

Scheduling In PGMT

by

Joseph Collins

Ali Boroujerdi

Richard S. Stevens

September 29, 2002

The Processing Graph Method Tool (PGMT) product is being released under the GNU General Public License Version 2, June 1991 and related documentation under the GNU Free Documentation License Version 1.1, March 2000. <http://www.gnu.org/licenses/gpl.html>

Introduction

At NRL (Naval Research Laboratory) we wrote the PGM (Processing Graph Method) Specification [1] to describe the basic features of a support system for developing applications for a distributed network of processors. In the PGMT (PGM Tool) project we developed an implementation of PGM. This project was conducted with the support of the Advanced Systems Technology Office for Undersea Warfare in the Office of Naval Research. We demonstrated its ability to perform on a heterogeneous network of processors using the Unix operating system with the MPI (Message Passing Interface) communication protocol.

In PGMT, the execution of a Processing Graph involves interaction between the code of the Graph Class Library (GCL) and the PGM Execution Program (Pep). Graph execution in the GCL is described in the paper about distributed processing [2]. Also discussed there is the interaction between the GCL and Pep, as seen from the GCL point of view. We assume the reader has read that paper and is familiar with the ideas presented there. It would also be helpful to have read the paper on Graph construction [3].

Specifically, we assume that the reader is familiar with the PGM notion of a *Node*, which may be either a *Transition* or a *Place*. A *Transition* may be either a user-defined *Ordinary Transition* or a PGM-defined *Special Transition* (*Pack* or *Unpack*). A *Place* may be either a *Graph* or a *Graph Variable*. Each *Node* has a number of *input ports* and a number of *output ports*, by which they are connected to one another via directed arcs. A directed arc connects from a *Node* output port (in the *upstream Node*) to a *Node* input port (in the *downstream Node*), one of the two *Nodes* being a *Transition* and the other a *Place*. During graph execution, data tokens, which are passed from *Node* to *Node* along the directed arcs, are stored in *Places* and are processed in *Transitions*.

This paper discusses the functions of the Pep and how the Pep controls in Graph execution in a distributed processor environment, both during normal execution and during reassignment. We describe how the Pep performs the following functions:

- Initial assignment of *Transitions* to processors,
- Scheduling of *Transitions* for execution,
- Monitoring of Graph performance,
- Determination whether and when to reassign *Transitions* to processors,
- Determination of how to reassign *Transitions* to processors.

Both the GCL and the Pep were written in C++. In the following discussion we discuss the high level design of the Pep while striving to avoid any specific mention of the details of the language. Having said that, we include two interface specifications for the Pep in Appendices A and B.

Overview

First we say a few words about terminology, specifically about the notions of a *processor* and of a *process*. Whereas a *processor* is a physical machine (a CPU), a *process* is a thread of control. It is possible, therefore, for a single processor to execute several processes, e.g., by time-sharing. It is also possible to have each processor execute a unique process, so that no time-sharing of processes occurs on any one processor. In PGMT, we do the latter by dedicating each processor to a unique process. Moreover, we assume that there are no competing processes on any processor – i.e., that every processor is dedicated to executing the PGMT application. In the remainder of this paper, without confusion, we use the terms *processor* and *process* interchangeably.

1.1 Pep Requirements

The Processing Graph Method (PGM) Tool (PGMT) allows a user to define, compile and execute a PGM Graph on a parallel-computing platform. PGM Graphs are defined by the PGM-2.0 specification (1), and are made of Transitions and Places. The part of PGMT that governs the execution of a PGM Graph is called the PGM Execution Program, or Pep. Since PGMT executes the Graph on a parallel platform, Transition execution tasks must be assigned to specific processes and scheduled to execute in some specified order. The Pep, which is responsible for this, is required to do the following:

Create an initial assignment and schedule using a PGM2 Graph Specification, user provided performance parameters, a Machine Specification, and, if available, prior performance data.

Execute the Graph according to the schedule.

Analyze performance during run-time and store performance data for future runs.

Specify reassignments, when they occur, and new schedules during runtime.

Respond to a Graph suspension request by the Command Program.

We explain in this section how the Pep satisfies these requirements.

1.2 Concept of Operations

In the design of the Pep, we strived to provide local decision-making at the local level (within each processor) as much as possible and to resort to global decision-making only when a problem cannot be

solved locally. In this way, we reduce the need for communication between processors and thereby improve performance.

The PGM specification (REF) defines Command Programs. In PGM, the Command Program is distributed, consisting of multiple processes executing on multiple processes. Of these processes, at least one is devoted to Command Program functions throughout the execution of the Graph. We refer to these processes as *Command Program Processes*. Other processes are devoted to executing the graph; these are called *Graph Processes*. Command Program functions include things such as placing tokens on input ports and removing tokens from Graph output ports.

The Command Program creates the Graph and supplies input tokens to and removes output tokens from the Graph it has created. The Command Program constructs the Graph in each Command Program Process and in each Graph Process. As described in [2], the collection of Graph data structures that interact with the Pep represents a flattened version of the PGM Graph to be executed. When the Command Program constructs the Graph, the Graph is initialized with a null assignment, representing Graph suspension. Then in every Command Program Process and every Graph Process, the Command Program requests the Pep to create an assignment. Finally, just before graph execution begins, the Pep passes control of the Graph execution to the Pep in each Graph Process, which is called a *LocalPep*. The Command Program retains control of each Command Program Process. In the Command Program Processes, the Command Program supplies input tokens and removes output tokens from the Graph by interaction with the pseudo-transitions that represent the input and out graph ports.

To create the assignment, the Pep first identifies and assesses the prior performance of the Transitions on the processes provided for executing the Graph, and the communications loads. To identify the prior performance, the Pep reads from previously stored data files, if they exist, that record prior values of performance parameters. Using the performance assessment, the Pep creates an assignment of Transitions to LocalPep processes and a schedule for each LocalPep.

The assignment specifies a mapping of Transitions to processes for the execution of the Transitions. An assignment determines where each transition will execute until it is reassigned. To execute the Graph, the Command Program identifies a subset of its processes to be placed under Pep control, and in so doing, commands the Pep to start graph execution. In PGM, one of the processes controlled by the Pep is devoted to the UeberPep, while the other processes (the Graph Processes) are used by the LocalPeps to execute Transitions. The *UeberPep* is the master Pep object and controls the LocalPeps. On each process used to execute Transitions there is exactly one LocalPep. Each LocalPep oversees the execution of Transitions on that process. For performance characterization and control, we assume that there are no competing computational activities on the processes with LocalPeps.

The Transitions and Places in the Graph pass data tokens as a direct consequence of Transition execution, without interference from the Pep.

The Pep tracks Graph performance during Graph execution. The Pep collects execution time data as individual Transitions execute. Additionally, token sizes, Pack consume amounts and Unpack produce amounts are logged. (Recall that all Ordinary Transitions consume and produce exactly one token at the Place connected to each input port and to each output port, respectively. The Special Transitions Pack and Unpack are exceptions: Pack may consume any number of tokens, and Unpack may produce any number of tokens.) The performance parameters are periodically written to files. If while tracking the Graph performance, the performance becomes unacceptable, then the Pep computes a reassignment to better execute the Graph.

Upon receipt of a suspension signal from the Command Program, the Pep terminates execution of the Graph. If commanded to restart a suspended Graph, the Pep resumes execution of the Graph with the assignment and data stored in Places that existed at the time of suspension.

There are several things that the Pep does not do. The software data structures and code that actually construct the Graph, execute the Transitions, pass Tokens, and perform the reassignment of Transitions as specified by the Pep, are collectively called the PGMT Graph Class Library (GCL). The Pep has a large interface with the GCL, specified in Appendix B. The Pep also has a smaller interface with the Command Program, specified in Appendix A. Finally, the Pep uses another library, called the PGMT Middleware, which provides the calls for message passing, multiprocessor clock synchronization, and hardware system performance characterization. The details of the interface between the Pep and the Middleware are not given in this document.

1.3 Scheduling via Assignment, Priorities, and Capacities

The Pep schedules PGM Graphs by assigning Transitions to processes and assigning priorities to Transitions. In addition, capacities are specified for queues. A Transition is *ready to execute* or simply *ready* if both of the following conditions are met:

- (1) All of its upstream places have sufficiently many tokens for its execution, and
- (2) All of its downstream places have sufficient capacity to accept the tokens to be produced.

In a given process, among the ready Transitions that are assigned to that process, the one with the highest priority is executed next.

The Pep's scheduling of PGM Graphs is also dynamic. The Pep continuously monitors the performance of an executing Graph and evaluates whether re-scheduling is necessary. When the Pep schedules or re-schedules the Graph, it first takes a snapshot of the Graph's latest performance and uses that snapshot to

determine a good assignment of Transitions to processes and of priorities to Transitions. The Pep represents this snapshot as a PGM Task Graph with weighted nodes and edges representing communication times and Transition execution times. The Pep then proceeds to schedule using this PGM Task Graph.

The determination of whether or not a schedule is a good one depends on the objective function, where the best schedule optimizes the objective function. The objective function in PGM is defined as follows: The Pep seeks to find the best throughput schedule that meets a specified latency constraint. If the Pep cannot find such a schedule, it seeks to find the minimum latency schedule. The Pep compares schedules based upon the following compound rule.

Optimization Rule: If two schedules have latencies that do not meet the latency constraint, the schedule with less latency is better. If two schedules have latencies that do meet the latency constraint, the schedule with greater throughput is better. Finally, if one schedule meets the latency constraint and the other does not, the schedule that meets the constraint is better.

The Pep computes latency as the schedule length of the scheduled PGM Task Graph. To find the maximum throughput, The Pep minimizes the maximum load over all loaded resources. The Pep computes the loads for each resource in units of time. Resources include the processes, as computation resources, and the I/O ports and the network, as communication resources. We model these resources because any one can become a bottleneck.

Scheduling Details

As stated above, we make use of three basic scheduling constructs for executing PGM Graphs: Transition assignment, Queue capacities, and Transition priorities. In general, we compute an assignment of Transitions to processes and Transition execution priorities to balance computational and communication loads while meeting a specified latency. Capacities on the Queues are set sufficiently high so as to minimize the likelihood of the Graph becoming deadlocked and sufficiently low so that latency does not exceed the specified latency. Below, we discuss in detail how we do this, starting with the assignment algorithm. In our discussion, we detail various graph manipulations. None of these affects the structure of the PGM Graph; their only effect is to arrive at an assignment and a set of priorities. Before beginning our discussion, we make a distinction between *PGM Graphs*, or *Graphs* (capitalized), and general *graphs*. By the lower case term *graph* we mean those objects discussed in the branch of mathematics called *graph theory*. When talking about Graphs and graphs we must remember that a Graph is a graph, but a graph might not be a Graph.

1.4 Deriving a PGM Task Graph from the PGM Graph

To compute the assignment of the Transitions to processes the Pep first creates a PGM Task Graph corresponding to the activity of the PGM Graph. The Pep then uses a genetic algorithm to determine a good assignment of the PGM Task Graph to the multiple processes. The resulting assignment is used to assign the actual Transitions. This is possible because there is a one-to-one mapping between the tasks in the PGM Task Graph and the Transitions in the PGM Graph. We now describe how to construct this PGM Task Graph.

A PGM Graph may have feedback loops. To create a PGM Task Graph from the PGM Graph, we must remove these loops. First, a PGM DAG (Directed Acyclic Graph) is computed from the PGM Graph. In the following algorithm to construct the DAG, we use the following terminology:

The *Transition Dependency Relation* is the transitive closure of the set of Transition pairs (A, B) where B is downstream from A, i.e., where there is a Place in the PGM Graph whose upstream Transition is A and whose downstream Transition is B.

To *Remove* a Place P, we replace P by two aliases P' and P''. P' has the same number of output ports as P, these output ports being connected to the same Transition input ports as the output ports of P. Similarly, P'' has the same number of input ports as P, these input ports being connected to the same Transition output ports as the input ports of P. We also create a new in-port and a new out-port of the Graph. The input to P' is then connected to the new in-port of the Graph and the output of P'' is connected to the new out-port of the Graph. Producing a token to the alias P'' results in a token being placed at the new in-port, i.e., at P'. Thus a cycle is removed. If there is no GVar, multi-tailed Queue, or initialized Queue to be found in a dependency cycle, then the PGM Graph may deadlock.

We remove loops via the following algorithm:

Compute Transition Dependency Relation from the Processing Graph

Compute the sub-graph of Transitions that depend on themselves

While the sub-graph is non-empty

 Consider the set \mathcal{P} of Places P with upstream Transition A and downstream Transition B such that A and B are in the sub-graph. Choose P in \mathcal{P} such that P is initialized. If no initialized P exists, choose a P in \mathcal{P} that is not initialized.

 Remove the Place P.

 Re-compute the Transition Dependency Relation for the resulting Graph

 Re-compute the sub-graph of Transitions that depend on themselves

End While

Once the PGM DAG is created, the loads need to be computed to create the PGM Task Graph. We compute the loads in two steps: first we compute relative load factors, and then we compute the actual loads. The relative load factors capture the fact that the execution of a PGM Graph may result, for example, in Transition A executing n times for each execution of Transition B. In such a case we would say that Transition A has a load factor of n relative to Transition B. The loads for the tasks in the PGM Task Graph, which vary by processor, will be the load factor multiplied by the amount of time the corresponding Transition takes to execute on a specified processor. The actual time a Transition takes to execute on a specified processor is taken from timing prior transition executions. Similarly, if two Transitions are assigned to different processors, a message needs to be passed between the two processors. The load is computed using the load factor for the corresponding place, the corresponding token size and the data transmission rate for the connection between the two processors. The data transmission rate for the connection between the two processors is measured before the PGM Graph is run.

Average throughput for an assignment is computed using the loading created by the PGM Transitions, as well as the loading created by message passing. Throughput is estimated as the inverse load for the maximally loaded resource. The processing load imposed by an executing Graph is computed based upon the average amount of work required to produce a token at a specified output port. All of the resource loads are computed in units of time.

In the following discussion, it is important to remember, as described in [2], that each Place is automatically assigned to the same process as its downstream Transition(s). Thus, an assignment of Transitions to processes is sufficient to define where all Nodes are assigned.

We define $\mathbf{T}_{i,j}$ as the time required for Transition \mathbf{t}_i to execute on processor \mathbf{p}_j . We also define the assignment, \mathbf{A} , is represented as a Boolean matrix such that $\mathbf{A}_{i,j}=1$ means that Transition \mathbf{t}_i is assigned to execute on processor \mathbf{p}_j . A second Boolean assignment matrix, $\mathbf{\alpha}$, is computed for the Places, such that $\mathbf{\alpha}_{i,j}=1$ means that Place $\mathbf{\pi}_i$ is assigned to processor \mathbf{p}_j .

The load factor, λ_i , for the Transition \mathbf{t}_i is the mean number of executions of the Transition \mathbf{t}_i required to produce one token at the specified output. $\Lambda_{i,j}$ is the load created by Transition \mathbf{t}_i when assigned to execute on processor \mathbf{p}_j . It is computed as

$$\Lambda_{i,j} = \lambda_i * \mathbf{T}_{i,j}. \text{ The total processor load, } \mathbf{PL}_j \text{ on processor } \mathbf{p}_j, \text{ is } \sum_i \Lambda_{i,j} * \mathbf{A}_{i,j}.$$

Resource loading is also computed for communications loading on individual processor I/O ports and total communications loading of the network. The I/O load on a given processor is determined by adding the average time spent sending and receiving data at a given processor I/O port towards the purpose of

producing one token at the specified output. The average time spent sending and receiving messages for a given Place, π_i , at a processor's I/O port is found by the multiplying the average token size for that Place, β_i by the appropriate load factor, λ_i , and dividing by the port's rate capacity, c_j of a given processor, p_j , i.e., $\lambda_i * \beta_i / c_j = \tau_{i,j}$.

Load factors may be computed as follows. Call a port of a Transition ordinary if exactly one token is produced or consumed at that port for one execution of the Transition. Call a queue ordinary if it has only one input and one output. The load factors, λ_i , are computed node by node, by beginning at the designated out-port of the PGM Graph. At the (ordinary) Place connected to the designated out-port the load factor is one. For ordinary nodes connected to ordinary nodes, the load factors are the same. For a special Transition connected to an ordinary node via an ordinary port, the load factors are the same. The load factor of a node connected to the input port of a Pack Transition is the product of the load factor of the Pack Transition and the (mean) consume amount of the Pack. The load factor of an Unpack Transition is the load factor of the node connected to the Unpack Transition's output port divided by the (mean) produce amount of the Unpack. There is no load factor relationship between any node and a Graph Variable.

In order to more easily compute the loads we first decompose the PGM DAG into components we call Lumps. First, if the PGM DAG has multiple connected components, each of these is identified and called a SuperLump. By *connected component* we mean a sub-graph of the PGMDAG whose nodes may all be reached from each other by following edges (in either direction), and where that sub-graph contains all nodes of the PGM DAG that may be reached from any node within the sub-graph. We then further decompose the SuperLumps into Synchronized Lumps. The idea behind synchronized Lumps is that the part of the PGM Graph they represent has Transition-like properties: if a single token is placed at each input of the sub-Graph, execution to completion of that sub-Graph results in a single token arriving at each output.

The Synchronized Lumps of a SuperLump are identified by removing the arcs of the PGM DAG from the following locations: where multiple arcs connect to the input ports of a place; where multiple arcs connect to the output ports of a place; where the arc connected to the input port of a Pack; and where the arc connected to the output port of an Unpack. The resulting connected components are each a Synchronized Lump. A Graph Variable is considered to belong to a lump only if all its inputs and outputs are connected to ports of Transitions of a single lump. The load factor for each node in a Synchronized Lump is the same because, in the long term, we expect each node will execute the same number of times.

The Transition execution times, $\mathbf{T}_{i,j}$, the Unpack produce amounts, the Pack consume amounts, and the message sizes, β_i , are treated as random variables and must all be sampled and filtered. The latency of the running Graph must be periodically re-computed to make sure it has not exceeded the specified threshold. The load factors, λ_i , and the system throughput must also be periodically re-computed. Random variables are sampled and their statistics maintained in the following manner. A random variable, X , is an integer. A sample is maintained of the last N values of X . For each new value of X , a sum, Σ , of the N values of X in the sample will be computed and the sample mean is computed as Σ/N and stored. The value of N may be a computed number dependent on the values of the sampled variable. The default value of N is ten.

With the weights computed, the PGM DAG is a weighted DAG and may be interpreted as a task graph.

1.5 Scheduling the Task Graph

The scheduler schedules the PGM DAG as if it were a task graph. The assignment of PGM Graph Transitions to processes and of priorities to PGM Graph Transitions is then computed from the PGM DAG schedule.

A genetic algorithm whose general outline is as follows computes the assignment:

Start with a PGM DAG as a task graph;

Generate N initial schedules of tasks on the multiple processes;

Loop (through heuristically determined number of iterations)

 Using the N existing schedules, generate P distinct schedules via “genetic” recombination and mutation;

 For each Schedule compute the throughput and latency of the PGM DAG;

 Order the P schedules from best to worst, i.e., first by throughput (greatest throughput first) for those schedules that satisfy the latency constraint, then followed by the remaining schedules ordered by latency (least latency first).

 Select the N best distinct schedules;

Repeat.

The genetic algorithm is discussed again in § 3.5, and a more complete description of the genetic algorithm is given in § 6 below.

1.6 Computing Latency

Latency is readily defined for task graphs. One measures the time interval on a task schedule between specified input and output events. With PGM graphs the existence of many events on a task schedule are data dependent. Also, input events may have an arbitrarily large number of output consequences or output events may have an arbitrarily large number of input precedents. Consequently, we need to define dependency between tokens and then latency in a PGM graph.

We first define a relation of dependency between tokens. In a given Graph G , we say that token B is *dependent on* token A , if there is a transition T in G such that T consumes A and produces B during the same execution. The transitive closure of this dependency constitutes the *complete token dependency relation*.

A latency relationship may be defined four different ways in terms of dependent tokens and their respective places. Latency may be defined:

- (1) Between two tokens A and B where B is dependent on A . The latency is the minimum absolute time interval between production of A and the production of B .
- (2) Between a token at a graph output (out-token) and a place connected to a graph input (in-place). The latency is the absolute interval between the time the out-token is produced and the time that the last token, upon which the out-token is dependent, is produced to the in-place.
- (3) Between a token at a graph input (in-token) and a place connected to a graph output (out-place). The latency is the absolute interval between the time the in-token is produced and the time that the first out-token dependent on the in token is produced to the out-place.
- (4) Between an in-place and an out-place. The latency between two places is the minimum over: latencies between the in-place and all out-tokens produced to the out-place, as defined in (2) above, combined with; latencies between the out-place and all in-tokens produced to the in-place, as defined in (3) above.

We take definition (4) as the most meaningful for PGM graphs.

1.7 Latency Computation for the Task Graph

The Command Program specifies a maximum latency for scheduling purposes. Latency is specified between a place connected to the PGM2 graph's in-port and a place connected to a PGM2 Graph's out-port, as defined above in definition (4). However, we do not compute the exact latency as defined, rather

we estimate it. Operationally, latency is computed in the following manner. A task graph, called the PGM-TGraph is computed from the PGM DAG. For each node on the PGM DAG, there is a single corresponding task on the PGM-TGraph. The task execution times are found from $\mathbf{T}_{i,j}$, defined above. The PGM-TGraph is scheduled on a parallel processing machine and the times to execute message-passing tasks are attributed to the corresponding places in the PGM-TGraph. Each Place not sending messages (i.e., because its upstream and downstream transitions are assigned to the same processor) is represented as a zero-time task. The latency is then computed from the schedule for the PGM-TGraph as the critical path length between task graph nodes corresponding to the user specified places in the latency specification. This computation of latency follows from latency definition (4) above.

The task graph represents a “snapshot” of the Graph’s latest execution performance. The execution performance is determined by logging the rate of execution of tasks and the times those tasks take to be executed. If the PGM Graph were made up of ordinary Transitions and queues, there would be no variation of the relative rates of Transition firings. We must account, however, for the variable rates of Token consumption by Packs and the variable rates of Token production by Unpack.

1.8 Genetic Algorithm

The basic scheduling algorithm is a genetic algorithm. We use a combination of random generation and heuristics based on critical path methods to generate an initial population of assignment/schedules. We use the genetic algorithm methods to select and generate new graphs. New generations are calculated until there is a lack of progress in finding improved assignment/schedules. At this point the genetic algorithm terminates and returns the "best" assignment/schedule in the latest generation (i.e., the one with the highest throughput meeting the latency constraint, or - if none exists - the one with the smallest latency). The assignment/schedule thus returned is a *list schedule*, which has the following structure:

A list schedule is a sequence of pairs, one pair for each Transition. The first of each pair is the Transition identifier, and the second of each pair is the process identifier to which it is assigned. The order of the pairs in the list schedule is used as described in § 3.6 below to calculate the priorities of the Transitions for execution.

For the purpose of deciding to reassign, a new assignment and priority schedule are computed and compared to the current assignment and priority schedule. Reassignment thresholds may be set so that reassignment occurs when either of two cases occurs. In the first case, the current latency violates the user-specified latency threshold and a new assignment can reduce latency. In the second case, a new assignment can achieve a desirable improvement in throughput, where the threshold is specified as a percentage improvement over the old assignment.

A detailed discussion of the Genetic Algorithm is given in § 6 below.

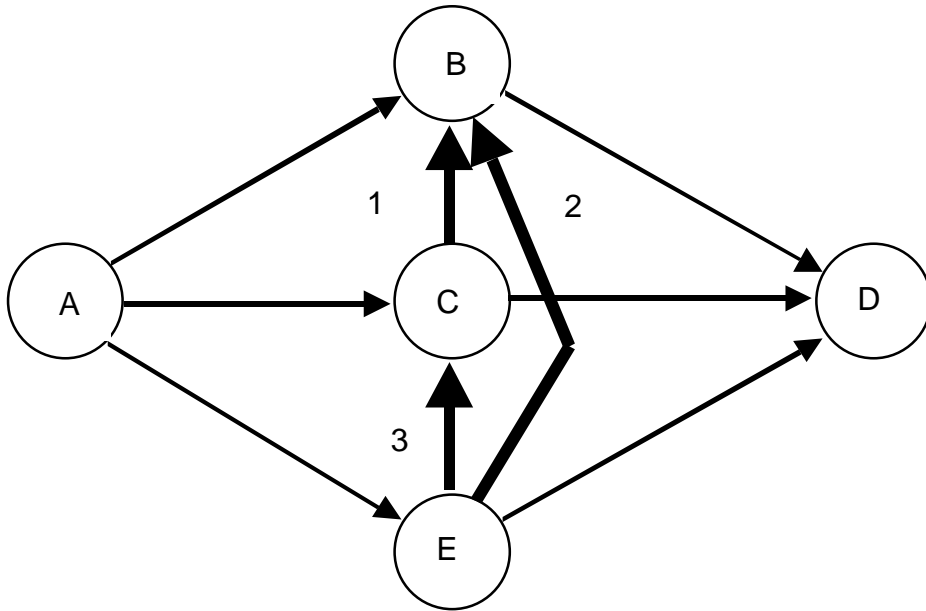
1.9 Computing Priorities

After obtaining a list schedule from the genetic algorithm, a set of priorities must be calculated. Both the PGM DAG and the list schedule are used to compute the priorities. The priorities are computed according to two basic rules. First, higher priorities are assigned to the Transitions producing Graph outputs. This is done to avoid excess latency. More generally, for any pair of Transitions, A and B, if B is downstream of A, then B has a higher priority. This generally results in only a partial order over the Transitions. The priorities constitute a total order over the Transitions. The second rule is to assign priorities such that for two Transitions on the same process where neither is upstream of the other in the PGM DAG, they will be executed in the order specified by the list schedule.

To illustrate, the following procedure computes the priorities from the PGM DAG and a list schedule. Assume the PGM DAG has N tasks, and a corresponding list schedule, L . The task graph has a *connectivity* relation, C , and a corresponding transitive closure of C , a partial order called the *dependency* relation, D .

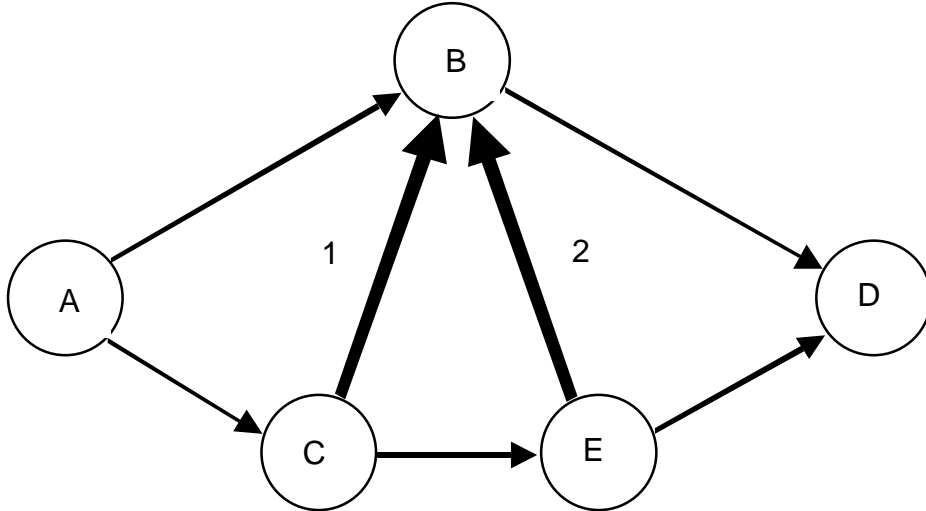
```
Relation R := C
Loop over i = 1; i <= N;
    Loop over j = 1; j <= N-i;
        if ( L(i), L(i+j) ) is in D then break;
        else add ( L(i+j), L(i) ) to R;
        End if
    j := j + 1
i := i + 1
Relation Priority := Transitive Closure(R);
End;
```

This procedure takes the relation represented by the edges of the DAG, C , and adds sufficient new edges to make R , such that the transitive closure of R is a total order, i.e., the priorities. In Figures 1 and 2 we show two examples of the application of this procedure. The added edges are bold and numbered in the order that they were added. The two examples are two different graphs having the same number of nodes and the same list schedule.



List Schedule: A, B, C, E, D
 Priorities (high to low): D, B, C, E, A

Figure 1



List Schedule: A, B, C, E, D
 Priorities (high to low): D, B, E, C, A

Figure 2

Note: The list schedule is a total order that, if reversed, extends the partial order of the task graph. One might wonder, therefore, why not just reverse the list schedule and use that to assign the priorities. The

answer lies in the difference between how two Transitions relative priorities are established, depending on whether one depends on the other. Suppose that v and w are two Transitions and that v precedes w in the list schedule. There are two cases:

- (1) w is dependent on v in the task graph. The higher priority goes to w in order to reduce latency, as explained in the first paragraph of § 3.6 above.
- (2) w is not dependent on v . The higher priority goes to v by virtue of its earlier appearance in the list schedule.

Simply reversing the order of the list schedule would satisfy case 1 but would violate case 2. The algorithm above satisfies both rules.

1.10 Capacity Assignment Policy

The principal reason for setting capacities is to constrain memory usage in the Graph and to limit the Graph's latency and variance in latency. To prevent deadlock, capacities must be set to at least the threshold amounts of the Transitions that read from them, or the produce amounts of Transitions that write to them. Setting Place capacities to be greater than the threshold amounts of Transitions that read them or the produce amounts of Transitions that write to them may allow data buildup internal to the graph. Whenever data builds up there is a consequential temporary increase in latency. With limited exceptions, therefore, we expect most capacities to be set at or slightly greater than the minimum value required to prevent deadlock. (Some exceptions may include, for example, higher (unbounded) capacities on queues connected to graph ports). It should be noted that capacities are a constraint, and perhaps a redundant one. We expect that the desired behavior with respect to latency will be achieved primarily through the Transition priorities.

The policy for controlling capacity is as follows.

The Command Program shall have no write access to capacities. Access may be allowed to read the queue content for queues connected to Graph ports. The Command Program may override the capacity constraints when writing tokens to the Graph.

The default value for capacity initialization is 1 (one). After initialization, the Pep may change capacities. The queue objects shall provide methods for the Pep to both read and write queue capacity values. The capacity value shall be an integer from zero to MAXINT (i.e., the maximum machine value of an integer). The capacity may be set to be less than the current content of a queue without error.

If the input Place to a Pack has insufficient capacity for the Pack to execute, the Pack object shall notify the LocalPep. Currently, *sufficient capacity* is defined to be $2 * T - 1$, where T is the *threshold*,

i.e., the number of input tokens needed for the Pack Transition to be ready. When so notified, the Pep adjusts the capacity of the input Place to $2*T-1$.

If the output Place of an unpack has insufficient capacity for the Unpack to execute, then the Unpack object shall notify the LocalPep. Currently, *sufficient capacity* is defined to be $2*P-1$, where P is the *produce amount*, i.e., the number of output tokens that the Pack Transition will produce. When so notified, the Pep adjusts the capacity of the input Place to $2*P-1$.

The Pep shall provide an interface to the Packs and Unpacks for alerting the Pep to the insufficient capacity conditions described above.

If the Pep changes the capacity of Places to zero, then the Pep is responsible for any resulting deadlock.

Note: The reason for the capacity adjustment to $2*T-1$ or $2*P-1$ is that in the case where a Pack Transition is downstream from an Unpack Transition, the larger of the two values is the smallest capacity of the intermediate Place that will ensure continued execution without deadlock.

Monitoring Graph performance

1.11 Tracking Performance

In order to decide when reassignment is desirable the Pep tracks the performance of the Graph. Each time a Transition executes, the execution time is logged. Each time a message is sent, its size is logged. Pack consume amounts and Unpack produce amounts are also tracked. A current estimate of the execution times of Transitions and the sizes of Tokens is maintained from this data. The current estimates of these performance parameter values are exponentially weighted averages of all prior values with emphasis on the most recent values. We use the following recursive filter to maintain these estimates:

For a given sequence of samples X_n and integer damping period $d > 1$, the estimates of the mean $x(n)$, and of the variance $v(n)$ are defined by:

If $1 \leq n < d$, set $\alpha = 1./n$; otherwise ($n > d$) set $\alpha = 1.0/d$. Then

$$x(n) = x(n-1) + \alpha * (X_n - x(n-1))$$

$$s(n) = s(n-1) + \alpha * (X_n * X_n - s(n-1))$$

$$v(n) = s(n) - x(n) * x(n)$$

We call the variable α the *damping factor* or the *filter coefficient*. The tracking filters for each tracked value are initialized with values stored from previous executions of the graph or by using default values. The default values are: 1.e-3 nanoseconds for transition timings; 1 byte for token sizes; 1 for Pack consume amounts; 1 for Unpack produce amounts; and 999 for nodes whose execution frequency is monitored. When reassignment is considered, or when performance data is stored to file, the current estimated, or filtered, values are used.

The variance is non-zero if the average of the square of the tracked variable is initialized to be somewhat greater than the square of the initial value for the average of the tracked variable. After initialization, any variation in the tracked variable guarantees the variance will be non-zero. So the average of the square of the tracked variable is initialized as $s(0) = \gamma * x(0) * x(0)$, $\gamma > 1$, where we prefer a value $\gamma = 1.1$.

Note: Currently, the variance is NOT initialized to be greater than zero, resulting in failure upon the first call. This can be fixed by initializing the square of the tracked variable to be greater than the square of the initial value for the mean of the tracked variable.

1.12 Storing Performance Information Between Graph Executions

Between executions of graphs the Pep needs to store performance information so that the Pep does not have to analyze performance from the ground up each time a graph is executed. Note that this performance is dependent on the physical processor, and so we are concerned about identifying the processor that is executing the transition.

For example, the following information must be stored:

(*< processor name >*, ProcessorDescription)

(*< processor name >*, *< processor name >*, ConnectionDescription)

(*< transition name >*, TransitionDescription)

(*< transition name >*, *< processor name >*, TransitionPerformance)

The Pep determines what values are stored for ProcessorDescription, ConnectionDescription, TransitionDescription, and TransitionPerformance, This is information private to the Pep, although it may also support future requirements to display performance information to the user.

The Pep needs to be able to know when processors and transitions have been encountered before, assuming that they can be expected to perform similarly to how they did in the past. We require unique

identifiers for processors, *processor name*, and for transitions, *transition name*. The identifiers here defined are to be persistent between run-times.

1.13 Initiating Reassignment

The Pep consists of UeberPep object and several LocalPep objects. They may each exist in one of four states, SUSPENDED, NORMAL, ALARMED, and REASSIGNING.

The Pep, meaning all UeberPep and LocalPep objects, begins in the SUSPENDED state. It receives a suspended Graph as an argument to a function to start the Graph. Each Pep (UeberPep or LocalPep) moves to the NORMAL state by this action. In the NORMAL state each LocalPep executes the graph and monitors performance.

Upon each execution of a Transition the execution time is processed, as described above, to provide a current statistical estimate of the mean and variance of the execution time for that Transition. A function is called to test whether either the Transition execution time or the sizes of generated tokens vary significantly from their hypothetical values, and whether the latency constraint is violated. This function returns *true* if and only if $\text{abs}(\text{mean} - \text{hypoth}) < \text{threshold} * \text{sigma}$, where *threshold* is currently set to 2. If the condition is false, the hypothetical execution time *hypoth* is reset to the current mean execution time *mean*, and the LocalPep further considers reassignment, as we describe below.

Figures 3 and 4 show the state diagrams for each LocalPep and for the UeberPep, respectively.

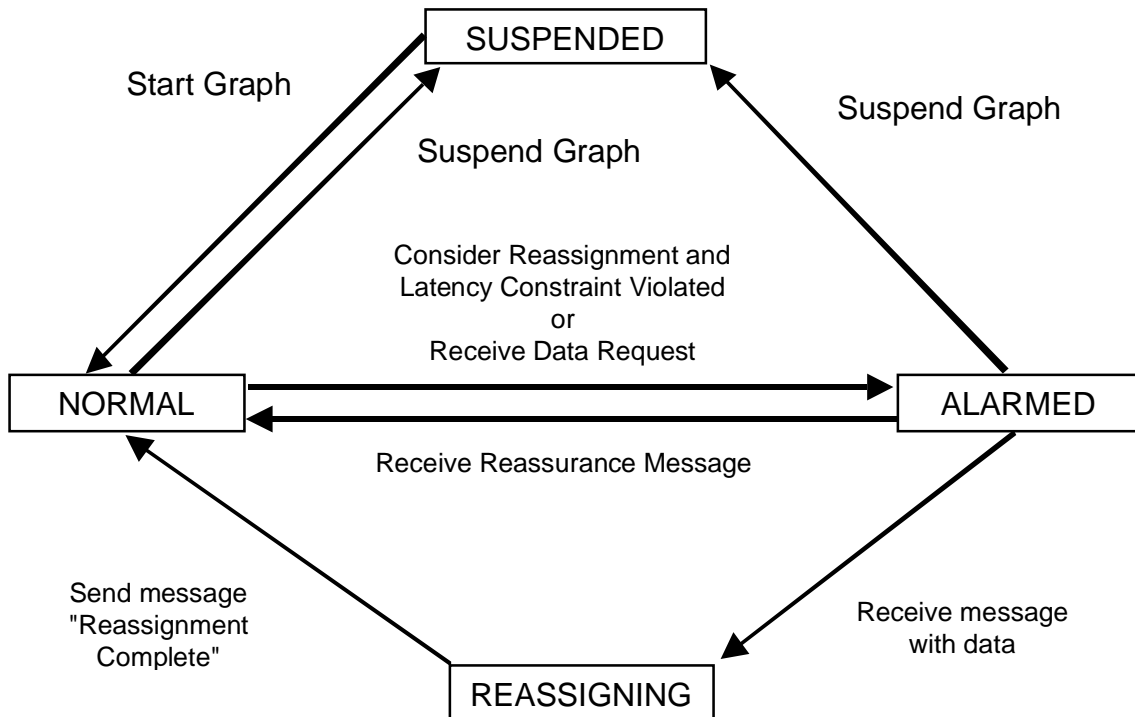


Figure 3: State Diagram for LocalPep

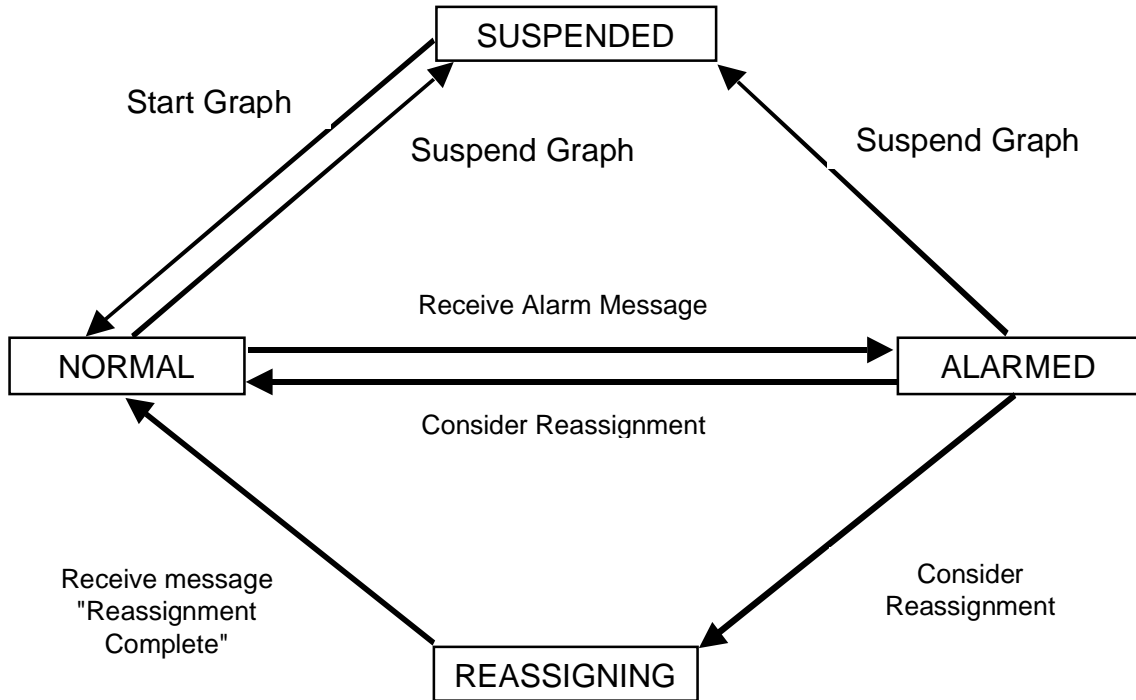


Figure 4: State Diagram for UeberPep

Note: In the previous paragraph, the function is described to return *true* if and only if $\text{abs}(\text{mean}-\text{hypo}) < \text{threshold} * \text{sigma}$, according to its original intent and design. However it is currently coded always to return *true* in order to eliminate performance variability during testing and debugging. Owing to insufficient time and resources the function we never coded the function according to its original design. The result is that every time the function is called (i.e., following every execution of a Transition), further reassignment is considered. Although this should be fixed, in our judgment the consequences are not serious, as we point out below following our discussion about further consideration of reassignment.

To further consider reassignment, in any state other than NORMAL, a LocalPep does nothing. In the NORMAL state, the LocalPep determines whether a significant performance change has occurred. To do this, the LocalPep recalculates the Latency and checks whether it violates the Latency constraint. This calculation is performed as described in § 3.4 above, and is based on the latest locally known load values.

If the Latency constraint was violated, the LocalPep then moves to the ALARMED state and sends an Alarm message to the UeberPep. When the UeberPep, in a NORMAL state, receives an alarm message, it moves to an ALARMED state. The UeberPep then sends a performance DataRequest message to every LocalPep. When a LocalPep receives a DataRequest message, it moves to an ALARMED state if it is in a NORMAL state. If the LocalPep is already in an ALARMED state it stays in an ALARMED state. In either case the LocalPep then sends the requested data to the UeberPep.

After receiving the new performance data from every LocalPep, the UeberPep considers reassignment in the same way that the LocalPep does, as described above. The UeberPep bases this calculation on the performance data received and thus has data that is more recent. If it decides that reassignment is unnecessary, it moves back to a NORMAL state and sends Reassurance messages to every LocalPep. Additionally, the UeberPep sends the merged, updated performance data to every LocalPep. Upon receipt of a Reassurance message each LocalPep transitions to the NORMAL state as well.

In the case that the UeberPep decides that reassignment is necessary (i.e., that the Latency constraint is not being met), it transitions to a REASSIGNING state, computes a reassignment and sends the reassignment data to every LocalPep along with updated and merged performance data. When each LocalPep receives this reassignment data it also moves to the REASSIGNING state. Each LocalPep call the reassignment function of the GCL to initiate reassignment. When the GCL finishes its reassignment tasks on a given process it notifies the LocalPep, which then moves to the NORMAL state and sends a ReassignmentComplete message to the UeberPep. When the UeberPep has received the ReassignmentComplete message from every LocalPep, it too transitions to the NORMAL state.

The UeberPep should only receive Performance Data Messages from a LocalPep while it is in the Alarmed State; it should only receive reassignment complete messages while it is in the REASSIGNING state. The UeberPep may receive Alarm messages, however, while it is in the NORMAL, ALARMED, or REASSIGNING states. We have already detailed the UeberPep's response to an ALARM message received while it is in the NORMAL state. When the UeberPep receives an Alarm message while it is in the ALARMED state, it does nothing. (In this case it has already sent a data request message to the same LocalPep). When the UeberPep receives an Alarm message while in the REASSIGNING state it responds with a Reassurance message. This situation can occur after a reassignment when a LocalPep individually returns to the NORMAL state before the UeberPep does, so a new round of Reassignment cannot begin until the last has been completed.

When a LocalPep returns to the NORMAL state from either the ALARMED or REASSIGNING states, it uses the merged, updated performance data to reset the tracking filters.

Note: To follow up on the earlier note about considering reassignment after every Transition execution, it turns out that the Genetic Algorithm, described in § 6 below, takes on the order of minutes to

determine a new assignment, while a reassignment is actually accomplished in a matter of a few seconds (i.e., some 2 orders of magnitude less time). Thus, it turns out that each Pep (LocalPep and UeberPep) spends most of its time in the ALARMED state. During this time each LocalPep conducts normal graph execution as described in § 5 below, and the UeberPep is running the Genetic Algorithm on its own dedicated processor, thus not interfering with the performance of Graph execution. If the Genetic Algorithm fails to find a reassignment that improves performance over the current assignment, then no reassignment occurs. Even so, the Pep will keep trying. But we conclude that there is little or no performance loss by this means.

1.14 User Control of Pep States

The Command Program may call the Pep function to Stop Graph. The effect is to put the currently executing Graph into the SUSPENDED state. To do this, it is sufficient for the Command Program to call this function in one of the Command Program Processors. The LocalPep responds by sending a message with a Suspend Graph Directive to the UeberPep. If in the NORMAL or ALARMED state, the UeberPep performs the following actions:

- Sends a message with a Suspend Graph directive to each of the LocalPeps.
- Switches to the SUSPENDED state.
- Returns from the function call to Start Graph. This returns control of the processor to the Command Program.

If in the REASSIGNING STATE, the UeberPep waits until Reassignment is complete and switches to the NORMAL state before proceeding as described above.

If in the NORMAL or ALARMED state, upon receipt of a message with a Suspend Graph directive, each LocalPep performs the following actions:

- Calls the Graph Library function to suspend graph execution.
- Switches to the SUSPENDED state.
- Returns from the function call to Start Graph. This returns control of the processor to the Command Program.

If in the REASSIGNING state, each LocalPep waits until Reassignment is complete and switches to the NORMAL state before proceeding as described above.

In the SUSPENDED state, to start graph execution, the Command Program calls the Pep function to Start Graph. The Command Program must make this call in every processor in which the graph was constructed. The results of this call depend on the individual processor.

- In each processor reserved for use by the Command Program, the function returns immediately, returning control to the Command Program.

- In each processor reserved for use by the Pep, the Pep does the following:
 - Switches to the NORMAL state.
 - Performs actions to support graph execution, either as the UeberPep or a Local Pep.
 - Retains control of the processor, returning control to the Command Program only when the Command Program calls the function to Stop Graph.

Section 4.3 describes Automatic Reassignment in the Pep. To support the ability of the user (author of the Command Program) to turn the automatic reassignment off and back on again, four additional functions are included the Command Program interface, described more fully in Appendix A:

- Stop Automatic Reassignment
- Start Automatic Reassignment
- Save Current Assignment
- Use a Previously Saved Assignment

When the Command Program calls one of these methods on a specific Command Program Processor, the Pep on that processor will send a message with a directive to the UeberPep and then return control immediately to the Command Program, not awaiting acknowledgement from the UeberPep. Upon receipt of each such directive, the UeberPep will queue it with any Suspend Graph directives. Each of these directives will be processed in the order received. Subsequent directives will not be processed until all the effects of the previous directive are final. In order to guarantee that the order of receipt is as intended, the Command Program should call these functions from the same process.

Note: This order of handling directives relies on the following guarantee by MPI about the order of receipt of messages in a given process. Suppose that in a given MPI communicator, two messages M1 and M2 are sent from process P1 to process P2. Then if M1 is sent from P1 before M2, then M1 will arrive in P2 before M2. No other guarantees about the order of message receipt are implied.

To support automatic reassignment the Pep has four states labeled ALARMED, NORMAL, REASSIGNING, and SUSPENDED, which are described in § 4.3 above. Three more states are used by the Pep to support these four methods for turning the automatic reassignment off and back on again. These states are: USER_CONTROLLED, SUSPENDED_UC, and USER_REASSIGNING. These three states form a branch of the state diagram entered from the NORMAL state. Figure 5 shows a diagram with these three states and the NORMAL state.

If the Command Program calls the function to stop Automatic Reassignment when the Pep is in the NORMAL state, the Pep responds by transitioning to the USER_CONTROLLED state from the NORMAL state. The UeberPep switches to USER_CONTROLLED from NORMAL by receiving the message AUTO_REASSIGN_OFF from a Pep on a Command Program process. This message may be received when the UeberPep is in any of the states NORMAL, ALARMED, or REASSIGNING. If

AUTO_REASSIGN_OFF is received when the UeberPep is in the ALARMED, or REASSIGNING states, the transition to USER_CONTROLLED occurs only after the UeberPep has returned to NORMAL. Receipt of the AUTO_REASSIGN_OFF message is idempotent, meaning that calling the function to Stop Automatic Reassignment twice in a row is the same as calling it once.

When the UeberPep transitions to the USER_CONTROLLED state it sends messages to the LocalPeps, causing them to transition to the USER_CONTROLLED state. It is possible that a LocalPep is in either a NORMAL or an ALARMED state when it receives this message. Other messages (REASSURANCE messages) from the UeberPep will cause the LocalPep to transition to the NORMAL state from where it can transition to the USER_CONTROLLED state.

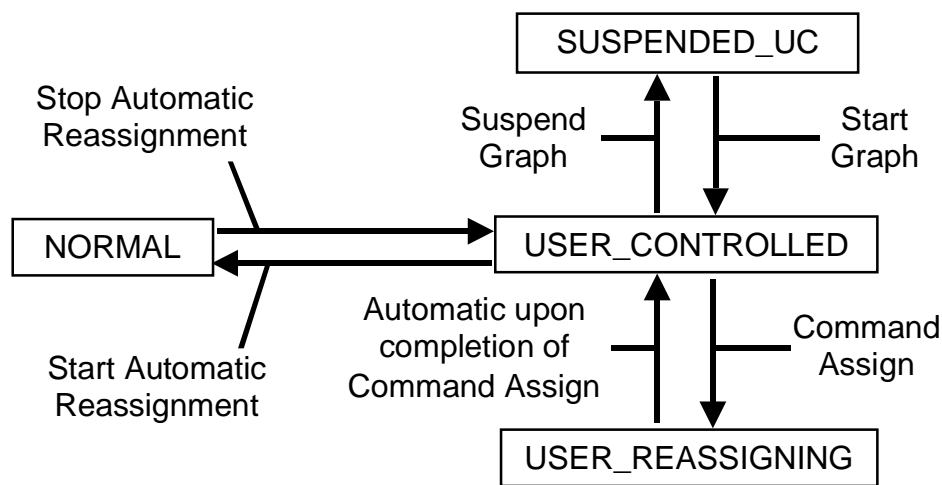


Figure 5: State Diagram for Pep under User Control

When the LocalPep is USER_CONTROLLED it changes to USER_REASSIGNING when it receives a message type USER_REASSIGNMENT from the UeberPep. When the LocalPep is finished reassigning with a user specified reassignment, it switches back automatically to USER_CONTROLLED.

The Pep switches from USER_CONTROLLED to NORMAL by receiving the message AUTO_REASSIGN_ON from a Pep object on a Command Program process. This message may be received when the Pep is in either the USER_CONTROLLED or USER_REASSIGNING states. The Pep will not transition back to NORMAL until after it has transitioned to the USER_CONTROLLED state. Receipt of the message AUTO_REASSIGN_ON is idempotent, so calling the function to Start Automatic Reassignment twice is the same as calling it once. This behavior is guaranteed by using a boolean variable, which is switched to true or false upon receipt of the appropriate message.

When in the USER_CONTROLLED state, if the Command Program calls the function to Stop Graph, the Pep responds as described above, except that it switches to SUSPENDED_UC. As stated above,

each Pep (i.e., UeberPep and each LocalPep) returns from the function to Start Graph, thus returning control of the processor to the Command program.

When in the SUSPENDED_UC state, if the Command Program calls the function to Start Graph, the Pep performs the same actions as described above for this function, except that it switches to the USER_CONTROLLED state. As stated above, the Command Program must call this function in every processor in which the Graph was constructed.

Given this design, it should not be possible for the Pep get into an undefined state. It is possible, however, for unpredictable results to occur if more than one Command Program process calls these methods.

Executing the Graph

While much of the implementation of the Pep is devoted to scheduling and monitoring of the graph execution performance, the principal effect of the Pep is to actually execute Transitions and thereby execute the Graph. As stated before, the Command Program initiates Graph execution by calling the Pep function to start the Graph and passing a subset of its processes to be used to execute the Graph. During Graph construction, the initial assignment of Transitions to processes was determined, and so each LocalPep begins executing ready Transitions immediately upon this function call.

Each LocalPep goes through a recurring loop. On each pass through the loop the LocalPep checks if there is a Transition ready to be fired. If there is a ready Transition then the one with the highest priority is executed. After logging the amount of time taken to execute the Transition, the LocalPep checks to see if there are any messages from the UeberPep and also allows the Graph Class Library time to send and receive messages.

As a consequence of the execution of other Transitions, or because of token flows at graph in-ports and out-ports, Transitions that previously were not ready may become ready to execute. Each time a transition becomes ready to execute, the graph library notifies the Pep that the Transition is ready. In some cases a Transition may go from being ready to not ready by some means other than being executed. In particular, this can happen for Pack transitions. In such a case the graph library notifies the Pep that this has occurred.

If a Pack is unable to execute because the upstream queue capacity is too low, it notifies the Pep by calling the appropriate Pep function to alert the Pep of a possible deadlock situation. If an UnPack is unable to execute because the downstream queue capacity is too low, it notifies the Pep by calling the same Pep deadlock alert function. Calling this function causes the Pep calls the queue function to adjust the capacity of the queue causing the deadlock.

To execute a transition, the Pep determines the Transition that has the highest priority among those that are ready, removes that Transition from its ready queue, and initiates execution by calling the Transition function to execute. The timings of the transitions are determined by calling clock functions before and after Transition execution. The Transition executions are assumed to occur without interruption, i.e., asynchronous data transfers are assumed not to affect the timings. At the end of Transition execution, the Transition must check whether it is again ready. If so, then the Transition must again notify the Pep that it is ready.

Finding an Assignment or Reassignment – The Genetic Algorithm

This section describes how the genetic algorithm in the UeberPep works to determine the mapping by which Transitions should be assigned or reassigned to processors.

This program is a scheduler implemented as a genetic algorithm. It computes a schedule that minimizes the maximum load on system resources (i.e., processors and communication channels) subject to a latency constraint. If the algorithm is unable to satisfy the latency constraint, it will return a schedule that minimizes latency. This version of the scheduler works for heterogeneous processors, i.e., the case where a given task takes different amounts of time to execute depending on the processor.

The scheduler is given a task graph specifying the precedence constraints. The solution is guaranteed to satisfy these constraints. However, being a genetic algorithm, the scheduler provides no guarantees regarding optimality.

The algorithm first generates a population of chromosomes, each representing a schedule. This population forms the first generation. Crossover and mutation operators to existing populations are then applied to construct successive generations. For the solution, the algorithm returns the best chromosome in the last generation.

In trying to minimize the maximum load, the scheduler considers two different types of load:

- (1) The load on a processor, which is computed in one of two ways:
 - (a) The number of time units between the first and last use of the processor.
Note that this includes any idle time.
 - (b) The number of time units the processor is actually used.
- (2) The load on a communication channel.

The computation of communication load depends on the network model. In one model, the communication load on each processor is computed. In the other, we are only interested in the communication load on the entire system, i.e., the sum of the communication loads on all processors.

Note that in the latter, communication is much more of a bottleneck. The maximum load on the system is the greater of the maximum processor load and the communication load.

In the following, we discuss the different components of the program in more detail. These should be studied in conjunction with the inline documentation as well as the code itself.

1.15 The Task Graph

The precedence constraints, execution times and communication costs are given to the scheduler in the form of a task graph. The construction of the task graph is described above in § 3.1. A task graph is essentially a collection of vertices (or nodes) connected by a number of directed edges (or arcs).

Each vertex represents a task to be scheduled on a processor. With each vertex is associated a Node Cost specifying the time required to execute the task represented by the vertex on each processor. An edge from vertex v to vertex w specifies that task v must end before task w starts. This constraint is present if, for example, task w requires the output produced by task v .

With each edge (v, w) is associated an Edge Cost specifying the time to transfer the output of task v from v 's processor to w 's processor. This quantity is calculated as the product of (1) the number of information units owing from task v to task w and (2) the time to transfer a single unit of data from v 's processor to w 's processor. The Edge Cost can be used to determine the cost of communication between tasks v and w for any pair of processors p_1 and p_2 on which v and w might be scheduled. The cost of communicating one unit of information between different pairs of processors is accomplished before graph construction and is stored in a global array.

Note: Suppose that v and w are two nodes that appear consecutively in a schedule derived from a task graph (i.e., v and w are in the i^{th} and $i+1^{\text{st}}$ pairs of the schedule). Then in the task graph, either there is an arc directly from v to w , or there is no directed path from v to w at all. To see this, suppose that there is a path in the task graph from v to w with an intervening node u . Then u must appear in the schedule between v and w . This violates the assumption that v and w are consecutive. This observation permits a rapid check for dependence between two consecutive nodes in the schedule, thus reducing execution time in various calculations involving schedules and the task graph.

1.16 The Chromosome

A chromosome is essentially a schedule that satisfies the precedence constraints specified by the task graph passed to its constructor. The schedule is stored in the chromosome as a vector of pairs. The presence of pair (t, p) in the sequence specifies that task t is assigned to processor p .

1.16.1 Calculating Task Starting Times

The starting times of the tasks are not explicitly stored in the schedule. Since this information is required for computing functions such as latency, it must be computed. For the first pair, this is time 0.

We can determine the time when a given task t in the task graph is runnable. Consider the set $Pr(t)$ of predecessors of t . We know that t is runnable when all of the tasks in $Pr(t)$ have completed execution and all of the output from those tasks have arrived in t 's processor. For a task u in $Pr(t)$, if u is in the same processor as t , the output of u is immediately available. If u is in a different processor from t , then the Edge Cost determines when the output from u is available in t 's processor. The task t is runnable when all of the output from the tasks in $Pr(t)$ are available in t 's processor.

Once we have determined the starting time of task t on processor p , we can compute when p is next available. This is easily determined, since the Node Cost associated with vertex t in the task graph gives the execution time of t in p .

Since the schedule stored in the chromosome satisfies the precedence constraints, as we get to a pair (t,p) in the schedule, we already know when t is runnable (since the execution times of all of its immediate predecessors in the task graph are already known). We also know that p is available when the most recent task scheduled on it ends.

1.16.2 Generating Chromosomes

As stated above, a chromosome is a schedule. The genetic algorithm is based on the generation of new chromosomes and evaluating each using an objective function to obtain its *fitness*. A number of the best chromosomes thus generated are kept as *survivors*, and the remaining ones are discarded. This process is repeated a number of times, and the single best surviving chromosome is returned as the schedule to be used. We describe the following algorithms for generating new chromosomes.

1.16.2.1 Random Chromosome Generation

The first algorithm generates a "random" chromosome that satisfies the precedence constraints.

Parameters used are

- The precedents constraints of the task graph,
- The number of available processors,
- The processors that can execute each task, and
- The desired bound on the latency of the schedule.

Note that no attempt is made to satisfy the latency constraint. Instead, it is merely stored and later used in computing the fitness of the chromosome.

Note also that the chromosome generated is not entirely random. This algorithm uses a parameter to control the expected length of a sequence of tasks assigned to the same processor. By a *sequence of tasks*, we mean a sequence of consecutive vertices on the same path in the task graph. Currently this parameter is set to the ratio (number of tasks) / (number of processors). The justification for this parameter is that by assigning consecutive tasks to the same processor, communication costs are reduced.

Finally, note that it is possible that two or more different tasks v and w must be assigned to the same processor. To account for this an infinite Edge Cost is assigned for the pair (v, w) . For each pair (v, w) in the task graph with infinite Edge Cost, the algorithm assigns v and w to the same processor.

1.16.2.2 A Second Random Chromosome Generation

The second algorithm combines two chromosomes using one of two *crossovers*. This algorithm is identical to the first except it takes an additional argument: a schedule given as a vector of pairs. The chromosome is initialized to reflect this schedule. This constructor must be used with caution. It is the responsibility of the caller to ensure the schedule passed to the constructor is a legal one.

Each crossover operators combines two chromosomes, called the *parents*, to produce a third one, called the *child*. Recall that a chromosome is a schedule, which is a sequence of pairs, exactly one pair for each task.

The first crossover starts by selecting a number k at random, where $1 < k < n$, where n is the total number of tasks. The child inherits pairs indexed by $1, \dots, k$ in the sequence from the first parent in the same order that they appear in that parent. This leaves $n-k$ tasks that do not yet appear in the child. The remaining pairs in the child are inherited from the second parent. These are pairs (t,p) such that t does not appear in any pair inherited from the first parent. The relative order of these pairs in the child is identical to their relative order in the second parent.

The second crossover operator is identical to the first except in this case, two numbers i and j are chosen at random so that $0 < i < j < n$. The child inherits pairs indexed by $i+1, \dots, j$ from the first parent.

1.16.2.3 Mutation

Mutation derives a new chromosome from an existing one. There are two mutation operators.

The first mutation operator selects a task at random and reassigns it to a randomly chosen processor that can legally execute the task.

The second mutation operator swaps two consecutive pairs (selected at random) in the sequence denoting the schedule. It should be pointed out that the only pairs considered for this operation are those whose swap will not violate any precedence constraints. Recall the note at the end of § 6.1 above, where we observe that it is sufficient to check whether there is a directed arc from one to the other.

Note that swapping (t,p) and (t', p') has no immediate effect if $p \neq p'$, because the two tasks are assigned to different processors. This does not result in a different schedule (i.e., each task's starting time remains unchanged). However, the swap will produce a different chromosome that may later result in a different schedule following additional mutations and crossovers. On the other hand, if $p = p'$, then the tasks being swapped are assigned to the same processor, thus reversing the priorities of the two tasks, which leads immediately to a different schedule.

1.16.3 Fitness

Four different fitness functions have been implemented. They all return the negative of the maximum load implied by the chromosome if the latency does not exceed the desired bound passed to the

algorithm that constructs the chromosome. Otherwise, they return the negative of the latency. Returning the negative of the maximum load or latency supports the notion that a smaller maximum load or latency implies a better schedule. Latency is defined as the elapsed time between the start of the first task and the end of the last task.

To determine the maximum load, the load on each of the system's resources is determined first. The maximum load is the maximum of the loads on all the system's resources. A resource is either a processor or a communication channel. Processor load is computed in two ways, and communication load is computed independently in two ways. This leads to a total of four ways to compute load, which in turn results in four fitness functions.

The first way to compute a processor load is to compute the number of time units the processor is actually in use. The second way is to compute the difference between the times of first and last use; this will include any idle time as part of the load.

The first way to compute communication load is to compute the load on the entire system assuming there is one resource for all communication. The second way to compute communication load is to compute the load on each processor's communication channel and take the maximum of all loads on the communication channels.

In all four cases the maximum load on the system is the maximum of (1) the maximum processor load and (2) the communication load.

It is worth pointing out that the maximum load and latency are computed *at the same time* in each of the fitness functions for efficiency although separate computation of these quantities would have resulted in a cleaner design.

It should also be noted that the calculated latency is always greater than or equal to the calculated load. Thus, suppose the desired latency is L . Suppose also that two chromosomes $C1$ and $C2$ have fitness values $F1$ and $F2$, where $F1$ is the negative latency of $C1$ and $F2$ is the negative load of $C2$. It follows that $F1 > -L > F2$.

1.17 Population of Chromosomes

A population is a collection of chromosomes. The collection is stored in a list, which also holds each chromosome's fitness.

To calculate the next generation of chromosomes, a number of crossovers are performed on the population. Each crossover is performed on two randomly selected chromosomes. Currently the number of crossovers performed is one tenth of the population size. Each chromosome resulting from a crossover is added to the pool of candidates for the next generation along with the chromosomes in the current generation.

Two possible mutations of each chromosome resulting from a crossover are also added to the pool. For each such chromosome, the first mutation is generated with a fixed probability using the first mutation

operator. The second mutation is generated with a second fixed probability using the second mutation operator.

The chromosomes in the next generation are selected from the pool of candidates based on their fitness, i.e., the chromosomes with highest fitness are chosen to populate the next generation.

1.18 The Scheduler

The Scheduler operator is given a task graph, the number of processors, and a desired latency constraint. Several different versions of this operator have been implemented. Every version computes a schedule that satisfies the precedence constraints specified by the task graph and minimizes the maximum load on the system resources within the latency constraint. If unable to satisfy the latency constraint, each version will try to minimize latency. Each version first generates an initial population of chromosomes and then computes successive generations until a termination condition is satisfied. At that point, the best schedule is the fittest chromosome in the last generation.

The various versions of the scheduler differ in the number of chromosomes generated for the first generation, the population size, and the termination condition.

In the first version, a fixed number (the population size) of random chromosomes are generated to form the first generation. Another fixed number (preordained number of generations) of generations are computed and the fittest chromosome in the last generation is returned as the best schedule.

The second version is identical to the first except it does not compute a fixed number of generations. Instead, it keeps computing new generations until there is no noticeable improvement in the fitness of the best chromosome from one generation to the next.

The third version is identical to the first except that the number of chromosomes initially generated is a fixed multiple (initial population factor) of the population size. From these, a number (the population size) of the fittest chromosomes are selected to form the first generation. This will usually result in a first generation that is on the whole fitter than it would be otherwise since we generate more chromosomes than we need and only select the best to populate the first generation.

The fourth version is identical to the second except that the population size is set to a fixed multiple (the population size factor) of the size of the task graph. This will result in a population size proportional to the size of the task graph. They are needed to generate high quality schedules.

Finally, the fifth version combines the features of the third and fourth. The population size is the product of the population size factor and the size of the task graph. The number of chromosomes initially generated is the product of the initial population factor and the population size. From these, a number (the population size) of the fittest chromosomes are selected to form the first generation.

Each version of the scheduler returns a sequence of triples consisting of a task number, a processor number, and a priority. From this sequence a list schedule is generated, which is a sequence of pairs, each pair identifying a Transition and its assigned processor identifier. Triples containing tasks that do

not correspond to Transitions are not used in the list schedule. The priorities determine the order of the pairs in the list schedule.

One point deserves further explanation here. In this application, the task graph does not represent a set of tasks to be executed once. Rather they will be executed repeatedly. Given an edge (v,w) in the task graph, while it is true that the k th execution of task v must occur before the k th execution of task w , there is no implied order on the k th execution of task w and the $(k+1)$ st execution of task v . This is where priorities come in. If two tasks scheduled on the same processor are ready to execute, the one with the higher priority is executed first. By giving downstream tasks higher priorities, we ensure that if possible, every task is executed for the k th time before any task is executed for the $(k+1)$ st time. This will minimize the buildup of tokens in the interior of the task graph. But more importantly, this strategy tends to minimize latency since latency is essentially the time required for one execution of the task graph.

Future Work

In this section we discuss some of the shortcomings of the Pep in order to identify areas where we believe effort should be applied to improve the accuracy of performance monitoring and thus to improve performance.

1.19 Reassignment

Much of this paper is devoted to discussion of how the Pep monitors performance and how decisions are made whether and how to reassign transitions to processors. We note that in the current implementation of PGMT, it is possible to turn off reassignment and to use an initial assignment that is determined in advance. The result is *static* assignment, in which the initial assignment is used for as long as graph processing continues. We believe this to be a good and useful feature.

Assuming that reassignment is turned on, we have two comments about its implementation in PGMT. The first is in the area of monitoring performance, and the second has to do with the consequences of the time required for the Genetic Algorithm.

1.19.1 Monitoring Performance

The Pep monitors performance of Transitions, as described in § 4 above. However the Pep does not monitor performance of Communication.

In the development of the Pep, we discussed this problem at considerable length, and we concluded that with the use of MPI for passing messages between processors, there is no way to determine the time cost of communication. While it might be simple to record the time when a message is sent, it is not possible to record the arrival time, because MPI does not cause an interrupt in the application. The only way the application can determine the presence of an incoming message is by polling. Moreover, when an MPI

message is detected and read, MPI does not support the reporting of the time when it actually arrived in the MPI process.

To complicate matters in the PGM environment, we designed the Pep and the GCL so that Transition execution proceeds until its execution is complete, i.e., without interruption even to poll for incoming messages. Thus, one or more messages may arrive during the execution of a single Transition. If the Transition execution lasts for a relatively long time, then the potential delay in detecting and reading a message after its arrival would render meaningless any attempt at recording a message's arrival time.

While the Pep does not monitor communication performance, it does attempt to model communication performance, based on some testing before graph construction. This testing involves the sending of test messages between processors and recording the times. Messages of various sizes are sent in order to determine the following:

- The cost of sending an empty message.
- The marginal cost of sending a message as a function of its size.
- The difference in clock settings in the various processors.

Although this information is useful in determining a simple model of the communication between processors, there is no attempt to model contention for communication channels. Thus, if communication costs are small compared to the capacity of the network, the model may be accurate. However, as communication load becomes higher, the Pep has no accurate model by which to determine the load. As a result, the estimated load will probably be lower than the actual load.

1.19.2 Timeliness of Reassignment

Recalling the note at the end of § 4.3, we observe that the Genetic Algorithm, described in § 6 above, typically takes a number of minutes to arrive at a new schedule. This is potentially a very long time to continue processing while the previous assignment was judged to be inadequate to meet the desired latency constraint. Moreover, the processing environment may have changed in the mean time, implying that the reassignment may no longer be appropriate.

How to improve on this problem is not a trivial question. While genetic algorithms are, in general, time-consuming, they require polynomial time, thus making them attractive for large problems like the reassignment problem in PGM. Indeed, PGM Graphs with hundreds or even thousands of nodes are possible.

1.20 Pack Input Ports and Unpack Output Ports

In the construction of the PGM Task Graph in § 3.1 the discussion speaks of "the input port of a Pack," and "the output port of an Unpack." We note that a Pack Transition may have as many as five input

ports, only one of which is not ordinary. Similarly, an Unpack Transition may have as many as two output ports, only one of which is not ordinary. The calculation of Synchronous Lumps does not distinguish among these input ports, and thus removes some connections that might be left intact. The result is that in the current Pep, what might well be a single Synchronized Lump is separated into more than one. An improved algorithm for calculating the Synchronized Lumps should be devised that would make this distinction. We regard this problem to be of little significance compared with the ones we describe above in § 7.1.

References

1. *Processing Graph Method 2.1 Semantics*, by David J. Kaplan and Richard S. Stevens, Naval Research Laboratory, Washington, DC, April 1, 2001.
2. *Distributed Processing in PGM*, by Richard S. Stevens, Naval Research Laboratory, Washington, DC, August 13, 2002.
3. *Graph Construction In the Processing Graph Method Tool*, by Richard S. Stevens, Naval Research Laboratory, Washington, DC, September 9, 2002.

PGMT / Command Program Interface

The following are C++ declarations that serve to define interfaces between the Pep, Graph Library and the Command Program.

A call made by the Command Program to create a graph

```
#include <map>
#include <utility>
typedef vector<MW_ProcessID> CPRanks
typedef vector<MW_ProcessID> GraphRanks
typedef string graphPortFamilyName;
typedef pair<graphPortFamilyName, vector<int> > graphPortID;
typedef map<graphPortID, MW_ProcessID> GraphPortAssignment;

new GraphClassName ((CPRanks) cpRanks, (GraphRanks) gRanks,
    (GraphPortAssignment) portAssignment, double maxLatency = 1.e18);
```

Notes:

1. cpRanks represents a complete set of processes, having MW_ProcessIDs, that are reserved for use by the Command Program. gRanks represents all processes reserved for use by the Pep to execute the graph. cpRanks and gRanks are disjoint. The Command Program decides the values of these.
 2. This call must be made on all processes reserved for use by the Pep to execute the graph and on a process from which the Command Program will call methods to pass data to and from the graph and call stopGraph.
 3. An exception shall be thrown if the arguments are inconsistent.
 4. The units of maxLatency are nanoseconds. If maxLatency is not specified by the Command Program, a default is given.
-

A call made by the Command Program to start executing a graph;

```
static void Pep::startGraph(const GCL_Graph& theGraph);
```

Notes:

1. This call must be made on all processes where the graph was constructed. On those processes reserved for use by the Command Program *startGraph* will return promptly. On those processes reserved for use by the Pep to execute the Graph, *startGraph* will only terminate by use of the *stopGraph* command (see below).
-

A function that the Command Program calls to stop the Pep executing the graph;

```
static void Pep::stopGraph(const GCL_Graph& theGraph);
```

Notes:

1. This call will be made on a single process where the command program is running.
 2. An exception will be thrown if there is no graph running.
-

A function that the Command Program calls to direct the Pep to stop automatic reassignment.

```
Static void Pep::stopAutoReassignment ();
```

Notes:

1. The Pep continues to use the current assignment.
-

A function that the Command Program calls to direct the Pep to start automatic reassignment.

```
Static void Pep::startAutoReassignment ();
```

A function that the Command Program calls direct the Pep to save the current assignment.

```
Void Pep::saveAssignment(char *outFileName)
```

Notes:

1. Executes a *stopAutoReassignment*, sends a save message, then executes a *startAutoReassignment*.
 2. If there is an I/O failure, a warning is sent to standard error and the command has no other effect.
-

A function that the Command Program calls to direct the Pep to use a previously saved assignment.

```
Void Pep::commandAssign (char *inFileName)
```

Notes:

1. Executes a stopAutoReassignment, then sends a message to the UeberPep to use the indicated file for assignment.
 2. If there is an I/O failure, a warning is sent to standard error and the command has no other effect.
-

PEP / GCL Interface

The following are C++ declarations that serve to define interfaces between the Pep and the Graph Library.

Graph Object

A function that is called by the Pep to suspend the graph;

```
static void GCL_Graph::suspendGraph();
```

Notes:

1. This must be called in every process where there is a LocalPep. It is incumbent on the Pep to make sure that all LocalPeps issue the command to suspend execution by calling this procedure.
 2. The command program calls a suspend method (see below, *stopGraph*) of the Pep from a process that the command program runs on. The Pep sends messages to all LocalPeps executing the graph which then cease execution of Transitions. The Pep (via LocalPeps) then calls the GCL to suspend by using this call. When all tokens have been received on a particular processor that were sent to that processor, this call will complete.
 3. The current assignment information is part of the state of the suspended Graph.
-

A function to revive a suspended graph;

```
static void GCL_Graph::reviveGraph();
```

Notes:

1. The Command Program calls the Pep to start the graph (see above, *startGraph*) and the Pep calls *GCL_Graph::reviveGraph* to allow the graph to place any ready Transitions on the ReadyQueue.
 2. The graph takes whatever steps are necessary to make the last assignment effective.
-

A function call made by the Pep to allow the GCL to process incoming messages.

```
static void GCL_Graph::receiveMessages();
```

Notes:

1. This call will be made after each Transition execution.
 2. Receiving messages should not occur as a result of a call to *GCL_Trans::execute()*, as that will affect the monitoring of the time for the Transition execution.
-

A function call to get the number of nodes in a graph.

```
static unsigned int GCL_Graph::getNodeCount();
```

Note: This method returns the total number of nodes in the flat graph.

A function that returns the address of a Transition or Place given the GCL_NodeID;

```
typedef unsigned int GCL_NodeID;  
static GCL_Node * GCL_Graph::getNode(GCL_NodeID);
```

Notes:

1. The address returned is the address of the Transition or Place in the local process. It is the Pep's responsibility to cast the returned address to the respective subclass.
 2. This method can be called for any Place or Transition object.
 3. If the NodeID is invalid, an exception will be thrown.
-

A function that returns the run-time ID of a Transition;

```
GCL_NodeID GCL_Node::getNodeID();
```

Notes:

1. Under-the-hood, GCL_NodeID will be typedef to unsigned int. Every Node (Transition or Place) has a unique GCL_NodeID.
-

A function that returns the kind of Node, e.g., Queue or Graph Variable;

```
enum GCL_NodeType {
```

```

    CPNODE,
    ORDINARY_TRANSITION,
    PACK,
    UNPACK,
    SWITCH,
    MERGE,
    ORDINARY_QUEUE,
    FUNNY_QUEUE,
    ORDINARY_GVAR,
    FUNNY_GVAR
};

static GCL_NodeType GCL_Graph::getNodeTypes(GCL_NodeID);

GCL_NodeType GCL_Node::getNodeTypes();

```

Notes:

1. Under-the-hood, GCL_Place and GCL_Tran are derived from GCL_Node, and the public method getNodeTypes is defined in the GCL_Node class.
-

A function that returns the CPRank of a CP Node;

```
static MW_ProcessID GCL_Graph::getCPRank(GCL_NodeID);
```

Notes:

1. This method first determines that the argument of type GCL_NodeID is the node identifier of a CPNode (i.e., of a pseudo-transition that serves as a graph inport or outport and interacts with the command program). If the argument identifies a CPNode, the method returns the MW_ProcessID (one of the cpRanks) to which the CPNode is assigned. If the argument does not identify a CPNode, the method throws an exception.
 2. The Pep will call this function during execution of Pep::createAssignment() to determine the process to which each CPNode is assigned.
-

A function that returns identifiers for a node that persist between runtimes and is used to record execution performance;

```

#include <string>

class GCL_StringIDs{
public:
    GCL_StringIDs (string familyName, string className, const string
        &graphClassName, const string &graphVersion, const string
        &graphFileName);

    const string &get__familyName () const; // double underscore
    const string &get__className () const;
    const string &get__graphClassName () const;
    const string &get__graphVersion () const;
    const string &get__graphFileName () const;
}

static const GCL_StringIDs & GCL_Graph::getStringIDs (GCL_NodeID);

```

1. User-defined Transition class names are in the name-space scope of the included graph that they are immediately contained within. Graph classes are of global scope. Primitives are contained in their own included graph. The term *graph* in the attribute names refers to the graph immediately containing the node.
2. The function *getStringIDs* shall be callable when *createAssignment* is called.
3. If the GCL_NodeID is invalid, either out of range or referring to a CPNODE, an exception will be thrown.
4. This method imposes specific requirements on the GUI and Translator to provide the required information.

A function call to notify the GCL of a Reassignment;

```

static void GCL_Graph::startReassignment ( const GraphRanks & localPepRanks );

```

Notes:

1. This call shall be made from every LocalPep. The new assignment information is made available to the GCL via calls to *Pep::transitionAssignment*.
 - 1a. The argument provides a list of the process identifiers of every LocalPep. This list shall be in the same order in every LocalPep that makes the call.
2. The GCL will initiate the process of reassignment by initiating the sending all messages required. The return from this call does not imply that the reassignment has been completed: only a call to

finishedReassignment signifies that. A call to *Pep::finishedReassignment* (see below) shall not be made within this method.

3. The LocalPep shall not call *GCL_Place::setCapacity()* between its calling *startReassignment* and the GCL subsequently calling *finishedReassignment*.

A method to indicate if a place has been initialized by the user:

```
static bool GCL_Graph::isPlaceInitialized(GCL_NodeID place);
```

Notes:

1: The Place does not need to provide actual token amounts or values, just show whether the user has specified initialization.

2. This method must be callable from the method *Pep::createAssignment(...)* and thereafter.

A method to return the process ID to which a place has been assigned.

```
static MW_ProcessID GCL_Graph::getPlaceAssignment(GCL_NodeID place);
```

Notes:

1. This may only be called after the Pep has returned from a call to *Pep::createAssignment(...)*.

2. This method shall not be called on a process after a call to *GCL_Graph::startReassignment()* until a subsequent call is made by the GCL to *Pep::finishedReassignment()*.

Places

A function that returns the current capacity of a Place;

```
unsigned int GCL_Place::getCapacity();
```

Notes:

1. Returns INT_MAX for a Graph Variable.

A function that modifies the capacity of a Place;

```
void GCL_Place::setCapacity(unsigned int);
```

Notes:

1. We specify that the argument is unsigned int, because we attach no meaning to a negative capacity.
 2. Setting the capacity of a graph variable is a no-op. A warning shall be generated.
-

A function that returns the current Content (in number of Tokens);

```
unsigned int GCL_Place::getContent();
```

Notes:

1. A graph variable will always have a content of 1. A warning shall be generated if this method is called for a Graph Variable.
-

A function that returns the ID of the nth Token on the Place;

```
GCL_TokenID GCL_Place::getTokenID(unsigned int);
```

Notes:

1. Tokens in a Place are indexed starting at 1. Thus the argument is 1 to indicate the first token, i.e., the one currently at the “head” of the Place. If the argument is greater than the current content or less than one an exception is thrown.
 2. GCL_TokenID is typedef to unsigned int. The first token ID is 1, and subsequent token IDs are incremented sequentially.
 3. If the Place is a GVAR an exception will be thrown if the argument is not 1.
-

A function that returns the MPI message size in bytes;

```
unsigned int GCL_Place::getMessageByteCount();  
unsigned int GCL_Place::getMessageByteCount(unsigned int);
```

Notes:

1. When no argument is specified and the downstream Transition is on a different processor from the Transition producing the token, *getMessageByteCount* returns the size of the message last sent. This call is only meaningful immediately after a Transition producing the token (or tokens) has been

executed. If the Transition is an Unpack, the message size includes all Tokens produced (even if the tokens are sent individually). This imposes a requirement on Middleware. The Place must get the size of a message sent from the Middleware each time a token is sent in a message. The Place shall retain the most recent value and return it whenever the method is called (with no argument).

2. When no argument is specified and there is no message that has been just sent, an exception shall be thrown. There may be more than one such exception type.

3. When an argument is specified, *getMessageByteCount* returns the size of the MPI message size that would be required to pass the token. The *unsigned int* argument indicates the position of the token on the queue.

4. When an argument is specified and the indicated token position is invalid an exception shall be thrown.

Transitions (All)

A method to execute a Transition;

```
#include <vector>
const vector<unsigned>& GCL_Trans::execute();
```

Notes:

1. GCL_Trans is the common base class for all Transitions, Special and Ordinary.

2. Calling this method assumes that the specified Transition is ready for execution. An exception will be thrown if the specified Transition is not ready.

3. The Pep will only call this method after first removing the Transition from its ready queue. If the Transition is ready to fire again after execution, the execute method must push it back onto the ready queue (see *Pep::addMeToReadyQueue* below).

4. Each element of the referenced vector (return value) will contain the size, in number of leaf nodes, of a corresponding family of input tokens to the Transition. The size of a given family of inputs shall always be indicated by the same element index. The order of the indicated family sizes shall not vary if the Transition class definition does not vary.

Special Transitions

A function that returns the next Consume Amount for a Pack Transition;

```
unsigned int GCL_Pack::getNextConsume();
```

Notes:

1. If the next Consume amount is undefined an exception is thrown.
-

A function that returns the last Consume Amount for a Pack Transition;

```
unsigned int GCL_Pack::getLastConsume();
```

Notes:

1. The last Consume amount may be undefined because the Pack has not fired since the beginning of graph execution or since the last reassignment, in which case an exception is thrown.
-

A function that indicates if the Consume Amount is constant for a Pack Transition;

```
bool GCL_Pack::isConsumeConstant();
```

Notes:

1. Indicates whether the Consume Amount is constant (true) or cannot be determined to be constant (false).
-

A function that returns the next Threshold Amount for a Pack Transition;

```
unsigned int GCL_Pack::getNextThreshold();
```

Notes:

1. If the next Threshold amount is undefined an exception is thrown.
-

A function that returns the last Threshold Amount for a Pack Transition;

```
unsigned int GCL_Pack::getLastThreshold();
```

Notes:

1. The last Threshold amount may be undefined because the Pack has not fired since the last reassignment, in which case an exception is thrown.
-

A function that indicates if the Threshold Amount is constant for a Pack Transition;

```
bool GCL_Pack::isThresholdConstant();
```

Notes:

1. Indicates whether the Threshold Amount is constant (true) or cannot be determined to be constant (false).
-

A function that returns the Place ID of the Place connected to the input port of a Pack Transition;

```
GCL_NodeID GCL_Pack::inputPlaceID();
```

A function that returns the last produce amount for an Unpack Transition;

```
unsigned int GCL_Unpack::getLastProduceAmt();
```

Notes:

1. The Unpack Transition must retain the most recent unpack amount and return its value whenever this method is called.
 2. If an empty token is unpacked, then the last produce amount is zero.
 3. If the Unpack has never previously executed, in which case the last produce amount is undefined, an exception will be thrown.
-

A function that returns the next Produce Amount from an Unpack;

```
unsigned int GCL_Unpack::getNextProduceAmt();
```

Notes:

1. This function should only be called if the input token is available, otherwise an exception will be thrown.

A function that returns the Place ID of the Place connected to the output port of an Unpack Transition;

```
GCL_NodeID GCL_Unpack::outputPlaceID();
```

Interfaces to the Pep

A function called by the GCL to create an initial assignment of the Transitions to the processes;

```
typedef unsigned int GCL_NodeID;
typedef struct{
    GCL_NodeID from;
    GCL_NodeID to;
} GCL_ConnectionPair;
typedef vector<GCL_ConnectionPair> GCL_Topology;

static void Pep::createAssignment(const GCL_Graph&, const CPRanks&,
                                const GraphRanks&, const GCL_Topology&,
                                int NumberOfNodes, double maxLatency);
```

Notes:

1. The function createAssignment uses a barrier synchronization and must be called from all processes that the graph is being built on.
 2. The topology will be constructed as the flat graph is being constructed. The topology will be a data member of the main graph object.
 3. The command program must pass CPRanks and GraphRanks to the graph constructor so that they can in turn be passed to the createAssignment method.
 4. The units of maxLatency are nanoseconds.
-

A function to provide the assignment information to the GCL;

typedef unsigned int ProcessRank;

ProcessRank Pep::transitionAssignment(GCL_NodeID);

Notes:

1. This function may be called after *createAssignment* has returned. Calling *TransitionAssignment* prior to calling *createAssignment* or during an ongoing reassignment will result in an exception being thrown.
 2. An exception will be thrown if an invalid NodeID is passed.
-

A function for the GCL to push a Transition onto the ready queue;

*static void Pep::addMeToReadyQueue(GCL-Tran *);*

Notes:

1. If the Transition is already on the ReadyQueue an exception will be thrown.
-

A function for the GCL to remove a Transition from the ready queue;

*static void Pep::removeMeFromReadyQueue(GCL-Tran *);*

Notes:

1. Typically, the Transition will be Special except when the Graph is suspending or undergoing reassignment.
 2. If the Transition is not already on the ReadyQueue an exception shall be thrown. If the ReadyQueue does not exist an exception shall be thrown.
-

A function for the GCL to alert the Pep that a Special Transition will not be able to fire unless the Place capacity is changed for the Place that its Special port is connected to.

static void Pep::deadlockAlert(const GCL-Tran&);

Notes:

1. The value of the argument indicates the deadlocked Special Transition.
-

A function call for the GCL to notify the Pep that Reassignment is complete on a given processor;

```
static void Pep::finishedReassignment();
```

Notes:

1. This method shall be called exactly once for each call by the Pep to the method *GCL_Graph::startReassignment*. By calling this method the GCL signifies that reassignment initiated by the prior call to *GCL_Graph::startReassignment* is complete on that process. Calls outside of this context will cause an exception.
-