps for the solution of the mapping problem. We deal primarily with Step 1 and Step 2, since Step 3 simplifies when shared memory architectures are used. In Step 1, a heuristic algorithm is used to schedule the application computation modules and data blocks into parallel clusters. The algorithm minimizes queueing delay among processors by assigning module pairs with the most communication to the same processor, provided (a) they do not have to be executed in parallel, (b) they do not overutilize the processor, and (c) they fit into its local memory. The output of this step is a number of clusters of modules that has the same parallelism as the application. Step 2 is the reduction of the clusters obtained in Step 1 to the number of available processors in the system. This is necessary mainly for two reasons: first, the application's parallelism in general can be much higher than the number of available processors. Second, there are architectures in which the number of processors can be adjusted to equal the number of clusters obtained. Multiple bus interconnection architectures present such a feasibility. In the Banyan interconnection system, the number of processors can be increased (or decreased) only by powers of 2. Thus if the number of clusters obtained is not a power of 2, then it is cost effective to use a number of processors equal to the next higher power of 2, less than the number of clusters. In this case, Step 2 is unavoidable. Step 3 is an arbitrary assignment of the clusters obtained in Step 2 to the system's processors, provided memory constraints are satisfied.

The mapping problem requires the modeling of the application and of the parallel system. The computations of an application are assumed to be partitioned and they are modeled by a precedence directed logic graph. This graph can be stochastic when the data flow in the
application is not known a priori as is often the case in real time applications or it can be
deterministic as often in the case in numerical applications. Next we describe both a stochastic
and a deterministic graph.

3.1 A Stochastic Data Flow Graph Representation

We denote by $G = (M, \Gamma)$ a directed parallel, cyclic and weighted AND - EOR logic
graph. Throughout, we assume that the partition of each application is represented by such a
graph. Let $M = \{m_i, i = 1, 2, \ldots, N_p\}$ be a set of weights corresponding to each program
(module) of the graph. Each $m_i$ represents the execution time requirement of the $i$-th program.
For each pair of programs $(i,j)$, we denote by $p_{ij}$ the probability of passing control from $i$ to $j$
and $\gamma_{ij}$ the expected amount of data transferred. Data transfers are measured in an ITUs (Information Transfer Unit) which are the smallest unit of information whose queueing delay due to
communication can be determined. Each link in the graph $L$ is associated with the weights
$\{l_{ij} = (\gamma_{ij}, p_{ij}); i,j = 1, \ldots, N_p; i \ne j\}$. Note that $L$ also represents the precedence graph for
the application. If program $i$ passes information to program $j$, then $i$ precedes $j$ in the execution.
Information may flow in both directions between a pair of program modules, we assume
the application is such that no infinite loops exist.

The synchronization requirements of the partitioned application are defined in terms of an
AND or EOR logic in the I/O of each node. It is worth noticing that an AND logic on the output
links of a node indicate the potential parallel execution of the modules associated with these
links. Figure 1, presents an example of a stochastic model that includes internal data blocks
referenced by each module. These internal data blocks are associated by the code and are not
part of the data from the application.

3.2 A Deterministic Data Flow Representation

We want to determine the total processing time requirements of the application. For this
we apply a Markov Chain analysis [LOW 73] to the stochastic graph $G$ and transform it into a
deterministic graph $G'$. The transformation $G'$ is defined by $(M', L')$ where

\[
M' = \{m'_i = m_if_i; \ i = 1, \ldots, N_p\},
\]

\[
L' = \{l'_{ij} = f_ip_{ij}\gamma_{ij} + f_jp_{ji}\gamma_{ji} \text{ if } i > j, \}
\]

\[
l'_{ij} = 0 \text{ if } i \le j \text{ for } i,j = 1, 2, \ldots, N_p\},
\]

For each program $i$, $f_i$ is the number of times it is executed, $m'_i$ indicates its total processing
time requirement, while $l'_{ij}$ is the total amount of ITUs transferred between programs $i$ and $j$.
The parallelism of the application (represented by AND logic) is implemented in a matrix form
by the precedence matrix $\Delta = \{\delta_{ij}; \delta_{ij} = 1 \text{ if } i \text{ and } j \text{ programs can be executed in parallel;}$
\[ \delta_{ij} = 0 \text{ otherwise}. \] The degree of parallelism of the application is the maximum number of program modules that can be executed in parallel. Our methodology uses the parallelism as assigned by the user. We call this the assigned degree of parallelism, it can be less than the actual degree of parallelism. The assigned degree of parallelism may be interpreted as the amount of parallelism that the user wishes to maintain in the computation. The information in the \( \Delta \) matrix is in the \( G \) graph and can be easily obtained from it. Notice that the matrices \( L' \), \( M' \) and \( \Delta \) constitute part of the input to the allocation algorithms to be described.

Figure 1. An example of a stochastic data flow graph model.
3.3 Performance Analysis of the Parallel System Architecture

Performance models of parallel multiprocessor systems are used to derive the queueing delay processors incur in communicating among themselves. This delay is due to two factors, (a) accessing the common interconnection network and (b) accessing the common memory modules. Several system architectures have been considered namely:

- **System 1:** Single bus and common shared memory system. [HOUS 87b]
- **System 2:** Single bus and distributed shared memory system. [HOUS 87b]
- **System 3:** Multiple bus and distributed memory system. [HOUS 87d]
- **System 4:** Banyan switch and distributed shared memory system. [HOUS 87d]

A comparative performance analysis of these architectures has been performed, and in all cases, the interconnection network queueing delay versus the utilization $D(u)$, has been computed. The performance analysis leads to a number of performance measures. The main variable is the average number of active processors, $AP$, in the system, i.e., processors doing computation in their local memory. From the $AP$, $D(u)$ can be computed as discussed in Section 4.2.3. The speed up, $S$, of the application running on the machine must be bounded as follows

$$1 \leq S \leq AP.$$

3.4 Performance Measures of the Algorithm Mapper

The *algorithm mapper* produces schedules of modules to processors and a number of performance measures, which relate to the system use and the efficient execution of the application. The main measures computed are directly related to the analytical systems model. The first is the average processor utilization $u_p$. To define $u_p$, we introduce

$$K = \text{the number of processors in the system},$$
$$A_j = \text{the set of all module indices in } G(A) \text{ assigned to processor } j,$$
$$u_p^i = \text{utilization of processor } i = \sum_{j \in A_j} m_j,$$

and then define

$$u_p = \frac{1}{K} \sum_{i=1}^{K} u_p^i.$$

Moreover, the average processor utilization $u_p$, relates to the active processors in the system $AP$ as follows,

$$u_p = \frac{AP}{K}.$$

Our second performance measure in the speed up $S$ defined in terms of $T_{seq} = \text{execution time of}$
the application by a single processor (sequential execution) and $T_{REAL} =$ execution time of the application by the parallel system. We define

$$S = T_{seq}T_{REAL}.$$ 

4. THE ALGORITHM MAPPER RESOURCE ALLOCATION SYSTEM

The algorithm mapper software system is composed of three main parts (a) a preprocessor, which takes the partitioned application and produces the information about its graphical representation and the input data to the mapping heuristics, (b) the heuristic algorithms for the mapping problem, and (c) a user friendly interface. The interface is interactive and displays the input and output of the mapping heuristics on a SUN workstation employing color graphics.

4.1 The Allocation Algorithm

In Step 1, we formulate and solve the module allocation problem. It is stated formally as a constrained minimization problem as follows:

**Input:**
(a) An application which consists of communicating program modules and data blocks.
(b) Specifications of a given distributed system: processor speeds, memory module sizes and $D(u)$ characterizing the interconnection network queueing delay vs. utilization.

**Problem:** Allocate the application modules and data in order to minimize the queueing delays due to interprocessor communications into (1) clusters of modules allocated to individual processors and (2) clusters of data allocated to memory modules. This is subject to the

(a) **Distributed System Constraints:**
(i) size of memory modules,
(ii) processor utilization capacity,

(b) **Application Constraints:**
(i) a fixed time allowed for executing the application,
(ii) program modules that can be executed in parallel will be executed in parallel.

Note that the objective of minimum processing time will be achieved as a result of parallel processing and minimizing the queueing delays due to interprocessor communications.

Next, we define variables and parameters of this problem and formulate it mathematically.

4.1.1 Parameters of the application

The annotated graph $G(A)$ is the data structure with the following elements.
\( N_p \) = Number of program modules in the application,
\( N_D \) = Number of data blocks in the application,
\( L' \) = Matrix representation of the intermodule communications in the application,
\( l'_{ij} = \) total number of ITUs transferred between the \( i \)-th module and \( j \)-th module,
\( = 0 \) if both modules are allocated to the same processor,
\( m_i \) = Total execution time of the \( i \)-th module (in TU's),
\( t_i \) = Code storage for the \( i \)-th module,
\( d_i \) = Number of ITU's in the \( i \)-th data block,
\( f_i \) = Number of times the \( i \)-th module is executed,
\( \Delta \) = Matrix representation of the parallelism in the application,
\( \delta_{ij} = 1 \) if the \( i \)-th and the \( j \)-th module may be executed in parallel,
\( \delta_{ij} = 0 \), otherwise,
\( H \) = Matrix representation of the data references in the application,
\( h_{ij} = 1 \) if the \( i \)-th module is referencing data from the \( j \)-th data block,
\( h_{ij} = 0 \), otherwise,
\( H' \) = Matrix representation of the data communication in the application,
\( h'_{ij} = f_i h_i d_j \) total number of ITU's referenced by the \( i \)-th module from the \( j \)-th data block.

### 4.1.2 Parameters of the architecture

The annotated graph \( G(M) \) is the data structure with the following elements. In this case the machine is completely homogeneous so no actual graph is needed in its representation.

\( K \) = Number of processors of the machine,
\( \mu^c_i \) = Storage capacity of the private memory of the \( i \)-th processor for storing code,
\( \mu^d_i \) = Storage capacity of the private memory of the \( i \)-th processor for storing data,
\( \eta_i \) = Permitted utilization of the \( i \)-th processor,
\( C \) = Capacity of the interconnection network,
\( D(u) \) = Queueing delay versus utilization characteristic of the interconnection network.

### 4.1.3 The allocation model

The mathematical formulation requires further quantities to express the allocation:

\( Q \) = Matrix representation of the assignment of program modules,
\( q_{il} = 1 \) if the \( i \)-th module is allocated to the \( l \)-th processor,
\( q_{il} = 0 \) if both modules are allocated to the same processor,
Matrix representation of the assignment of data blocks,
\[ x_{il} = 1 \] if the \( i \)-th data block is allocated to the \( l \)-th processor,
\[ x_{il} = 0 \] otherwise,

\[ R = \] Auxiliary matrix
\[ r_{ij} = 1 - \sum_l q_{il} q_{jl} \] if the \( i \)-th and \( j \)-th modules are allocated to the same processor,
\[ r_{ij} = 0 \] otherwise,

\[ S = \] Auxiliary matrix
\[ s_{ij} = 1 - \sum_l q_{il} x_{jl} \] if the \( i \)-th module and the \( j \)-th data block are assigned to the same processor,
\[ s_{ij} = 0 \] otherwise.

The three times associated with the computations of the \( l \)-th processor are:

\[ T_e = \] Program execution time
\[ = \sum_{i=1}^{N_r} m_i q_u, \]

\[ T_q = \] Queueing delay time due to communications
\[ = \sum_{i=1}^{N_r} q_u \sum_{j=1}^{N_r} r_{ij} r_{ij}^d \] where \( d \) is the average queueing delay per ITU in the interconnection network (see Section 4.1.4 for details of the computation of \( d \)),

\[ T_d = \] Queueing delay due to data block references
\[ = \sum_{i=1}^{N_r} q_u \sum_{j=1}^{N_r} s_{ij} h_{ij} d \] where \( d \) is the average queueing delay per ITU in the interconnection network (see Section 4.1.4 for details).

The architecture constraints for the \( l \)-th processor are expressed in terms of these quantities as follows:

Module storage:
\[ \sum_{i=1}^{N_r} t_i q_u \leq \mu^T. \]

Data storage:
\[ \sum_{i=1}^{N_r} d_i x_{il} \leq \mu^T. \]

In addition to these constraints on individual processors, there is also the parallel processing constraint represented by \( \Delta \); two modules executed in parallel cannot be assigned to the same processor. More explicitly, if \( \delta_{ij} = 1 \) then \( r_{ij} = 0 \). This assures the exploitation of the parallelism in the application.

Note that this model of the computation is not completely realistic. It does not include the algorithmic synchronization delays of the program modules. As discussed in Section 4.1.6, after the allocation is made one must make a small additional computation to estimate the actual
execution time of the algorithm.

4.1.4 The heuristic allocation algorithm

The heuristic algorithm used for the allocation problem involves an artificial parameter $T$, called the time frame:

$$ T = \text{total time allowed for execution of any processor (in TU's) }$$

This introduces an additional constraint into the problem, namely, for the $l$th processor

$$ T_e + T_q + T_d \leq \eta_l / T. $$

Thus the algorithm has the following input/output:

**INPUT:** $T, \mu_i, \eta_i, d_i, N_P, N_D, L, \Delta, H, \mu^p, \mu^d, \eta_l, C, K, D(u)$

**OUTPUT:** $Q$ = assignment matrix of the program modules,

$X$ = assignment matrix of the data blocks,

$N_{opt}(T)$ = optimal number of clusters of program modules and data blocks,

$u^l_p$ = utilization of the $l$-th processor, $l = 1, 2, \ldots, N_{opt}(T) = p$.

We now outline the heuristic algorithm. The first step is to select a value of the time frame $T$. It must not be too small, otherwise the application cannot execute in such a short time. If $T$ is too large, then all the program modules are placed in a few processors and the computation is not as distributed as it might be. The heuristic algorithm has five phases as follows:

1. **Initialization**

Start by allocating each module to a different processor ($r_{ii} = 1, r_{ij} = 0, i \neq j$). Search the $H'$ matrix by row to determine if any data block is referenced by a single module. If so, make it local to the processor by entering the data block in the processor's private memory list and by deleting this entry from the $H'$ matrix.

2. **List pairs of program modules eligible for merging**

Search the $L'$ matrix and locate all program modules pairs eligible for merging, that is, $\delta_{ij} = 0$, and choose those with the maximum number of ITU's transferred. If no such pair exists, stop.

3. **Find a pair of modules for merging**

For each pair $(p_i, p_j)$ in the list from Phase 2 calculate:

(a) Total amount of ITUs to be communicated had $p_i$ and $p_j$ been merged (intermodule communications plus external data block accesses.)

$$ D_{ij} = \sum_{s=1}^{N_P} \sum_{t=1}^{N_P} h'_{st} - \sum_{r=local} (h'_{r} + h'_{p}) + \sum_{s=1}^{N_P} \sum_{t=1}^{N_P} l'_{st} - l'_{ij} \text{ (in ITUs)} $$
(b) Interconnection network utilization due to $D_{ij}$

$$u(i,j) = \frac{D_{ij}}{C \times T}$$

(c) Interconnection network queueing delay due to $D_{ij}$

$$d = d(i,j) = D(u(i,j)) = \frac{k - AP}{u(i,j)}.$$ 

The function $D(u)$ depends on the machine architecture, see Section 4.2.3.

(d) Processor utilization had $p_i$ and $p_j$ been merged.

$$U^{(i,j)} = \{m_i' - m_j' + \sum_{r=1}^{N_r} (h_i' + h_j') - \sum_{r=\text{local}} (h_i' + h_j')d(i,j) + \sum_{r=1}^{N_r} (h_i' + h_j')d(i,j)/T \}$$

If $U^{(i,j)} > \eta_i$ then $p_i$, $p_j$ are ineligible for merging, delete them from the list. If all pairs are ineligible go to Phase 2, otherwise find the pairs for which $U^{(i,j)} \leq \eta_i$, if any. Then select the pair whose merging allows the maximum number of data blocks to be assigned to private memory. If there is more than one, then select the pair which yield the lowest processor utilization. If there is still more than one, pick one pair at random.

4. Merge a pair of modules

Merge the module pair $(p_i, p_j)$ from Phase 3 by adding $p_j$ to the list of modules of processor $n$ containing $p_i$. $m_n' = m_n' + m_j'$, $r_{nj} = r_{jn} = 1$. Delete $p_j$ as follows:

for $k < i$ and $k \neq j$, $i_{kj}' = i_{kj}' + i_{kj}'$ and

for $k > i$ and $k \neq j$, $i_{jk}' = i_{jk}' + i_{jk}'$ and

Further, examine the $H'$ to find if there is any data blocks made local due to this merging and, if any, add them to the private memory of this processor $n$, and delete them from the $H'$ matrix.

5. Reset the ineligible pairs

After a merge has been completed, a new graph has resulted with one less node and fewer links (as computed in Phase 4). Reset all ineligible pairs of modules (from previous merge) to eligible and return to Phase 2.

4.1.5 The output of the allocation algorithm

A sample output of the algorithm is given in Table 1. The output includes the total (%) utilization of each processor which is the processing time plus communication time of modules assigned to a processor.
### ALLOCATION SOLUTION AND WORKLOAD STATISTICS

number of modules is 41.
estimated elapsed time is 70.0 & capacity is 151515.2 time units
allowed real time for processing is 70.0 time units

<table>
<thead>
<tr>
<th>id</th>
<th>total (%) utilization</th>
<th>proc (%) utilization</th>
<th>processor assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51.57</td>
<td>51.20</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>69.76</td>
<td>69.60</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>24.19</td>
<td>23.75</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>50.20</td>
<td>50.00</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>45.27</td>
<td>45.19</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>55.23</td>
<td>55.40</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>67.74</td>
<td>65.67</td>
<td>10</td>
</tr>
</tbody>
</table>

**Average:** 52.07 51.55

### PROCESSOR UTILIZATION

<table>
<thead>
<tr>
<th>id</th>
<th>memory inter-module communication cost</th>
<th>interprocessor communication cost</th>
<th>total cluster processing time</th>
<th>total processor workload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39000</td>
<td>35.8</td>
<td>39035.8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>650</td>
<td>46.8</td>
<td>65618.8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>46500</td>
<td>16.6</td>
<td>46616.6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>28900</td>
<td>35.6</td>
<td>28935.8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>14000</td>
<td>31.6</td>
<td>14831.6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>35200</td>
<td>36.8</td>
<td>39238.8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>220000</td>
<td>46.0</td>
<td>220646.0</td>
<td></td>
</tr>
</tbody>
</table>

**Average:** 55028.67 36.1 55064.7

delay per item 0.000007

### APPLICATION’S COMMUNICATION & PROCESSING COST

(in time units)

### APPLICATION’S COMMUNICATION REQUIREMENTS

(in time units)

<table>
<thead>
<tr>
<th>id</th>
<th>data reference via interconnection</th>
<th>interprocessor data transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0</td>
<td>39300.0</td>
</tr>
<tr>
<td>2</td>
<td>0.0</td>
<td>8300.0</td>
</tr>
<tr>
<td>3</td>
<td>0.0</td>
<td>46500.0</td>
</tr>
<tr>
<td>4</td>
<td>0.0</td>
<td>20900.0</td>
</tr>
<tr>
<td>5</td>
<td>0.0</td>
<td>14830.0</td>
</tr>
<tr>
<td>6</td>
<td>0.0</td>
<td>35200.0</td>
</tr>
<tr>
<td>7</td>
<td>0.0</td>
<td>2200000</td>
</tr>
</tbody>
</table>

### Table 1. Output of the allocation algorithm, the variable id is the index of the program module or data block.

<table>
<thead>
<tr>
<th>id</th>
<th>memory module instruction size</th>
<th>utilization</th>
<th>instructions blocked allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100.00</td>
<td>7.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>100.00</td>
<td>10.00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>100.00</td>
<td>3.00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>100.00</td>
<td>4.00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>100.00</td>
<td>5.00</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>100.00</td>
<td>6.00</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>100.00</td>
<td>6.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>memory module instruction size</th>
<th>utilization</th>
<th>instructions blocked allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100.00</td>
<td>7.00</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>100.00</td>
<td>10.00</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>100.00</td>
<td>3.00</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>100.00</td>
<td>4.00</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>100.00</td>
<td>5.00</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>100.00</td>
<td>6.00</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>100.00</td>
<td>6.00</td>
</tr>
</tbody>
</table>

*** there are no common data blocks ***

Table 1. Output of the allocation algorithm, the variable id is the index of the program module or data block.
4.1.6 Analysis of the use of the time frame in the allocation algorithm

When the time frame \( T \) changes, a different clustering of \( G(A) \) is obtained. Ideally, we would like to find the shortest time frame \( T \) for which the application can be run. Let \( T_{PAR} = \text{the shortest time frame for which the machine can run the application } A \text{ in parallel.} \)

At the end of Step 1, \( T_{PAR} \) is the shortest time frame for which the application is allocated to as many processors as its degree of parallelism. In Step 2, \( T_{PAR} \) is shortest time frame for which the number of clusters obtained equals the number \( K \) of processors available. Thus \( T_{PAR} \) is the time frame for which optimal clustering is obtained.

As mentioned earlier, the model of the computation does not include the time for delays due to the algorithmic synchronization of the application. That is, the time one program waits on its predecessors to finish is not included. The time frame \( T_{PAR} \) includes only the times for execution, communication and data reference within a cluster. Synchronization delays are an attribute of the applications and their effect can be easily computed once the allocation is made. One merely "executes" the algorithm abstractly using the information in \( G(A) \). The resulting execution time is denoted by \( T_{REAL} \).

From the output of the allocation algorithm, we can compute the average number of active processors, \( AP \), (see Section 3.3) as follows

\[
AP = \sum_{i=1}^{K} u_p^i
\]

We can obtain \( T_{PAR} \) and \( T_{REAL} \) from the algorithm and then the following bounds hold

\[
1 \leq S = \frac{T_{seq}}{T_{REAL}} \leq \frac{T_{seq}}{T_{PAR}} \leq AP.
\]

4.2 The Algorithm Mapper System

The algorithm mapper is a software system which maps any application to a shared parallel memory architecture system. It is made up of a preprocessor, the heuristic allocation algorithm called ALLOC, and a user friendly graphical interface.

The allocation algorithm was initially implemented in Pascal [STEV 82] for a single time frame and a hypothetical system queueing delay function. All input data were assumed known. No preprocessor or user interface was available. The code was inefficient and did not completely solve the mapping problem.

The present allocation algorithm ALLOC is implemented in the language C with a variable time frame. A library of performance functions has been added of four multiprocessor architectures, namely
(a) Single bus with shared common memory,
(b) Single bus with distributed shared memory modules (two port memory),
(c) Multiple bus with shared memory modules,
(d) Processors and shared memory with a Banyan interconnection.

In the case of a multiple bus with shared memory module architecture, the number of busses can be set equal to one, thus obtaining a single bus with distributed shared memory architecture or set equal to the number \( K \) of processors, thus obtaining a crossbar interconnection of processors with distributed shared memory.

The time frame \( T \) is varied. Initially a large value of \( T \) is used, which decreases until \( T_{\text{PAR}} \) is obtained, i.e., the shortest time frame for which the number of clusters obtained equals the application parallelism in Step 1, or after the parallelism reduction Step 2, it equals the number of available processors in the system.

After an allocation has been obtained and the clusters formed, the execution of the application is simulated in order to obtain \( T_{\text{REAL}} \), i.e., the actual real time required to execute the application architecture. This simulation routine is also in the C program.

4.2.1 The algorithm mapper preprocessor

The preprocessor PALLOE is an interface between the user and the allocation program ALLOE which automatically creates its input data file. The preprocessor is necessary because the amount of data needed by ALLOE is very large, especially for large graphs.

PALLOE is written in the C programming language and currently runs on a DEC VAX-11/780 computer under the Berkeley operating system (4.3 BSD). All applications programs for PALLOE should be written in C. There is also a version of the preprocessor for applications written in FORTRAN.

PALLOE uses either an abstraction of the actual application or an instrumented version of it or a combination. In any case, the application must be partitioned into subroutines corresponding to the program modules to be used in the parallel implementation. Further, the data communicated between these modules must be explicitly specified as to destination and size (in bytes). The execution time of the code in a program module can either be specified explicitly or the actual code is compiled and timed during a sequential execution of the application. In summary, PALLOE determines the execution times and total communication of modules from this special version of the application. We now give more details about the features of the input to PALLOE.
3. `finis()`
   This must be the last statement of the main program and signals the final computations and creation of the output files.

*Examples*

Suppose we have the following graph of four modules. Then the main program would look like:

```c
main 0
{
    [ initialize data ... ]
    process(module1, 1, ptr1)
    cobegin()
        process(module2, 1, .1, ptr2)
        process(module3, 1, .5, ptr3)
        process(module4, 1, .4, ptr4)
    coend()
    finis()
}
```

*Clock Control*

The preprocessor maintains a table of virtual clocks, one for each process (program module), and an incremental "stop watch" variable `clock_` for the currently executing process. Together these determine the virtual time for the current module.

There are two ways to alter the virtual timer while executing (note that only `clock_` should be altered).

D.1. *Increment the clock manually.* Insert lines like

```c
    clock_ += execute_time;
```

into a subroutine, and compile it with `cc` (C compiler). For example, a hardware complex multiply module could be modeled as

```c
extern unsigned int clock_;  
typedef struct {float re, im;} COMPLEX;  
cmult(a, b, c) /* A = B*C, a is a pointer to A, etc. */  
    COMPLEX *a, *b, *c;  
    { int tm = 892;  
        clock_ += tm;  
    }
```

where `tm = 892` is the number of virtual nanoseconds needed to obtain the result. `tm` could also be a parameter to `cmult()` if execution time variations need to be modeled.
The user writes for each module a different subroutine (which might call other subroutines, but NOT subroutines that are modules). When a module needs to pass information to another module, the PALLOC statement send (pid, nbytes) is used, where pid is the identifying number of the destination module and nbytes is the number of bytes sent. A module receives information by using the PALLOC statement receive (message) to indicate acceptance of information from any other module. The preprocessor maintains a table for each module with all the necessary data, and which is updated as the preprocessor executes.

In any module the following can be used:

General PALLOC Statements

1. send(pid, nbytes) -- in order to send nbytes to pid module, were pid is the number of the destination module,
2. receive(messages) -- in order to receive information from any other module,
3. getpid() -- this returns the number of the current module at run time,
4. clock_on() clock_off() -- used for execution time control; see below under Clock Control.

Main Program Constructs

After writing the code for the various modules, the user must explicitly specify the sequence in which the modules are going to be executed. Two constructs are provided by the preprocessor, sequential and parallel. They are controlled by the main program of the application. The purpose of the main program is to initialize the data and specify the module sequence. The following statements are provided:

1. process (name, pid, prob, parmptr)
   where name - is a pointer to the program module to be executed,
   pid - is the number of the current module,
   prob - is the probability with which information is being sent from the current module to the successor module,
   parmptr - is a pointer to a structure which contains data to be passed to the successor module.
   This tells the preprocessor that this ‘name’ is one module of the computation. The pid must be assigned by the user, this is provided so that the user has better control.

2. cobegin() coend()
   These specify that the modules bracketed are to be executed in parallel.
D.2. **Increment the clock automatically subject to start and stop signals.** Write the process routine without any reference to `clock_` (except through two controls described below) and compile it with the C compiler ‘simcc’ instead of the usual ‘cc’. This modified C compiler generates its code by looking at the “usual” assembly code and inserting an increment to `clock_` for each VAX instruction. Timing data is taken from a table of (VAX instruction, execution) pairs. The default table has execution time = 0 nsec for all instances, but this can be altered easily by supplying another table. The clock is initially turned off; it can be started with a call to `clock_on()` and turned off again with a call to `clock_off()`. For example, diagnostic output can be isolated from the simulation timing by writing the code:

```c
    clock_off();
    printf(...);
    clock_on();
```

The ‘simcc’ compiler command line option ‘-i’ can be used to force the clock to be turned off initially.

Actually, there are two routines named `clock_on()` and `clock_off()`. Calls to parameterless subroutines are compiled into a single instruction like

```
calls $0, _clock_off
```

where the value of the starting address of `_clock_off` (i.e., `clock_off()`) is to be supplied by the linker loader. When `ccsf` (the component of simcc which inserts increments to `clock_`) sees the above instruction, it simple copies the following instructions until it sees

```
calls $0, _clock_on
```

at which point it begins to insert increments to `clock_` as if the “intervening” code was not present. Note that the clock on/off switch retains its setting across subroutine boundaries, since it acts solely as a textual modification. A final hazard is the compiler’s removal of dead (unreachable) code; e.g., do not place these switches after an infinite loop with no exit.

The automatic timing scheme is not without faults, such as:

1. **Dependence on the simulator’s host machine for the simulated instruction set model.** While the VAX instruction set is rich and varied, there are many operations which require more than one instruction to implement. It is then difficult to formulate an appropriate timing model for new hardware operations.

2. **Invariant instruction execution times.** We should be able to give different weights to instructions using register access, memory access, indirect memory access, autoincrements, etc., and to ignore certain “bookkeeping” instructions such as stack pushes before a function call or arithmetic type conversions enforced by the C compiler.
3. **Compiler dependence.** The C compiler optimizations do not always generate high quality code from readable programs. For example, unnecessary address calculations may be performed while stepping through an array; this problem is correctable by using pointer variables, but such programs are more difficult to produce.

4.2.2 The algorithm mapper graphics user interface

*AllocTool* is a graphical interface to the allocation algorithm. It helps the user to specify the computation graph, to enter the required data for the algorithm, and to display the results in a graphical form. In general, for a specific application, the user has to do the following steps to use the allocation algorithm:

(a) Run *AllocTool* (which is trivial).
(b) Draw the application data flow graph.
(c) Specify the various data that are required for the algorithm.
(d) Run the allocation algorithm and display the results.
(e) (Optional) Store the application description in a file for later use.

Steps (b) and (c) can be replaced by loading an application data file previously prepared.

*AllocTool* uses the Sun View library routines and should work on any Sun workstation. In the following paragraphs, we use the terminology of Sun View library when referring to windows and specific items within these windows. These terms from Sun View will be in *ITALICS*.

The tool is composed of two basic *FRAMES* (windows). The first one is the control *PANEL*, that controls the functions of the tool and the second is a frame containing a *CANVAS* window for images and a small *PANEL* on the top for diagnostic messages. In the following paragraphs, we describe the operation of these windows.

**Message PANEL**

The message *PANEL* is the subwindow at the top of the second *FRAME*. It has two lines that display diagnostic and error messages. The first line has three mouse images that represent the mouse left, middle and right buttons. On the right of each image is displayed a short text describing the function that the respective button will perform in the current state of the tool. If no text is displayed for a button, that means no function is assigned to this button.

The second line is for general purpose diagnostic messages.
Main PANEL

This PANEL is composed from several PANEL ITEMS, that we describe starting from the top and going down.

Dir ITEM. Is a TEXT item and is used to display the current directory. When the tool starts, the directory is assumed to be the current user directory, afterwards the user can change to any directory he likes by changing the text in the dir ITEM. To change the text, you click the left mouse button on the text, erase any unwanted characters using the backspace, and complete the new directory. If the directory is not a valid one, no error message is displayed, but when the tool tries to use this directory one gets an error message “directory not found”.

File ITEM. This is also a TEXT item and can be used in similar ways with the dir ITEM described above. The file ITEM is used to enter the name of a file that contains data for an application after using AllocTool. These data can be loaded (using the load BUTTON), edited and rerun through the allocation algorithm. Also one can specify in the file ITEM the name of the file where the current data application is stored (using the store BUTTON). All the files are in the directory specified by the dir ITEM.

Load BUTTON. This is a BUTTON used to load the data for the application from a specified file (file ITEM). These data have been produced using AllocTool. They contain data for the image of the application graph, as well as data for the various parameters of the allocation algorithm. If the file specified does not exist, then an error message is displayed. During the time the tool is loading the data the cursor is transformed to an hourglass and no other operation is possible in the tool.

To initiate the loading process, one clicks the left mouse button when the pointer is inside the BUTTON area.

Store BUTTON. This is also a BUTTON and is used to trigger the store process to place the current application data in the file specified in the file ITEM. Data are stored for both the image of the computation graph and the algorithm parameters. If the file used already exists (is not a new file), the tool asks the user to confirm overwriting of the file.

Quit BUTTON. This is used to exit from AllocTool. The current application data are not saved by default anywhere, so to save them, one must use the store BUTTON.

Histo BUTTON. This BUTTON displays a processor utilization histogram after an execution of the algorithm has been completed.

Help BUTTON. (not implementated yet).

Edit BUTTON. One can edit the file specified in the file item using this BUTTON.
**Param BUTTON.** Using this BUTTON, one enters or changes the values for several parameters of the allocation algorithm. A popup window appears on the screen that displays the parameter names and their current values. The left mouse button selects the parameter to change, and one can enter the new value in the value field for this parameter. When finished, select the done BUTTON from this window and the new values are stored. Note that the window is a blocking one; all the input from the mouse or the keyboard is directed to this window. Nothing else in the screen can accept input simultaneously.

**Clear BUTTON.** This BUTTON clears the image of the application graph from the CANVAS (if any) and initializes the internal database of the tool, so a new application can be started (entered or loaded).

**Redraw BUTTON.** This redraws the image of the data flow graph of the current application.

**Exec BUTTON.** This BUTTON executes the allocation algorithm. The tool produces an output file which is fed to the allocation program to get a result file. The first file is named current_file.alloc and the second current_file.res, where current_file is the file name currently specified in the file item. While execution takes place, the cursor is transformed to an hourglass and no other function is possible. After the completion of the program, the resulting allocation is displayed on the application graph. Each node now has a label specifying the processor number to which it has been assigned. If the workstation supports color, different processors are represented by different colors. NOTE: It is important to make sure that all the required data are given in order for the allocation algorithm to work correctly.

**Grid CHOICE ITEM.** This is a CHOICE ITEM, that is, an item which has values from a small set, here ON or OFF. If the grid is ON, a grid is displayed on the CANVAS in order to assist the drawing of the application data flow graph. If the grid is OFF, the grid disappears.

**Label CHOICE ITEM.** This is also a CHOICE ITEM like grid and its values (SHOW or HIDE) control showing or hiding the labels on the edges of the data flow graph.

**Show Canvas Window BUTTON - Hide Canvas Window BUTTON.** This is a single BUTTON, but the label string changes every time it is used. The use is the obvious one, to show and hide the canvas window where all the graphics is displayed.

**Command CHOICE ITEM - Object CHOICE ITEM.** These two CHOICE ITEMS are used for drawing. In order to make the graph, one selects commands and objects. (Not all the possible combinations are valid). The data flow graph is composed from four different objects:

- circles, (that represent computation blocks)
- squares, (that represent memory blocks)
- edges, (that represent the communication paths = graph edges)
number, (that are the labels on the edges)

The mouse and mouse buttons are used for the selection. Most of the operations are performed by pressing and depressing a mouse button at a specific point in the CANVAS window. The following commands are used to create a graph.

**ADD**

This adds some object in the current graph. The add command is not valid for number objects, there is another way of assigning the labels for the graph edges.

To add a circle (computation block) after selecting ADD, move the cursor in the canvas subwindow and point the location where the center of the circle is to be. Then click the left mouse button and the new computation block appears on the canvas. One can enter a square (memory block), the same way, but the point the cursor points to is the top left corner of the square.

To add an edge (line) one specifies a source node and a destination node. Do that by pointing and clicking the left mouse button inside the image of the source node, and the right button inside the image of the destination node. It is important to enter the source node first. If the edge exists, then an error message is displayed. This produces a straight line edge.

If the edge is to be a broken line, then the intermediate points must be inserted in order, using the middle mouse button. So, first insert the source using the left mouse button, then the intermediate points (if any) using the middle mouse button, and finally the destination using the right mouse button.

**DELETE**

The delete command is used to remove objects from the graph using the mouse. To select a node for deletions, just click inside the node image. If a node with edges is deleted, all the incident edges are deleted also. To select an edge for deletion, specify the source and destination nodes. Thus, this command deletes an object simply by selecting it. The delete command, like add, does not apply to number objects.

**MOVE**

This command is used to relocate an object. It is not valid for edges as they follow the moves of the nodes they are attached to.

To move an object, first select it (the selection procedure is the same as in delete). Then point to the new location and click using the middle mouse button. One can repeatedly move the same object without selecting it again, since the tool remembers the last selection. To move
a number object (edge label), select the edge as in the delete command.

**DRAG**

The drag command is a "continuous" version of move, the selected object follows the track of the mouse on the CANVAS. An object is selected to drag as in move command. Hold down the middle mouse button and drag the pointer around the CANVAS to an appropriate new location, then release the mouse button. The object image is relocated to this point.

**ALIGN**

The align command is applicable only to graph nodes and makes graphs with some nodes horizontally or vertically aligned. The horizontal alignment of the nodes is important because vertical position is used to infer the parallelism of the graph, essential information for the allocation algorithm. Computation blocks that can be executed in parallel must be drawn in the same horizontal level. The grid choice item is supplied to aid in this. Vertical alignment is for appearance (symmetry) purposes. The nodes are selected using the left mouse button. The first node selected is assumed to be the "pivot" node, it represents the start of the alignment. All the other nodes are aligned relative to it. After selecting the nodes, click the middle mouse button for vertical alignment or the right mouse button for horizontal alignment. Both circles and squares are aligned using the same function, so selecting either type will do.

**DATA**

The data command is a facility to enter data about objects into the data flow graph. Most of these data are used by the allocation algorithm, and some by the tool itself. One can enter data only for edges and nodes. When data is the current command, one selects an object and a popup blocking subwindow appears. It has named data fields and one can specify values as with the param BUTTON.

An edge is selected in the same way as for adding an edge (see add). The subwindow that appears has the following fields:

*Source node*, the source node for the edge,

*Destination node*, the destination node,

*Probability*, the probability that execution of the source node is followed by execution of the destination node,

*Communication*, the amount of communication (in ITU's) that is transferred on this edge.

The last two fields are data for the allocation algorithm and must be entered by the user. The first two are displayed for convenience to identify the edge and should not be edited. After completing any changes in the subwindow, click the done BUTTON and the subwindow disappears. The new values are now stored in the tool's database.
The window for a computation block has the following fields:

* **Label**, the label in the circle,
* **Radius**, the radius of the circle,
* **Level**, the parallelism level number,
* **Processor**, the number of the processor assigned to this block by the allocation algorithm,
* **Initial Probability**, the probability of this block to be the first one in the computation,
* **Expected Time**, the estimated time for this block to complete execution,
* **Instruction Block Size**, the number of bytes for the code of the instruction block.

Except from the radius and level which are inferred from the graph image data, the rest are data for the allocation algorithm. The user should specify them prior to execution. The processor number is an output variable computed by the allocation algorithm.

The window for a memory block has the following fields:

* **Label**, the label of the memory block,
* **Size**, the width of the square,
* **Level**, the level in the graph,
* **Processor**, the number of the processor where this block is assigned,
* **Datablock Size**, the size of the data (memory) block.

Of these fields, only the last one must be entered by the user and it is used in the algorithm.

All data fields for the data command contain initial default values. These are, in general, not realistic values.

A sample output of the graphics user interface is shown in Figure 2. The numbers of the graph nodes are the processor numbers to which the modules are allocated. Processor utilization is also shown graphically.

### 4.2.3 Shared memory architecture models

Four different shared memory architectures (see Section 3.3.3) are analyzed and their queueing delay functions $D(u)$ presented. In all cases the overall system organization is the same and it is introduced first.

The systems are composed of $k$ processors and $k$ memory modules, (although we are assuming that the number of processors is the same as the number of memory modules, the same analysis can be applied when the number of processors is less than the number of memory modules). Each processor has its own private memory module, where the program and the data are stored. If processor $i$ wants to communicate with processor $j$, it prepares a message and sends it to memory module $j$ where it can be accessed by processor $j$. We assume asynchronous communication, so any processor can be in any one of three states:
Figure 2. Sample output of the graphics user interface of AllocTool. Colors are used in the large CANVAS FRAME (center) to indicate the processor assignment (or numbers for black and white workstations). Processor utilization is shown at the upper right, the control PANEL at the upper left.
1. The processor is executing a program in its local memory,
2. The processor is sending a message to another processor,
3. The processor is blocked waiting for the interconnection network to deliver a message to another processor.

Processors in the first state are considered active processors doing useful work. Processors are considered to be in the first state (executing) even when they are idle waiting for data. We do not account for the time taken by processors to read messages, since this is considered to be part of the program executed by the processor.

We assume that the time between the generation of messages is an exponentially distributed random variable with mean \( \frac{1}{\lambda} \), and the length of the message is an exponentially distributed random variable with mean \( \frac{1}{\mu} \). We also assume that an access request from processor \( i \) is directed to memory module \( j \) with probability \( p_{ij} = \frac{1}{k} \), i.e., the communication pattern is homogeneous. When a processor sends a message to another processor, it stops executing its program. Then, if the interconnection network can establish a path from the source processor to the destination memory, it does so instantaneously with no delay and the processor begins to send its message. When the message is completed, the processor returns to its active state. If there is contention at the interconnection network or destination module, the processor is put in a queue, "blocked", waiting for the contention to be resolved before transmitting its message.

The performance analysis of these architectures is documented in detail in [HODS 87], [HOUS 88]. The main performance measure obtained is the average number of active processors, \( AP \). We have also obtained the average queueing delay per message, \( D(u) \), and the average utilization of the communication network. We summarize these results below.

System 1: Single bus and shared common memory architecture.

Set \( \rho = \lambda/\mu \) then

\[
AP_1 = \frac{1}{\rho \sum_{j=0}^{k} \frac{\rho j!}{(k-j)!}} - 1
\]

and

\[
D_1(u) = \frac{(k - AP_1)}{u}.
\]
A three processor example of this system is shown in Figure 3 and $D_1(u)$ is plotted in Figure 4.

Figure 3. Single bus and shared common memory architecture with 3 processors $P_i$ and 3 private memories $PM_i$.

Figure 4. Delay vs. utilization $D_1(u)$ of the interconnection network of System 1 of Figure 3.

System 2: Single bus and distributed shared memory architecture (double port memory modules).

Set

$$\rho = \frac{\lambda}{\lambda + 2\mu}$$
then we have

\[
AP_2 = \frac{(1 - \rho) \sum_{j=0}^{k} \frac{\rho^j k!}{(j - k)!}}{\rho \sum_{j=0}^{k} \frac{\rho^j k!}{(j - k)!}},
\]

\[n = AP_2 \rho/(1 - \rho),\]

\[D_2(u) = (k - AP_2/(1 - \rho))/u.\]

A three processor example of this system is shown in Figure 5 and \(D_2(u)\) is plotted in Figure 6.

Figure 5. Single bus and distributed shared memory architecture with 3 processors \(P_i\), 3 private memories \(PM_i\), and 3 common memories \(CM_i\).

Figure 6. Delay vs. utilization \(D_2(u)\) of the interconnection network of System 2 of Figure 5.
System 3: Multiple bus and distributed shared memory modules architecture. It is of order $k \times m \times b$ where $k$ is the number of processors, $m$ the number of memories and $b$ the number of busses.

Set $\rho = \lambda / \mu$ and define the array $p_j(l)$ by

$$p_j(l) = p_j(l - j) + p_{j-1}(l - j) + \cdots + p_1(l - j) + p_0(l - j)$$

with initial conditions

$$p_j(l) = 0 \quad l < j$$
$$p_0(l) = 0 \quad l > 0$$
$$p_j(l) = 1 \quad j \geq 0$$

Then set

$$\beta_l = \frac{\sum_{j=1}^{b-1} jp_j(l) + b \sum_{j=0}^{l-b} [p_0(j + b)p_{m-b}(l - 2b - j + m)]}{\sum_{j=1}^{b-1} p_j(l) + \sum_{j=0}^{l-b} [p_0(j + b)p_{m-b}(l - 2b - j + m)], \quad l \geq 0}$$

and

$$P'_k = \left[1 + \sum_{j=0}^{k-1} \left[p^{k-j} \frac{k!}{j!} \prod_{i=1}^{j} \beta^{-1}_i\right]\right]^{-1}$$

$$P_k = \sum_{i=1}^{k} p^{k-i} \frac{k!}{i!} \prod_{j=0}^{k-i} \beta^{-1}_j P'_k \text{ for } i = 1 \text{ to } k - 1.$$ 

We then have

$$AP_3 = \sum_{i=1}^{k} P_i$$

Further we have

$$u = 1 - P_0$$

$$D_3(u) = (k - u / \rho) / u.$$ 

Figure 7 illustrates this system and $D_3(u)$ is plotted in Figure 9 for a system with $k = m = 8$, $b = 1$ or 8.
**System 4:** Banyan switch and distributed shared memory architecture.

Set $p = \lambda/\mu$ and

$$P_0 = \left[\sum_{i=0}^{k} \frac{k!}{(k-i)!} \prod_{j=1}^{i} \frac{1}{c(j)}\right]^{-1}, \quad P_i = P_0 \frac{k!}{(k-i)!} \prod_{j=1}^{i} \frac{i}{c(j)},$$

where $c(j) = \lceil \log_2(i) \rceil$, $f_1(i) = \phi(f_{i-1}(i))$ and $\phi(i) = \begin{cases} \frac{i(2k - 0.5i - 1.5)}{2(k-1)} & i > 1 \\ 1 & i = 1 \end{cases}$.

Then we have

$$AP_4 = \sum_{i=1}^{k} iP_i$$

With $u(x) = \max(x, 0)$, let

$$L = \sum_{i=1}^{k} u(i - c(i))P_i,$$

then we have

$$u = 1 - P_0$$

and

$$D_4(u) = (k + Lp)(k - L)$$

In Figure 8 the System 4 architecture is illustrated and $D_4(u)$ is plotted in Figure 9.
4.3 A Parallel Implementation of the Algorithm Mapper

The allocation algorithm runs initially for a large value of the time frame $T$ and subsequently the value of $T$ is decreased until the number of clusters equals the parallelism of the graph in Step 1, or equals the number of processors in Step 2 (see the discussion at the end of
Section 1). The smallest value of $T$ for which Step 1 or Step 2 is completed is $T_{PAR}$. Note that in each iteration of the algorithm, the allocation heuristic is executed producing a number of clusters and schedule of modules for the current value of $T$. When $T$ changes, the output of the allocation algorithm changes as well. Moreover, the output of one iteration is not used in the next iteration. This attribute makes the algorithm a suitable candidate for parallel execution of the multisection extension of the bisection algorithm for locating zeros of functions. See [MIRA 69], [RICE 71] for further details. We have implemented a parallel version using the Sequent, a parallel machine which houses (in the configuration we used at the Computer Science Department at Purdue) 20 processors.

Any number $k \leq 20$ processors can be used. Our particular version for the parallel algorithm mapper is as follows.

1. Estimate $T$ as best one can.
2. Take as initial interval $[a, b] = [T/2, T]$ hoping that the guess is good enough for this interval to contain $T_{PAR}$. Set $T_0 = 0$.
3. Divide $[a, b]$ into $k - 1$ equal subintervals by the values $T_i = a + (i - 1)(b - a)/(k - 1), i = 1, 2, \ldots, k$.
4. Assign the $i$th processor the value $T_i$ and run the allocation program with it.
5. Let $[T_j, T_{j+1}]$ be the interval such that (a) $T_{j+1}$ gives the smallest number of clusters, and (b) the number of clusters for $T_j$ is larger than that of $T_{j+1}$. If $j \neq 1$, take $[a, b]$ to be $[T_j, T_{j+1}]$. If $j = 1$, take $[a, b]$ to be $[\max(T_0, 2T_1 - T_2), T_1]$. If $b - a \leq \varepsilon$ = convergence tolerance, then go to 6, otherwise go to 3.
6. Set $T = a$ and produce output. If the number of clusters is less than the parallelism of the application, then find that interval $[T_l, T_{l+1}]$ where $l$ is the smallest index where the number of clusters exceeds those of $T$. Set $T_0 = T_l$, $a = T_l$, $b = T_k$ and go to 3.

A little analysis shows that this algorithm has speed up of $\log_2(k + 1)$ which is worthwhile for small values of $k$.

4.4 Time Complexity and Optimality of the Algorithm Mapper

The time complexity of the allocation algorithm is approximately proportional to the number of links in the application graph. This is a consequence of the search for the links that carry high communication and the formation of lists of such links until a merge can be performed. After each merge, the same search starts over again. In Table 2, timing data from three representative applications are shown. These data strongly suggest that the time complexity is no worse than linear in the number of links in $G(A)$. 
<table>
<thead>
<tr>
<th>Application</th>
<th>Number of links</th>
<th>Time</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real time</td>
<td>27</td>
<td>1.18</td>
<td>22.9</td>
</tr>
<tr>
<td>PDE collocation</td>
<td>77</td>
<td>4.1</td>
<td>18.8</td>
</tr>
<tr>
<td>Robot arm</td>
<td>230</td>
<td>21.1</td>
<td>10.9</td>
</tr>
</tbody>
</table>

Table 2. Experimental data on the complexity of the allocation algorithm. Time for execution is given in seconds on a VAX 11/780 along with the ratio of links to time.

A symmetric and computationally simple partition of a Cholesky decomposition application is used to test the optimality of the algorithm mapper. A \((4 \times 4)\) partition of the application graph is discussed in detail in Section 5.1. The degree of parallelism of the graph is four and four clusters were obtained for System 2 with four processors. By using simulation, \(T_{\text{PAR}}\) has been shown to be the minimum possible elapsed time for this application (see Figure 13). Moreover, the processor utilizations were well balanced (see Figure 12).

5. EXAMPLE STUDIES USING THE ALGORITHM MAPPER

Several applications have been tested to confirm the analysis and heuristics. All except one are actual computations for specific applications. The exception is an application whose graph is stochastic and has been used in [BATR 78] to illustrate that a merging heuristic will perform satisfactory for any general stochastic graph. In [BATR 78], a similar allocation algorithm was applied to this application, but it was not actually programmed and a number of mistakes have been made because of limited testing. We give an account of this in [HOUS 88] and it is not described further here.

The three real applications tested are, (1) a Cholesky decomposition algorithm for the solution of linear equations, (2) a partial differential Equation (PDE) problem, using the collocation method, and (3) the solution to the Newton-Euler equations, for the mechanical movement of a robotic elbow manipulator. In all three cases, a partitioning of the application is given and the algorithm mapper system used to obtain an allocation. The partitions are from previous work: PDE collocation application, [HOUS 87]; Cholesky decomposition, [OLEA 85]; robot elbow application, [KASH 85].

5.1 Cholesky Decomposition Application

We consider the parallelization of Cholesky algorithm for the factorization of symmetric matrices. Figure 10 shows the computation of each module [OLEA 85]. We initially used a data flow language SIMON [FUJI 85] to specify the computational modules and their
communication and synchronization requirements. We then executed these programs using the SIMON non-shared memory multiprocessor simulator. Figure 11 shows the graph $G(A)$ and the values of the various workload parameters (node processing time and blocking time, communication traffic among nodes) obtained by setting SIMON's switching delay to zero (i.e., assume there are no communication delays). The blocking time or algorithm synchronization delay of a module is the time that the module must wait for its inputs before its computation can start. The processing time is the time to execute the code in a module. These times are given in the vectors $b$ and $m$ of Figure 11. The application was also run by our preprocessor and similar input data for the allocation were obtained.

The matrix is partitioned into blocks so that the maximum assigned degree of parallelism is equal to the number of processors available, so we have as many processors as the assigned degree of parallelism of the application. A single bus multiprocessor and distributed shared memory architecture is used. Thus only Step 1 of the mapping problem needs to be taken. We have run this application for various assigned degrees of parallelism (number of processors), we only report on the case of assigned degree of parallelism four here. Figure 12 shows the output of the allocation algorithm. Note that no internal datablocks are necessary in this application, since no reference to data is necessary. By default the number of internal datablocks are set equal to one per module of size zero.
$K := 0$

WHILE (TRUE) BEGIN

$K := K + 1$;

IF ($K = i$ and $K = j$) THEN

BEGIN

$X := \text{sqrt}(X)$;

IF ($j \neq N$) PUT (OUT_EAST,$X$);

IF ($i \neq N$) PUT (OUT_SOUTH,$X$);

BREAK

END

ELSE IF ($K = j$) THEN

BEGIN

GET (IN_SOUTH,$Y$);

$X = X/Y$

IF ($j \neq N$) PUT (OUT_EAST,$X$);

IF ($i \neq N$) PUT (OUT_SOUTH,$Y$);

BREAK;

END

ELSE IF ($K = i$) THEN

BEGIN

GET (IN_EAST,$Y$);

$X = X/Y$

IF ($j \neq N$) PUT (OUT_EAST,$Y$);

IF ($i \neq N$) PUT (OUT_SOUTH,$X$);

BREAK

ELSE

BEGIN

GET (IN_SOUTH,$Y$);

GET (IN_EAST,$Z$);

$X = X - Y \cdot Z$;

IF ($i \neq N$) PUT (OUT_SOUTH,$Y$);

IF ($j \neq N$) PUT (OUT_EAST,$Z$);

END

END WHILE

Figure 10. Computation of node $(i,j)$ in the SIMON data flow language for a parallel block Cholesky algorithm. The operations $\text{sqrt}(X)$, $X/Y$, etc. are on block submatrices of the matrix to be factored.
Figure 11. Procedence graph $G(A)$ of the parallel Cholesky decomposition algorithm for a 4 by 4 block decomposition of a symmetric matrix. The numbering of modules is specified in each node and the processing times and blocking times are given in this order. The communication traffic is indicated as a weight on the links of the graph. (Module processing times $m = (24, 30, 24, 27, 31, 32, 27, 34, 38, 39, 21, 28, 34, 45)$; Module blocking times $b = (0, 4, 14, 24, 15, 27, 37, 48, 33, 45, 61, 66, 51, 64, 76, 88)$.

From Figure 12 we see that $T_{PAR} = 250$, $T_{seq} = 622$ (139+152+170+161), and $T_{REAL} = 336$. The processor utilization $u^T_p$ is (total cluster processing time)/(time frame $T$), for example $u^T_p = 139/250 = 0.566$. The number $AP$ of active processors is calculated from $\sum u^T_p$ to be 2.488. The speed up is $S = T_{seq}/T_{REAL} = 1.851$. We see that the following relationship holds.

$$1 \leq S = T_{seq}/T_{REAL} = 1.851 \leq T_{seq}/T_{PAR} = 2.488 \leq AP = 2.488$$

When the time frame $T$ is decreased below $T_{PAR}$, the result is more and more clusters. We have used simulation to verify (see Figure 13) that the minimum elapsed time of this application does indeed occur at $T = T_{PAR}$. See [HOUS 87] for more details.
Figure 12. Output of a $4 \times 4$ case of the Cholesky decomposition application of Figures 10 and 11.
Figure 13. Simulation results for the elapsed time and average transmission delay of the different $G'(A)$'s obtained for the Cholesky $4 \times 4$ decomposition application.

5.2 PDE Collocation Application

This PDE application is complete from the description of a problem in a very high level language through the graphical display of the computed solution. Figure 14 shows a schematic diagram with the steps of the computation, it has a assigned degree of parallelism 6. Not shown in Figure 14 are vertical data communication paths between corresponding nodes on the fan-in/fan-out of the tree structure in the center. The PDE problem solved is a general one on a non-rectangular domain. A multifront method is used based on a nested dissection partition of the domain. Gauss elimination is used to eliminate unknowns in the interior of each domain. The numbers shown at each node are the ‘computation units’ for a particular instance of this problem corresponding to using a 20 by 20 mesh, a finite element method with cubic basis functions, and a 40 by 40 plotting grid. The domain boundary has 3 pieces and, at the fourth step, all but one processor are working on the interior of the domain. A computational unit here is about 1000 arithmetic operations, plus associated memory and control operations.

Figure 15 shows the same graph with the additional communication paths. These paths transfer $LU$ factorizations of subblocks of the linear systems from the initial solve phase to the back substitution phase. The numbers shown are data transfers required in units of 1000 words. The complete graph as displayed by the algorithm mapper is shown in Figure 16 (the communication has been scaled by a factor of 1000). We again use a single bus multiprocessor and distributed shared memory architecture with the number of processors equal to the assigned degree of parallelism of the application. Thus only Step 1 of the mapping problem needs to be taken. In Figure 17 the output from the algorithm mapper of the allocation algorithm is shown. All modules having the same number are allocated to the same processor.
Read Problem Description

Process 3 Boundary Pieces
Merge Boundary Data Structures
Identify All Elements
Process Interior/Boundary Elements

Create 6 Frontal Areas

Interior Assembly + Elimination

Merger and Elimination in Neighboring Areas
Merger and Elimination in Neighboring Areas
Merger and Elimination in Neighboring Areas
Backsolve for Neighboring Unknowns
Backsolve for Neighboring Unknowns
Backsolve for Unknowns
Tabulate Answer for Plot
Tabulate Flow Field

Create Plot Data Structure

Compute Contours, Color for 6 Views

Figure 14. Annotated graph $G(A)$ for the PDE application. The numbers at the nodes are units of computation to be done.
Figure 15. The graph of Figure 14 showing additional communication edges and the number of units to be communicated along each edge.

Figure 18 shows another form of the output of the algorithm mapper system. No internal data blocks exist in this application, thus datablocks by default are set to equal one per module of size zero.
Figure 16. The complete graph of the PDE collocation application as displayed by the algorithm mapper system. This is at the input stage, the communication along edges is shown and the computational modules numbered. At the upper left is the message PANEL discussed in Section 4.2.2.
Figure 17. The output of the algorithm mapper system for the PDE collocation application. The allocation of computational modules to processors is indicated by the numbering of the nodes. Nodes with the same number are allocated to the same processor.
Figure 18. Additional output of the algorithm mapper system for the PDE collocation application.
Similar calculations can be made from Figure 18 as with the previous application. Thus $T_{PAR} = 70$, $T_{seq} = 252.587$, $T_{REAL} = 71$ and $AP = .512 + .6968 + .2375 + .5 + 4513 + .554 + .6567 = 3.608$. The speed up is 3.54 and we have

$$1 \leq S = T_{seq}/T_{REAL} = 3.54 \leq T_{seq}/T_{PAR} = 3.6 \leq AP = 3.608$$

See [HOUS 87] for more details.

### 5.3 Robotic Elbow Manipulator Application

In [KASH 85] a partition of a robot elbow manipulator computation is given at the equation level, i.e., the computational modules represent the solution of an equation. We use this partition with slight modifications. Figure 19 shows the precedence graph of this application; the numbers assigned to the modules identify them. Modules are organized on different levels and modules on the same level can be executed in parallel. The execution times of modules are given in [KASH 85] in msec for an Intel 8087 processor. The partition in [KASH 85] requires little communication among modules and communication is not considered there. We have modified this by including in the execution time, $t_e$, of a module, both the processing time $t_x$ and communication time $t_y$, that is $t_e = t_x + t_y$. We also assume that synchronization delay is included in $t_e$. If a module has several outgoing links, we distribute the communication times $t_y$ uniformly among them. We have modified this application to obtain three applications with different behaviors. We take the execution time $t_e$ to be constant and divide into communication and computation so that the values of the computation/communication ratio $r = t_x/t_y$ are $1/10, 1/1$ and $10/1$. The latter corresponds closely to the original application.

We vary the computation/communication ratio $r$ of this application to study its effect and to show various properties of the allocation algorithm. Usually a fine partition requires more communication than a coarse partition, but at the same time, the number of modules is increased or decreased respectively. By varying $r$, we change the partition grain without affecting the number of modules. Let $c_p$ be the number of clusters obtained by the algorithm in Step 1, then in general $c_p$ is greater than or equal to the parallelism of the application.

For this application, a comparative study has been performed for two architectures, the multiple bus and Banyan switch, both with distributed shared memory.
Figure 19. The precedence graph of the 105 computational modules of the robot elbow manipulator. The maximum assigned degree of parallelism is 11.
Schedules for the Multiple Bus and Distributed Shared Memory Architecture

This application, shown in Figure 19, is allocated to a \( k \times m \times b \) multiple bus and distributed shared memory (see Section 4.2.3) architecture. We choose the number \( k \) of memory modules \( (m) \) and processors \( (k) \) to be equal. A \( 13 \times 13 \times 1 \) and a \( 13 \times 13 \times 8 \) system are used. The number \( c_p \) of parallel clusters obtained is equal to the number of processors in the system. Outputs of the heuristic algorithm for different values of the computation/communication ratio \( r \), and the two multiple bus systems are shown in Figures 20 and 21. We give a schedule of assigned modules to each processor, the processor utilization \( u_p \) and the optimal time frames \( T_{PAR} \) obtained. We also give the total processor utilization which includes both the processing and communication delay times required by all modules assigned to the same processor.

Comparing Figures 20 and 21, we observe for the case of \( r = 1, 1/10 \), that \( T_{PAR} \) is shorter for the 8 bus system (Figure 21a,b) than for the 1 bus system (Figure 20a,b). This is expected, since the 8 bus system has higher bandwidth and thus less delay. In addition, in each case different processor schedules are obtained. This is the effect that queueing delay has on the scheduling. In the case of \( r = 10/1 \), queueing delay does not play a significant role in scheduling and the 1 bus (Figure 20c) and 8 bus (Figure 21c) systems have identical schedules. We have measured the utilization for this case and we find that it is in an area (see Figure 9) where there is no significant difference in the queueing delay for 1 and 8 bus systems. We note that processor utilizations are low due to the parallelism constraint between modules.

In Figure 22, a schedule is shown which is based on minimizing the amount of communication among modules (the \( H' \) matrix) without assigning a cost to it, i.e., the queueing delay \( D(u) \) is zero. A different schedule is produced as compared to the same \( r = 1/10 \) case for the 1 bus system in Figure 20. The significant queueing delay in the 1 bus system also increases \( T_{PAR} \) substantially. If we compare Figure 22 with the \( r = 1/10 \) for the 8 bus system in Figure 21b, then one observes that the same schedule has been obtained. The value of \( T_{PAR} \) is practically unchanged. This is due to the fact that a \( 13 \times 13 \times 8 \) system is practically a crossbar switch and queueing delay is almost non-existent. One should not forget that if there are modules with communication among them, they are scheduled into the same processor unless they are parallel modules. This eliminates most of the high communication between modules.
Figure 20a. Schedule of module assignments of the robot elbow manipulator for $13 \times 13 \times 1$ multiple architecture and moderate computation/communication ratio, $r = 1/1$. 

Figure 20b. Schedule of module assignments of the robot elbow manipulator for the $13 \times 13 \times 1$ multiple bus architecture and low computation/communication ratio, $r = 1/10$. 

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Processor id & Total (%) & Utilization (%) & Modules assigned \\
\hline
1 & 51.13 & 45.33 & 1 57 61 63 66 67 69 72 92 98 104 \\
2 & 70.31 & 68.67 & 3 37 49 59 65 71 73 86 87 93 97 99 100 101 102 103 105 \\
3 & 50.50 & 36.49 & 3 7 9 24 36 42 48 54 78 \\
4 & 35.34 & 20.85 & 4 10 16 46 \\
5 & 35.34 & 20.85 & 5 11 17 47 \\
6 & 36.35 & 21.98 & 6 12 18 60 \\
7 & 36.60 & 21.76 & 8 14 15 29 \\
8 & 51.65 & 38.30 & 13 41 43 53 55 62 68 91 \\
9 & 71.26 & 57.34 & 19 20 21 22 23 26 31 32 33 34 35 \\
10 & 83.36 & 64.59 & 25 30 40 52 76 77 80 84 90 95 \\
11 & 51.77 & 32.41 & 27 38 44 50 74 88 \\
12 & 58.79 & 46.69 & 28 55 58 64 70 81 85 96 \\
13 & 75.02 & 55.52 & 39 45 51 75 79 82 83 89 94 \\
\hline
\end{tabular}
\caption{Schedule of module assignments of the robot elbow manipulator for $13 \times 13 \times 1$ multiple architecture and moderate computation/communication ratio, $r = 1/1$.}
\end{table}
System: $13 \times 13 \times 1, r = 10/1, T_{\text{PAR}} = 3304$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.97</td>
<td>12.38</td>
<td>1 27 30 61 67</td>
</tr>
<tr>
<td>2</td>
<td>42.08</td>
<td>40.16</td>
<td>2 6 12 18 37 49 60 87</td>
</tr>
<tr>
<td>3</td>
<td>89.98</td>
<td>88.03</td>
<td>3 9 39 41 45 51 53 75 79 83 89 91 94 100</td>
</tr>
<tr>
<td>4</td>
<td>26.67</td>
<td>25.31</td>
<td>4 10 16 46</td>
</tr>
<tr>
<td>5</td>
<td>26.67</td>
<td>25.31</td>
<td>5 11 17 47</td>
</tr>
<tr>
<td>6</td>
<td>67.46</td>
<td>65.20</td>
<td>7 8 14 15 24 29 36 42 48 54 78</td>
</tr>
<tr>
<td>7</td>
<td>25.25</td>
<td>24.76</td>
<td>13 43 57 63 69</td>
</tr>
<tr>
<td>8</td>
<td>70.90</td>
<td>69.60</td>
<td>19 20 21 22 23 26 31 32 33 34 35</td>
</tr>
<tr>
<td>9</td>
<td>34.31</td>
<td>33.29</td>
<td>25 28 55 66 72 92 98 104</td>
</tr>
<tr>
<td>10</td>
<td>71.10</td>
<td>70.70</td>
<td>38 44 50 73 74 82 88 93 99 102 105</td>
</tr>
<tr>
<td>11</td>
<td>74.85</td>
<td>74.00</td>
<td>40 52 76 77 80 84 90 95 101</td>
</tr>
<tr>
<td>12</td>
<td>65.36</td>
<td>64.37</td>
<td>56 59 62 65 68 71 86 97 103</td>
</tr>
<tr>
<td>13</td>
<td>51.73</td>
<td>51.17</td>
<td>58 64 70 81 85 96</td>
</tr>
</tbody>
</table>

Figure 20c. Schedule of module assignments of the robot elbow manipulator for the $13 \times 13 \times 1$ multiple bus system architecture and high computation/communication ratio, $r = 10/1$.

System: $13 \times 13 \times 8, r = 1/1, T_{\text{PAR}} = 2125$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>58.87</td>
<td>58.35</td>
<td>1 61 66 67 72 73 87 92 93 98 99 100 101 102 103 104 105</td>
</tr>
<tr>
<td>2</td>
<td>44.27</td>
<td>30.12</td>
<td>2 27 37 41 49 53 91</td>
</tr>
<tr>
<td>3</td>
<td>81.15</td>
<td>62.35</td>
<td>3 9 39 45 51 75 79 82 83 89 94</td>
</tr>
<tr>
<td>4</td>
<td>32.92</td>
<td>21.65</td>
<td>4 10 16 46</td>
</tr>
<tr>
<td>5</td>
<td>32.92</td>
<td>21.65</td>
<td>5 11 17 47</td>
</tr>
<tr>
<td>6</td>
<td>34.00</td>
<td>22.82</td>
<td>6 12 18 60</td>
</tr>
<tr>
<td>7</td>
<td>74.58</td>
<td>55.76</td>
<td>7 8 14 15 24 29 36 42 48 54 78</td>
</tr>
<tr>
<td>8</td>
<td>61.06</td>
<td>52.24</td>
<td>13 43 56 59 62 65 68 71 86 97</td>
</tr>
<tr>
<td>9</td>
<td>70.36</td>
<td>59.53</td>
<td>19 20 21 22 23 26 31 32 33 34 35</td>
</tr>
<tr>
<td>10</td>
<td>27.30</td>
<td>22.12</td>
<td>25 57 63 69</td>
</tr>
<tr>
<td>11</td>
<td>57.89</td>
<td>48.47</td>
<td>28 55 58 64 70 81 85 96</td>
</tr>
<tr>
<td>12</td>
<td>76.96</td>
<td>64.71</td>
<td>30 40 52 76 77 80 84 90 95</td>
</tr>
<tr>
<td>13</td>
<td>44.00</td>
<td>31.29</td>
<td>38 44 50 74 88</td>
</tr>
</tbody>
</table>

Figure 21a. Schedule of module assignments of the robot elbow manipulator for the $13 \times 13 \times 8$ multiple bus architecture and moderate computation/communication ratio, $r = 1/1$. 
system: $13 \times 13 \times 8$, $r = 1/10$, $T_{PAR} = 3881$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%)</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.86</td>
<td>2.34</td>
<td>1 13 25 43 57 63 69</td>
</tr>
<tr>
<td>2</td>
<td>16.53</td>
<td>4.45</td>
<td>2 7 24 36 37 42 48 49 54 78 87</td>
</tr>
<tr>
<td>3</td>
<td>13.22</td>
<td>4.82</td>
<td>3 9 58 64 70 81 85 96</td>
</tr>
<tr>
<td>4</td>
<td>13.51</td>
<td>2.15</td>
<td>4 10 16 46</td>
</tr>
<tr>
<td>5</td>
<td>13.51</td>
<td>2.15</td>
<td>5 11 17 47</td>
</tr>
<tr>
<td>6</td>
<td>13.53</td>
<td>2.27</td>
<td>6 12 18 60</td>
</tr>
<tr>
<td>7</td>
<td>13.88</td>
<td>2.25</td>
<td>8 14 15 29</td>
</tr>
<tr>
<td>8</td>
<td>16.83</td>
<td>5.93</td>
<td>19 20 21 22 23 26 31 32 33 34 35</td>
</tr>
<tr>
<td>9</td>
<td>11.30</td>
<td>6.04</td>
<td>27 28 61 66 67 72 73 92 93 98 99 100 101 102 103 104 105</td>
</tr>
<tr>
<td>10</td>
<td>26.23</td>
<td>6.21</td>
<td>30 39 45 51 55 75 79 82 83 89 94</td>
</tr>
<tr>
<td>11</td>
<td>15.92</td>
<td>3.12</td>
<td>38 44 50 74 88</td>
</tr>
<tr>
<td>12</td>
<td>20.40</td>
<td>8.06</td>
<td>40 41 52 53 76 77 80 84 90 91 95</td>
</tr>
<tr>
<td>13</td>
<td>12.65</td>
<td>5.06</td>
<td>56 59 62 65 68 71 86 97</td>
</tr>
</tbody>
</table>

Figure 21b. Schedule of module assignments of the robot elbow for the $13 \times 13 \times 8$ multiple bus architecture and low computation/communication ratio, $r = 1/10$.

system: $13 \times 13 \times 8$, $r = 10/1$, $T_{PAR} = 3304$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%)</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.96</td>
<td>12.38</td>
<td>1 27 30 61 67</td>
</tr>
<tr>
<td>2</td>
<td>42.02</td>
<td>40.16</td>
<td>2 6 12 18 37 49 60 87</td>
</tr>
<tr>
<td>3</td>
<td>89.83</td>
<td>88.03</td>
<td>3 9 39 41 45 51 53 75 79 83 89 91 94 100</td>
</tr>
<tr>
<td>4</td>
<td>26.63</td>
<td>25.31</td>
<td>4 10 16 46</td>
</tr>
<tr>
<td>5</td>
<td>26.63</td>
<td>25.31</td>
<td>5 11 17 47</td>
</tr>
<tr>
<td>6</td>
<td>67.40</td>
<td>65.20</td>
<td>7 8 14 15 24 29 36 42 48 54 78</td>
</tr>
<tr>
<td>7</td>
<td>25.24</td>
<td>24.76</td>
<td>13 43 57 63 69</td>
</tr>
<tr>
<td>8</td>
<td>70.87</td>
<td>69.60</td>
<td>19 20 21 22 23 26 31 32 32 22 34 35</td>
</tr>
<tr>
<td>9</td>
<td>34.28</td>
<td>33.29</td>
<td>25 28 55 66 72 92 98 104</td>
</tr>
<tr>
<td>10</td>
<td>71.08</td>
<td>70.70</td>
<td>38 44 50 73 74 82 88 93 102 105</td>
</tr>
<tr>
<td>11</td>
<td>74.82</td>
<td>74.00</td>
<td>40 52 76 77 80 84 90 95 101</td>
</tr>
<tr>
<td>12</td>
<td>65.33</td>
<td>64.37</td>
<td>56 59 62 65 68 71 86 97 103</td>
</tr>
<tr>
<td>13</td>
<td>51.72</td>
<td>51.17</td>
<td>58 64 70 81 85 96</td>
</tr>
</tbody>
</table>

Figure 21c. Schedule of module assignments of the robot elbow manipulator for the $13 \times 13 \times 8$ multiple bus architecture and high computation/communication ratio, $r = 10/1$. 
- 52 -

system: none, $r = 1/10$, $T_{PAR} = 3880$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%)</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.78</td>
<td>2.34</td>
<td>1 13 25 43 57 63 69</td>
</tr>
<tr>
<td>2</td>
<td>16.38</td>
<td>4.45</td>
<td>2 7 24 36 37 42 48 49 54 78 87</td>
</tr>
<tr>
<td>3</td>
<td>13.12</td>
<td>4.82</td>
<td>3 9 58 64 70 81 85 96</td>
</tr>
<tr>
<td>4</td>
<td>13.37</td>
<td>2.15</td>
<td>4 10 16 46</td>
</tr>
<tr>
<td>5</td>
<td>13.37</td>
<td>2.15</td>
<td>5 11 17 47</td>
</tr>
<tr>
<td>6</td>
<td>13.39</td>
<td>2.27</td>
<td>6 12 18 60</td>
</tr>
<tr>
<td>7</td>
<td>13.73</td>
<td>2.25</td>
<td>8 14 15 29</td>
</tr>
<tr>
<td>8</td>
<td>16.69</td>
<td>5.92</td>
<td>19 20 21 22 23 26 31 32 33 34 35</td>
</tr>
<tr>
<td>9</td>
<td>11.24</td>
<td>6.04</td>
<td>27 28 61 66 67 72 73 92 93 98 99 100 101 102 103 104 105</td>
</tr>
<tr>
<td>10</td>
<td>25.97</td>
<td>6.20</td>
<td>30 39 45 51 55 75 79 82 83 89 94</td>
</tr>
<tr>
<td>11</td>
<td>15.76</td>
<td>3.11</td>
<td>38 44 50 74 88</td>
</tr>
<tr>
<td>12</td>
<td>20.24</td>
<td>8.05</td>
<td>40 41 52 53 76 77 80 84 90 91 95</td>
</tr>
<tr>
<td>13</td>
<td>12.55</td>
<td>5.06</td>
<td>56 59 62 65 68 71 86 97</td>
</tr>
</tbody>
</table>

Figure 22. Schedule of module assignments of the robot elbow manipulator. The queueing delay is set to zero and a low computation/communication ratio, $r = 1/10$ is used.

*Schedules: Banyan Switch and Distributed Shared Memory Architecture*

The results of applying the allocation algorithm to this application with the Banyan switch and distributed shared memory architecture are shown in Figure 23a,b,c. Twelve or thirteen parallel clusters are obtained. Our observation has been that for symmetric application graphs or almost symmetric graphs, the number of clusters obtained is equal to the assigned degree of parallelism of the application. When the application graph does not possess any symmetry, like the one here, the number of clusters may be greater than the graph's assigned degree of parallelism. This is due to the allocation algorithm's mechanism of module merging which must satisfy the parallelism and time frame constraints at the same time. This is a limitation of our heuristic algorithm, since it does not exhaust all possible schedules.
**Table 1.** Schedule of module assignments of the robot elbow manipulator for the Banyan switch architecture and moderate computation/communication ratio, \( r = 1/1 \).

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%)</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49.54</td>
<td>45.20</td>
<td>1  57  61  63  66  67  69  72  92  98  104</td>
</tr>
<tr>
<td>2</td>
<td>74.22</td>
<td>72.99</td>
<td>2  37  49  59  65  71  73  82  86  87  93  97  99  102  103  105</td>
</tr>
<tr>
<td>3</td>
<td>40.67</td>
<td>26.44</td>
<td>3  6  9  12  18  60</td>
</tr>
<tr>
<td>4</td>
<td>31.62</td>
<td>20.79</td>
<td>4  10  14  46</td>
</tr>
<tr>
<td>5</td>
<td>31.62</td>
<td>20.79</td>
<td>5  11  17  47</td>
</tr>
<tr>
<td>6</td>
<td>71.65</td>
<td>53.56</td>
<td>7  8  14  15  24  29  36  42  48  54  78</td>
</tr>
<tr>
<td>7</td>
<td>28.82</td>
<td>22.60</td>
<td>13  30  43  56  62  68</td>
</tr>
<tr>
<td>8</td>
<td>67.58</td>
<td>57.18</td>
<td>19  21  22  22  23  26  31  32  33  34  35</td>
</tr>
<tr>
<td>9</td>
<td>60.56</td>
<td>52.20</td>
<td>25  39  45  51  75  79  83  89  94  100</td>
</tr>
<tr>
<td>10</td>
<td>46.79</td>
<td>32.32</td>
<td>27  38  44  50  74</td>
</tr>
<tr>
<td>11</td>
<td>55.60</td>
<td>46.55</td>
<td>28  55  58  64  70  81  85  96</td>
</tr>
<tr>
<td>12</td>
<td>87.66</td>
<td>78.64</td>
<td>40  41  52  53  76  77  80  84  90  91  95  101</td>
</tr>
</tbody>
</table>

**Figure 23a.** Schedule of module assignments of the robot elbow manipulator for the Banyan switch architecture and moderate computation/communication ratio, \( r = 1/1 \).

**Table 2.** Schedule of module assignments of the robot elbow manipulator for the Banyan switch architecture and low computation/communication ratio, \( r = 1/10 \).

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%)</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.79</td>
<td>2.34</td>
<td>1  13  25  43  57  63  69</td>
</tr>
<tr>
<td>2</td>
<td>16.39</td>
<td>4.45</td>
<td>2  7  24  36  37  42  48  49  54  78  87</td>
</tr>
<tr>
<td>3</td>
<td>13.22</td>
<td>4.82</td>
<td>3  9  58  64  70  81  85  96</td>
</tr>
<tr>
<td>4</td>
<td>13.38</td>
<td>2.15</td>
<td>4  10  16  46</td>
</tr>
<tr>
<td>5</td>
<td>13.38</td>
<td>2.15</td>
<td>5  11  17  47</td>
</tr>
<tr>
<td>6</td>
<td>13.40</td>
<td>2.27</td>
<td>6  12  18  60</td>
</tr>
<tr>
<td>7</td>
<td>13.75</td>
<td>2.25</td>
<td>8  14  15  29</td>
</tr>
<tr>
<td>8</td>
<td>16.71</td>
<td>5.92</td>
<td>19  20  21  22  23  26  31  32  33  34  35</td>
</tr>
<tr>
<td>9</td>
<td>11.24</td>
<td>6.04</td>
<td>27  28  61  66  67  72  73  92  93  98  89  100  101  102  103  104  105</td>
</tr>
<tr>
<td>10</td>
<td>26.00</td>
<td>6.20</td>
<td>30  39  45  51  55  75  79  82  83  89  94</td>
</tr>
<tr>
<td>11</td>
<td>15.77</td>
<td>3.11</td>
<td>38  44  50  74  88</td>
</tr>
<tr>
<td>12</td>
<td>20.26</td>
<td>8.05</td>
<td>40  41  52  53  76  77  80  84  90  91  95</td>
</tr>
<tr>
<td>13</td>
<td>12.56</td>
<td>5.06</td>
<td>56  59  62  65  68  71  86  97</td>
</tr>
</tbody>
</table>

**Figure 23b.** Schedule of module assignments of the robot elbow manipulator for the Banyan switch architecture and low computation/communication ratio, \( r = 1/10 \).
system: 13 processor, Banyan switch, $r = 10/1$, $T_{PAR} = 3306$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.95</td>
<td>12.37</td>
<td>1 27 30 61 67</td>
</tr>
<tr>
<td>2</td>
<td>42</td>
<td>40.14</td>
<td>2 6 12 18 37 49 60 87</td>
</tr>
<tr>
<td>3</td>
<td>89.88</td>
<td>87.99</td>
<td>3 9 39 41 45 51 53 75 79 83 89 91 94 100</td>
</tr>
<tr>
<td>4</td>
<td>26.62</td>
<td>25.30</td>
<td>4 10 16 46</td>
</tr>
<tr>
<td>5</td>
<td>26.61</td>
<td>25.30</td>
<td>5 11 17 47</td>
</tr>
<tr>
<td>6</td>
<td>67.36</td>
<td>65.17</td>
<td>7 8 14 15 24 29 36 42 48 54 78</td>
</tr>
<tr>
<td>7</td>
<td>25.23</td>
<td>24.75</td>
<td>13 43 57 63 69</td>
</tr>
<tr>
<td>8</td>
<td>70.83</td>
<td>69.57</td>
<td>19 20 21 22 23 26 31 32 33 34 35</td>
</tr>
<tr>
<td>9</td>
<td>34.27</td>
<td>33.27</td>
<td>25 28 55 66 72 92 98 104</td>
</tr>
<tr>
<td>10</td>
<td>71.05</td>
<td>70.67</td>
<td>38 44 50 73 74 82 88 93 99 102 105</td>
</tr>
<tr>
<td>11</td>
<td>74.79</td>
<td>73.96</td>
<td>40 52 76 77 80 84 90 95 101</td>
</tr>
<tr>
<td>12</td>
<td>65.30</td>
<td>64.34</td>
<td>56 59 62 65 68 71 86 97 103</td>
</tr>
<tr>
<td>13</td>
<td>51.69</td>
<td>51.14</td>
<td>58 64 70 81 85 96</td>
</tr>
</tbody>
</table>

Figure 23c. Schedule of module assignments of the robot elbow manipulator for the Banyan switch architecture and high computation/communication ratio, $r = 10/1$.

5.4 Reduction of Parallelism in the Robot Application

Here we investigate the possibility of using the algorithm mapper system's allocation algorithm to reduce the parallelism of an application. This corresponds to Step 2 of the mapping problem as described in Section 5. We use the robot application and the Banyan switch architecture. The number of parallel clusters obtained in Figure 23a,b,c are twelve or thirteen, and we now assume that only an 8 processor system is available and thus we need to reduce the number of clusters to 8 from 12 or 13. To reduce parallelism we use the same heuristic allocation algorithm with a simple modification, we eliminate the parallelism constraint. We use as the input to the heuristic algorithm for Step 2 the clusters obtained in Step 1. Each of these clusters is regarded as a single module and the communication between clusters forms the communication between modules. Thus the graph output from Step 1 is the input to Step 2, except that the parallelism constraints are removed. The communication cost is found as in Step 1. Since parallelism between the modules of the new graph is not a constraint, it is always feasible to cluster the modules into a predetermined number of processors (in this case 8), by adjusting appropriately the time frame $T$ parameter. The results for the three cases of $r$ used in Step 1 are presented in Figure 24a,b,c. Note the module 1 of Figure 24 corresponds to cluster 1 of Figure 23 and so on for the rest of the modules. We also note that processor utilizations are much higher than in Step 1.
The time frame $T_{PAR}$ may increase or decrease when the parallelism is reduced. If communication dominates the work, then $T_{PAR}$ should be smaller. One sees this to be the case when $r = 1/10$ (compare Figures 23b and 24b). If computation dominates the work, then $T_{PAR}$ should be larger because there are fewer processors to do the computation. One sees this to be the case when $r = 10/1$ (compare Figures 23c and 24c). In an intermediate case where communication and computation are of similar amounts, then the effect on $T_{PAR}$ is unclear. One sees for this particular application that $T_{PAR}$ is increased slightly (compare Figure 23a and 24a).

**system: 8 processor, Banyan switch, $r = 1/1$, $T_{PAR} = 2528$**

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43.34</td>
<td>39.54</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>64.94</td>
<td>63.86</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>82.49</td>
<td>59.51</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>62.68</td>
<td>46.86</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>75.82</td>
<td>65.44</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>59.12</td>
<td>50.02</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>40.93</td>
<td>28.27</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>62.66</td>
<td>54.76</td>
<td>12</td>
</tr>
</tbody>
</table>

**Figure 24a.** Parallelism reduction for the schedule shown in Figure 23a for the robot application and Banyan switch architecture. The number of clusters is reduced from 12 to 8.
system: 8 processor, Banyan switch, \( r = 1/10 \), \( T_{PAR} = 1278 \)

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80.13</td>
<td>28.30</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>76.15</td>
<td>34.92</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>78.42</td>
<td>23.50</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>66.31</td>
<td>15.86</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>43.17</td>
<td>20.16</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>72.18</td>
<td>26.86</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>70.24</td>
<td>27.81</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>40.13</td>
<td>19.17</td>
<td>13</td>
</tr>
</tbody>
</table>

**Figure 24b.** Reduction of parallelism for the schedules shown in Figure 23b for the robot application and Banyan switch architecture. The number of clusters is reduced from 13 to 8.

system: 8 processor, Banyan switch, \( r = 10/1 \), \( T_{PAR} = 3706 \)

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>74.25</td>
<td>73.10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>83.97</td>
<td>80.94</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>80.18</td>
<td>78.49</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>89.75</td>
<td>87.81</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>88.92</td>
<td>88.06</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>63.38</td>
<td>63.04</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>58.25</td>
<td>57.40</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>46.11</td>
<td>45.62</td>
<td>13</td>
</tr>
</tbody>
</table>

**Figure 24c.** Parallelism reduction for the schedules shown in Figure 23c for the robot application and Banyan switch architecture. The number of clusters is reduced from 13 to 8.

For comparison purposes, we also reduce the 13 cluster schedule obtained for the 13 processor multiple bus architectures to 8 clusters for an 8 processor system. The results are shown in Figures 25 and 26. Again, the processor utilizations are increased substantially and the \( T_{PAR} \) values decreased.
**Figure 25a.** Reduction of parallelism for the schedule shown in Figure 20a for the robot application and the multiple bus architecture. The assigned degree of parallelism is reduced from 13 to 8.

**Figure 25b.** Reduction of parallelism for the schedule shown in Figure 20b for the robot application and the multiple bus architecture. The assigned degree of parallelism is reduced from 13 to 8.
system: $8 \times 8 \times 1, r = 10/1, T_{PAR} = 3804$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72.34</td>
<td>71.22</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>81.81</td>
<td>78.86</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>78.12</td>
<td>76.47</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>87.44</td>
<td>85.56</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>86.64</td>
<td>85.79</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>61.75</td>
<td>61.42</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>56.76</td>
<td>55.92</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>44.93</td>
<td>44.45</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 25c. Reduction of parallelism for the schedule shown in Figure 20c for the robot application and the multiple bus architecture. The assigned degree of parallelism is reduced from 13 to 8.

system: $8 \times 8 \times 8, r = 1/1, T_{PAR} = 2421$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>82.00</td>
<td>78.66</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>89.32</td>
<td>72.26</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>71.20</td>
<td>54.71</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>79.34</td>
<td>58.01</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>65.44</td>
<td>48.93</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>61.73</td>
<td>52.23</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>89.00</td>
<td>76.18</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>50.79</td>
<td>42.53</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 26a. Reduction of parallelism for the schedule shown in Figure 21a for the robot application and the multiple bus architecture. The assigned degree of parallelism is reduced from 13 to 8.
system: $8 \times 8 \times 8$, $r = 1/10$, $T_{PAR} = 1134$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80.11</td>
<td>28.30</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>56.81</td>
<td>15.23</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>84.99</td>
<td>24.21</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>46.46</td>
<td>7.37</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>69.94</td>
<td>15.15</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>84.27</td>
<td>41.92</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>54.74</td>
<td>10.66</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>56.74</td>
<td>22.16</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 26b. Reduction of parallelism for the schedule shown in Figure 21b for the robot application and the multiple bus architecture. The assigned degree of parallelism is reduced from 13 to 8.

system: $8 \times 8 \times 8$, $r = 10/1$, $T_{PAR} = 3804$

<table>
<thead>
<tr>
<th>Processor id</th>
<th>Total (%) utilization</th>
<th>Utilization (%)</th>
<th>Modules assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>72.34</td>
<td>71.22</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>81.81</td>
<td>78.86</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>78.12</td>
<td>76.47</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>87.44</td>
<td>85.56</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>86.64</td>
<td>85.79</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>61.75</td>
<td>61.42</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>56.76</td>
<td>55.92</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>44.93</td>
<td>44.45</td>
<td>13</td>
</tr>
</tbody>
</table>

Figure 26c. Reduction of parallelism for the schedule shown in Figure 21c for the robot application and the multiple bus architecture. The assigned degree of parallelism is reduced from 13 to 8.

5.5 Performance Evaluation of Applications/Architecture Pairs

We now compare the performance of the Banyan switch architecture and multiple bus architecture for the robot elbow manipulator application. Three versions of the application are considered with computation/communication ratio values of $r = 1/1, 1/10$ and $10/1$. Both architectures have distributed shared memory and 8 processors, the multiple bus architectures
also have 1 or 8 busses. The primary comparison is on the basis of $T_{PAR}$, the shortest parallel execution time. We also show (1) the \textit{average total processor utilization} $u^I_p$ which, includes both processor time and queueing waits for communication, (2) the speed up, and (3) the \textit{efficiency} [SIEG 82] = (speed up)/$k$.

Figure 27 shows that the smallest $T_{PAR}$ value and the highest processor utilizations were obtained for $r = 1/10$, as one expects. The $8 \times 8 \times 8$ multiple bus architecture has a higher bandwidth than the Banyan network, and the schedule for an $8 \times 8 \times 8$ multiple bus architecture has the best performance. This demonstrates the usefulness of the mapping methodology in matching applications to architectures. Speed ups and efficiency factors for the 8 processor systems are presented in Figure 28.

<table>
<thead>
<tr>
<th>Banyan Switch Architecture</th>
<th>Multiple Bus Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r = 1$</td>
<td>(1 bus) $T_{PAR} = 2421$</td>
</tr>
<tr>
<td></td>
<td>$u^I_p = 73.68$</td>
</tr>
<tr>
<td></td>
<td>(8 busses) $T_{PAR} = 2529$</td>
</tr>
<tr>
<td></td>
<td>$u^I_p = 61.50$</td>
</tr>
<tr>
<td>$r = 1/10$</td>
<td>(1 bus) $T_{PAR} = 1278$</td>
</tr>
<tr>
<td></td>
<td>$u^I_p = 65.84$</td>
</tr>
<tr>
<td></td>
<td>(8 busses) $T_{PAR} = 1207$</td>
</tr>
<tr>
<td></td>
<td>$u^I_p = 67.05$</td>
</tr>
<tr>
<td>$r = 10/1$</td>
<td>(1 bus) $T_{PAR} = 3706.3$</td>
</tr>
<tr>
<td></td>
<td>$u^I_p = 73.10$</td>
</tr>
<tr>
<td></td>
<td>(8 busses) $T_{PAR} = 3804$</td>
</tr>
<tr>
<td></td>
<td>$u^I_p = 71.22$</td>
</tr>
</tbody>
</table>

\textbf{Figure 27.} Performance comparison of multibus and Banyan architectures for the robot elbow manipulator application. Three values of the computation/communication ratio $r$ are used. All architectures have 8 processors.
<table>
<thead>
<tr>
<th>Ratio $r$</th>
<th>Speed up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Banyan</td>
<td>Multiple bus</td>
</tr>
<tr>
<td></td>
<td>(1 bus)</td>
<td>4.836</td>
</tr>
<tr>
<td>$r = 1/1$</td>
<td>(8 busses)</td>
<td>4.836</td>
</tr>
<tr>
<td>$r = 1/10$</td>
<td>1.665 (1 bus)</td>
<td>1.763</td>
</tr>
<tr>
<td></td>
<td>(8 busses)</td>
<td>1.877</td>
</tr>
<tr>
<td>$r = 10/1$</td>
<td>5.744 (1 bus)</td>
<td>5.596</td>
</tr>
<tr>
<td></td>
<td>(8 busses)</td>
<td>5.596</td>
</tr>
</tbody>
</table>

Figure 28. Comparison of speed ups and efficiency for the 8 processor Banyan switch and multiple bus architectures.

From data in [KASH 85] the elapsed (sequential) time in a uniprocessor system, $T_{seq}$, can be calculated. In a uniprocessor system the communication cost is zero, so the sequential time depends heavily on the ratio $r$ of computation to communication. Then for each value of $r$, we have considered that we get $T_{seq}$ as shown in Figure 29.

<table>
<thead>
<tr>
<th>$r$</th>
<th>$T_{seq}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r = 1$</td>
<td>11709</td>
</tr>
<tr>
<td>$r = 1/10$</td>
<td>2128</td>
</tr>
<tr>
<td>$r = 10/1$</td>
<td>21281</td>
</tr>
</tbody>
</table>

Figure 29. Sequential elapsed time of the application for values of $r$.

In Figures 30 and 31 speed ups and efficiencies are given for the multiple bus architectures under the assumption that the number of processors equals the number of parallel clusters, i.e., after Step 1 of the algorithm is performed. This data is derived directly from the schedules in Section 5.3. Note that for $r = 1/10$ and the one or eight bus system, the speed up is actually less than 1, which indicates that the parallel system does worse than a single processor system. Thus, a partition where the communication is ten times the computation, is the worst partition of the three we have examined. This shows how our methodology helps us evaluate the various partitions of an application.
Figure 30. Speed up and efficiency data for the schedules of Figure 20a,b,c for the 13x13x1 multiple bus and distributed shared memory architecture.

<table>
<thead>
<tr>
<th></th>
<th>Speed up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>r = 1</td>
<td>5.308</td>
<td>.408</td>
</tr>
<tr>
<td>r = 1/10</td>
<td>.525</td>
<td>.040</td>
</tr>
<tr>
<td>r = 10/1</td>
<td>6.441</td>
<td>.495</td>
</tr>
</tbody>
</table>

Figure 31. Speed up and efficiency data for the schedule of Figure 21a,b,c for the 13 x13x8 multiple bus and distributed shared memory architecture.

<table>
<thead>
<tr>
<th></th>
<th>Speed up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>r = 1</td>
<td>5.510</td>
<td>.423</td>
</tr>
<tr>
<td>r = 1/10</td>
<td>.548</td>
<td>.042</td>
</tr>
<tr>
<td>r = 10/1</td>
<td>6.443</td>
<td>.495</td>
</tr>
</tbody>
</table>

We illustrate the performance analysis further for those schedules (see Figure 25 and 26) where the assigned degree of parallelism has been reduced. The speedup $S$ and average number of active processors $AP$, are shown in Figures 32 and 33. Note that the bounds discussed in Sections 3.3 and 4.1.6 hold. In the cases where communication is limited, i.e., $r = 10/1$ and $r = 1/1$, its cost is also negligible, thus $S = AP$. When $r = 1/10$, then the communication cost in terms of queueing delay is not negligible any more, and it affects the $T_{PAR}$ results. Thus $S = T_{seq}/T_{PAR}$ is lower than in the previous two cases. By definition, communication delay is not included in $AP$, thus $AP \geq S$ as stated previously and observed in both Figures 32 and 33. We may also calculate using $T_{REAL}$. For example for the schedule of Figure 20c, we obtain $AP = 6.4428$, $T_{seq} = 21281$, $T_{PAR} = 3304$, $T_{REAL} = 6113$ and the following relationships.

$$1 \leq S = T_{seq}/T_{REAL} = 3.481 \leq T_{seq}/T_{PAR} = 6.440 \leq AP = 6.4428$$
Figure 32. Speed up and active processors for the schedules of Figures 25 and 26 for the multiple bus and distributed shared memory architecture. The assigned degree of parallelism has been reduced from 13 to 8 in these schedules.

<table>
<thead>
<tr>
<th>Ratio r</th>
<th>$8 \times 8 \times 1$ system</th>
<th>$8 \times 8 \times 8$ system</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$AP$</td>
<td>$T_{seq}/T_{PAR}$</td>
</tr>
<tr>
<td>$r = 10/1$</td>
<td>5.5969</td>
<td>5.5968</td>
</tr>
<tr>
<td>$r = 1/1$</td>
<td>4.8351</td>
<td>4.835</td>
</tr>
<tr>
<td>$r = 1/10$</td>
<td>2.4144</td>
<td>1.763</td>
</tr>
</tbody>
</table>

Figure 33. Speed up and active processors for the schedules of Figure 24 for Banyan switch and distributed shared memory architecture. The assigned degree of parallelism has been reduced from 13 to 8 in the schedules.

<table>
<thead>
<tr>
<th>Ratio r</th>
<th>$AP$</th>
<th>$T_{seq}/T_{PAR}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r = 10/1$</td>
<td>5.7446</td>
<td>5.744</td>
</tr>
<tr>
<td>$r = 1/1$</td>
<td>4.6310</td>
<td>4.630</td>
</tr>
<tr>
<td>$r = 1/10$</td>
<td>1.9658</td>
<td>1.665</td>
</tr>
</tbody>
</table>

6. SUMMARY

We have formulated the mapping problem and described our algorithm mapper system and methodology. Four architectures have been considered and three realistic applications have been evaluated for them. These applications were Cholesky decomposition algorithm, a PDE collocation solution and a robot arm manipulation computation. The four architectures considered are: (1) single bus and shared common memory, (2) single bus and distributed shared memory, (3) multiple bus and distributed shared memory, and (4) Banyan switch and distributed shared memory. The allocation methodology has made use of the parallel architecture performance models to assign a cost to communication between parallel processors. This cost is the queueing delay in communicating messages between modules assigned to different processors. The allocation algorithm used is based on a merging heuristic that minimizes communication between processors in assigning parallel modules to different processors. The approach has also been used to evaluate various partitions of a single application.
We have evaluated the performance of application/architecture pairs. For example, we see that a high bandwidth parallel system will not always perform better for a particular application than a lower bandwidth system. Different partitions of the same application may "fit" better on different architectures. Our methodology provides the means for this evaluation. We have made extensive experimentation with the robot arm application with 105 modules. We see a dramatic improvement in execution speed up using even a suboptimal allocation method such as ours. In some small systems and applications, we are able to demonstrate optimality of results, but we do not expect this in general. The parallel architectures we have studied have shared memory and our current approach depends on this fact. We are pursuing a similar approach for nonshared memory architectures.

7. REFERENCES


IMACS, Rutgers University, 1984, pp. 191–194.


APPENDICES

APPENDIX I: Description of the Allocation Program

1. Main Program -- driver routine

It starts by calling the 'entry' procedure which reads the data for the problem and then calls 'calcmat' which initializes working matrices. Then it calls 'fpart' which implements one iteration of the algorithm with the initial time frame (T) that was given as input to the program. Depending on the value of 'mode' (input), it will call 'iterative' which calls 'fpart' repeatedly for various values of T by decrementing or increasing the time until it finds an interval [ta, tb] such that: |tb - ta| ≤ tolerance (see input) and the minimum number of processors have been obtained. Finally it will create the file "graph.g" for graphics software.

Then there is an parallelism reduction (optional, by setting 'mode' in input) for the case of Banyan networks and multiple bus architectures. In this case we can get the number of available processors we have in the network by providing this information (see 'kvalue' = # of processors available). For example, if the algorithm gives say 13 clusters, but we can have only 8 then by setting kvalue = 8, the parallelism reduction will reduce the number of clusters to 8 by deleting all the conflicts and calling repeatedly 'fpart' until we get 8 clusters with the minimum time frame T.

2. fpart

This subroutine implements the allocation algorithm. After initializing the working matrices needed for the current iteration it finds any datablocks that can initially be made local to a processor (call to 'findlocal' and 'makelocal' subroutine).

Then it finds processor pairs that have the largest interprocessor traffic and creates a list of these as candidates for merger ('fincands'). It selects the candidate pairs that causes the largest amount of data to be made local ('bcand'), and then selects the candidate pair that yields the lowest processor loading for merger ('scand').

If the processor loading for this pair is not too high, it merges those two, otherwise it puts this pair "on hold" so that it will not select it again ('putonhold'). The above is repeated until we exhaust all the possible candidates for merging.

Finally it calls 'calcutil' which calculates processor utilizations and then 'printsolution' which prints out the solution found (see output).
2. calmats

It calculates and initializes working clusters and matrices.

4. reconstruct

It is called from procedure ‘optimize’ before we start the parallelism reduction phase. By extracting the information from the current matrices that contain the data from the previous solution, we create new matrices for the new graph that will be input to the algorithm. The new graph is constructed in the following way: For each processor we create a node. For each node we get the processing and communication with the other nodes. In the new graph, we delete the conflicts so we can have a new merge.

5. allocate

The purpose of this procedure is to efficiently pack memory blocks into memory modules. Memblocks is the set of blocks that must be put into memory modules, mmc is the memory module capacity, and mat contains the sizes of the memory blocks. The procedure first calculates the first guess of how many memory modules it will take to contain the memory blocks. This number is tried and if there is no ‘success’, then the number is incremented until the memory blocks fit into the memory modules allocated.

The procedures which sort and pack the memory blocks require a different data structure than this procedure. Therefore the data must be converted before these routines can be used. On the first successful allocation the results are printed.

6. bestcands

This procedure is passed a list of candidates for merger by the main program. The function is to prune the list by allowing only the candidates that will provide the greatest amount of datablocks to be made local to remain in the list.

The procedure creates a test matrix ‘testmat’ for each of the candidate pairs which represents what the datablock reference matrix would be if that candidate pair were merged. Then the test matrix is passed to the ‘findlocal’ procedure which returns the total ‘amount’ of data that could be made local. This number is compared to a ‘best’ thus far and the candidate pair is left in the list only if is is greater or equal in value.
7. calcutil

The purpose of this procedure is to calculate the processor utilization of all the processors left in after partitioning. This is accomplished by calculating $\tau$ which is the total number of ITU's of information transferred through the interconnection scheme. From this the 'delayper-itu' can be found using the delay graph data. Data for four different delay functions are stored. The processor loading can then be calculated. The processor utilization is then simply $\frac{100 \times \text{loading}}{\text{info.tau}}$ (where info.tau is the allowed real time for processing, i.e. $T$).

If, in the course of calculating utilizations, a single process results in greater than 100 percent utilization, an error message is printed telling what process is the problem. This can happen if the allowed real time for processing is lower than what is required for that single process to run in a processor alone.

8. delay -- library of delay functions

(a) delay1
(b) delay2
(c) delay3
(d) delay 4

This function takes a list of coordinate pairs which represent points on a graph. Then given an 'abscissa' value it returns a calculated ordinate value based on a linear interpolation between the appropriate coordinate pairs.

9. findcands

This procedure searches through a matrix $b$ and creates a list of process pair candidates for merger. Numcands is the number of candidates found and "candlist" is the list of candidate pairs.

The values in the $b$ matrix represent communication between processes. The processes which have the 'largest' amount of interaction are the ones which are chosen for merger. The matrix is searched and a list of pairs with the largest interaction is built. A process pair is not eligible for merger if they are "on-hold" or are in the "conflict" matrix or if one of the processes is no longer "active". (Note: A pair is "on hold" means that this pair has already been examined and it was rejected.) A pair $(i,j)$ is in conflict if conflict $(i,j) = \text{true}$ and a pair is no longer "active" if it has already been merged.
10. findlocal

This procedure scans the matrix "omega" and composes a list of datablocks that are referenced by a single process. The elements of this list are candidates to be made local to that particular process. "Amount" refers to the total amount of data made local if the entire list was made local.

11. makelocal

This procedure takes the datablocks in "list" and makes them local by removing them from the "omega" matrix and placing them in the appropriate area of the datablock partition r. The datablocks are also removed from the common datablock set.

12. merge

Does all that is necessary computations to merge a process pair.

13. pack

This is a recursive procedure that attempts to pack items into boxes. It keeps rearranging the items until all fit into the given boxes or until it determines that there is no possible way to do it. If impossible, then the Boolean variable 'success' is set to 'false'.

There are three parts to an itemlist: 'items.id', which is the vector which contains the integers that identify the items; 'items.size', which is the vector that specifies the sizes of the items and 'items.num', which is the number of items in the list. 'Numboxes' specifies the number of boxes that are going to be used and 'space' is a vector that specifies the amount of space in each box. 'Packlist' is an array of sets which represents the partition of items that have been packed. The procedure operates as follows: upon being called the procedure tries to put the item on the bottom of the itemlist in a box, if it will not fit, it trys the next box and so on. If it cannot locate a box with enough space to place the item, it will return success=false to the calling program. However, if a space is found, then the item is removed from the list and put into the packlist, the amount of space required for the item is subtraced from the space vector, and the number of items is decreased by one. At this point, the pack procedure is called again to pack the remaining items. If it returns success=false, then the item is taken back from the packlist, the amount of space subtraced from the space vector is replaced, and the number of items is incremented by one, thus restoring the original status. If no other box can be found to hold it, success=false is returned.
14. pulonhold

The purpose of this procedure is to mark certain candidate pairs so that they cannot be selected for merger. This is done simply by setting to "true" the position in the Boolean 'onhold' matrix that corresponds to the given processor pair for all pairs in the candidate list (candlist).

A list of candidate pairs is put on hold when it is discovered that none of the candidates would yield a processor loading that is low enough for a merger. Putting on hold prevents reselection of the same candidates. (Note: Upon each actual merger, all candidates on hold are released because a merger will cause the processor loading calculations to be lessened due to less interconnection usage. Thus a candidate pair that failed the test for merger in one instance, might pass after other processes have been merged. This is handled as part of the merge procedure.)

15. selcand

The purpose of this procedure is to make a final selection from the list of candidates passed to it by selecting the candidate pair that will yield the lowest processor loading if merged. This is accomplished by first calculating $t_x$ which is the total number of ITU's of information transferred through the interconnection scheme. With this number, the 'delayperitu' is found by utilizing the graph function and the delay graph data. The 'delayperitu' is necessary in the formula to calculate loading. The predicted processor loading $t_{ij}$ (in TU's) of each processor pair as it is calculated. After calculation the value is stored in the vector 'tijvec'. As the $t_{ij}$ values are calculated, they are compared to a 'lowest', then the value of 'lowest' changes and that pair is remembered as having the lowest value by storing a pointer in the variable 'bestcand'.

It is this variable 'bestcand' which represents its final selection when it returns to the calling program. The Boolean variable 'processorloadingisnottoohigh' is set as an indicator that the lowest predicted processor loading is an acceptable value. The 'lowest' is compared to the value of 'eta*tau', that is, the processor utilization times the allowed real time for processing. If the 'lowest' value is within this range, then the merge would be acceptable and 'processorloadingisnottoohigh' would be set to 'true'.

16. sort

The purpose of this procedure is to bubblesort items before sending them to the pack procedure. The pack procedure is much more efficient timewise when packing, if it selects the largest items to pack first. The vector 'items.id' that contains the numbers that identify the
memory modules that are to be sorted, ‘items.size’ is a vector that contains the corresponding size of the given memory modules, and ‘items.num’ is the number of items in the list. Note that this procedure orders the items from smallest to largest, this is the way it should work as the pack procedure selects items from the bottom of the list first.

17. math library routines

This is the matmath (matrix mathematics) module. It is a collection of matrix operation procedures. The following procedures are used:

- outputmat
- outputmaff
- identify
- matmult
- matadd
- matinv
- scalmult
- transpose
- elemult
- rowmult
- sumelements (this is a function)

18. createdatagraph

Find the (x,y) coordinates for each process for input to a graphics program.

19. entry

Reads input, for complete description, see the INPUT section.

20. printsolution

The purpose of this program is to print out the processor numbers and utilizations and to allocate and print the required instruction memory modules and data memory modules for each given processor. This procedure uses the processor partition “s” and the datablock partition “r” which was developed by the algorithm to serve this purpose. The cluster “s” is scanned to find nonempty sectors, these represent a group of processes that are to be executed by the same processor. When found, a processor number is assigned and is printed along with the expected processor utilization. Then the associated instruction blocks are partitioned and assigned memory modules by the “allocate” procedure. It also causes the memory module assignment to be printed along with the efficiency of usage. The set of data blocks that are local to that
processor are then given a similar treatment.

After all of the processors and their associated information have been printed, the data blocks that could not be made local to any given processor are partitioned and their memory module assignments and efficiencies are printed out. At this point the program is finished. For a complete description of output, see the OUTPUT section referred to the output.

INPUT

The input of the allocation program is read exactly in the following sequence.

A. print level; (flag 111 - 000) [ integer ]

first flag (001): print info (debugging)
second flag (010): entry will start asking input
third flag (100): print info (debugging)

n.b. could have any combination of the above

1. the number of processes \((n)\)
2. the number of data blocks \((k)\)
3. the allowed real-time in TU's (time frame), for the complete execution of the application program \(T\)
4. the maximum allowed processor utilization (eta), (suggested to be 0.9)
5. the maximum number of itu's that can be transferred in the allowed real time \((C)\)

   (the maximum information handling capacity - C)
6. the instruction memory module size \((immc)\)
7. the data memory modules size \((dmmc)\)

B. The instruction block size vector \((x)\) reads \(n\) real numbers (one for each instruction block).

C. The data block sizes vector \((y)\) reads \(k\) real numbers (one for each data block).

D. The module expected execution time vector \((a)\) reads \(n\) real numbers (one for each process).

E. The module entrance probability vector \((e)\) reads \(n\) real numbers (one for each process).

F. The module transition probability matrix \((p)\) and the inter-module information (ITU's) transfer per control transition matrix \((\lambda)\) in the following way: For each module \((i)\), supply the number of modules that could follow this \(i\)-th module and then for each one of them, enter the module number, the probability that control will pass to this module and
the number of ITU's transferred to it, e.g., input for 1st module ($i = 1$):

2

3.1.0 180
5.0.5 120

so two modules follow the first module. Modules 3 with possibility 1.0 and ITU's passed are 180. Modules 5 with probability 0.5 and 120 ITU's passed to it.

G. The control data of the EXOR graph $(pp)$ for each module ($i$) supply the number of modules that could follow the $i$-th module, the module number for each one, and the probability to transfer, e.g., input for 1st module ($i = 1$):

1 3 1.0

so 1 module follows module 1, which is module 3 with probability 1.0.

H. The node-conflict table which indicates the modules not to be assigned to the same processor (conflict). (Note: if any two modules are meant to be run in parallel, then they cannot be ran by the same processor, so they are in conflict.) The input is given in the following manner: For each module, give the number of modules that are in conflict with the chosen module and then give the number of these modules, e.g., input for 3rd module ($i = 3$):

2

5 7

so 2 modules (5 and 7) are in conflict with module 3.

I. The module-data block reference matrix ($h$). For each module, give the number of data blocks referenced by this module and the datablock number for each one of them, e.g., input for the 3rd module ($i = 3$):

3

1 3

so 2 datablocks (1 and 3) are referenced by module 3.

K. The number of co-ordinate pairs needed to describe the delay graph (if no such pairs of numbers are used, then enter zero) and then give this number of pairs, e.g.,

3

0.0 1.0
0.5 1.1
0.9 1.5

(Note: First pair 'must' be (0.0, 1.0).)

L. The following data also needed:

1. $dtsize$: initial time is decremented by this $dtsize$. 
2. \textit{tolerance}: used by delay functions.

3. \textit{utilization bound}: suggested 100.

4. \textit{interflag}: choose one delay function (out of 4).
   (a) delay function 1
   (b) delay function 2
   (c) delay function 3
   (d) delay function 4

5. \textit{k-value}

6. mode
   0.
   1.
   2.
   3.

7. \textit{\epsilon-tolerance}: how close the iteration will come to the best solution.

\textbf{OUTPUT}

The output of the allocation program is as follows:

(1) \textbf{ALLOCATION SOLUTION AND WORKLOAD STATISTICS} number of modules is

estimated elapsed time is $T$ and capacity is $C$, time units
allowed real time for processing is (initial $T$), time units

(2)

\begin{tabular}{|c|c|c|c|}
\hline
id & memory instruction & module size & data blocks utilization \\
\hline
\end{tabular}

4. Statistics on processors utilization, (total utilization for every processor = processing + communication, and utilization of each one due to processing only). Also printed for each one the processes assigned.

\textbf{*** PROCESSOR UTILIZATION***}

\begin{tabular}{|c|c|c|c|}
\hline
id & total (%) & utilization & processes assigned \\
\hline
utilization & (%) & assigned \\
\hline
\end{tabular}
5. Statistics on application's communication requirements for each processor.

*** APPLICATION'S COMMUNICATION REQUIREMENTS ***

(in ITU's)

<table>
<thead>
<tr>
<th>id</th>
<th>data referenced via interconnection</th>
<th>interprocessor data transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>memory inter-module communication cost</th>
<th>interprocessor communication cost</th>
<th>total cluster processing time</th>
<th>total processor workload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

AVERAGE: | | | |

delay per ITU: __

6. Statistics on the common blocks assigned. IF there are common data blocks the output is:

<table>
<thead>
<tr>
<th>id</th>
<th>memory instruction</th>
<th>module size</th>
<th>common data blocks utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ELSE the output is:

"there are no common datablocks"
APPENDIX II: Instructions For Using the Preprocessor

This section summarizes the use of the preprocessor on one particular system. Filenames and directories may vary on your system.

After designing the algorithms to be simulated, create the main program and the functions you are going to use in one or different files. The purpose of the main program is only to initialize the data you may need and specify the sequence in which the modules are going to run. The last statement of the main program should be a call to 'finisO' so that the preprocessor will create the input file for the allocation program in the format that is read. If you set up the printflag it will also print the output explaining in detail the data collected. This is also useful for debugging purposes.

Next, create the files parameters.c and a makefile. The easiest approach is to copy the files parameters.c and sample.make from directory ~/alloc/pre/src and modify them to suit the allocation and directory. The make command is then invoked to compile the simulation program.

Automatic instruction timing is controlled by 'clock_onO' and 'clock_offO' and the 'simcc' command line option '-T file'. The T file contains a list of VAX instructions and supposed execution times; a complete list with 1 microsecond times is stored in ~/alloc/simcc/VAXinstructs. Default is zero msec for all instructions (if optin -T is not used). See also

In the directory ~/alloc/pre/sample there are some examples for using the preprocessor. One is the Cholesky decomposition.

A list of error messages appears in Appendix III.
APPENDIX III: Error Messages

It is possible to get the following error messages:

*** error >>> create: invalid process id 
  you will get this error when the argument 'pid' to
  process(code, pid, prob, parmptr)
  is: pid < 1 or pid > NPROCESSES.
  By default the NPROCESSES is 100 in the preprocessor
  If you need more, just edit the file src.c and then run 'make'

*** error >>> create: process id # already used
  that means that you already have created a process with id = #,
  but a process id must be unique

*** error >>> cobegin: nested cobegin's not allowed
  you can't have nested cobegin's probably you forgot a coend

*** error >>> coend: must follow a 'cobegin'
  probably you forgot a cobegin

*** error >>> coend: cobegin must be closed by a coend
  probably you forgot to close the last cobegin with a coend

*** error >>> numbering of processes must be sequential
  the numbering of processes must start at process #1 and
  continue without any gaps up to process #maxprocess