

# **Interfacing the Command Program to a PGM Graph**

by

**Christopher Scannell**

**June 18, 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>

This document describes information for the Processing Graph Method (PGM) Command Program writer to use in interfacing his software with a PGM Graph. It will discuss calls that need to be made to create graphs, start and stop them, write to and read from them via a Graph Port, save and restore assignments of PGM Transitions to processes, and enable and disable dynamic reassignment of PGM Transitions to processes by the PGM Execution Program (PEP). Where relevant, it will discuss the specific API's of the PEP and the Graph Class Library (GCL) needed by the Command Program writer. Examples will be given from the PGM demonstration application (GramGraph) and from a simpler application, Sort.

This document will not discuss the details of drawing a Graph in the GUI, developing user-defined data structures for use in a PGM Graph, compiling and linking a PGM application or the configuration of mpi-related files for the runtime invocation of PGM applications. Nor will this document discuss implementation details of the PEP that are hidden from the Command Program writer such as distinctions between Ueber PEP processes and Local PEP processes and whether or not multiple Local PEP processes can run on a single processor. Discussions of running multiple graphs within a PGM application, either serially or simultaneously, and the implications of doing so on the Command Program will not be entertained.

## **Model**

The current implementation of the Processing Graph Method Tool (PGMT) is based on the distribution of the same executable to all processes involved in the execution of the PGM application. The Command Program writer defines `CommandProgram::run()` which is invoked after all initialization of the Middleware and PEP have been completed. Once the execution of the run function terminates, the finalization methods of the various components of the PGM application are called and the application exits. For that reason, the run function is usually structured as a series of statements designed to prepare data structures that should be identical across all processes, such as creation of PGM graphs, followed by a switch statement that evaluates the rank of the PGM process and executes the appropriate case statement to perform the desired behavior for that process. All code prior to the switch statement is thus executed on all processes with similar results across all processes. This section of `CommandProgram::run()` is referred to as the part of the run method "before differentiation".

```

#include "PGMT.h"
#include "MyGraph.h"

void CommandProgram::run( int argc, char** argv ){

    MW_ProcessID myRank = Machine::getCurrentProcessID();

    /* port assignments defined here */

    CPRanks cpRanks;
    cpRanks.push_back( 0 ); /* the 0'th process is
                           a Command Program process */

    GraphRanks graphRanks;
    int numProcs = Machine::getNumProcesses();
    /* All other processes are PEP processes */
    for( int i=1; i<numProcs; i++ ) graphRanks.push_back( i );

    const GCL_Graph* theGraph;

    theGraph = new MyGraph( const CPRanks& cpRanks,
                           const GraphRanks& graphRanks,
                           const GraphPortAssignment& pAssignment,
                           double maxLatency );

    switch( myRank ){

    case 0:
        while( still_interested_in_running_graph ){
            /* Do stuff with Graph */
        }
        PGMT::stopGraph( *theGraph );
        break;

    default:
        PGMT::startGraph( *theGraph );

    }
}

```

**Figure 1 Sample CommandProgram run method fragment**

The current implementation of PGMT assumes that the creation of the PGM graph (and therefore the assignment of Graph Ports to Command Program processes) will be done during this time. The other interactions between the Command Program and the PGM Graph, related to the scheduling of Transitions, to the accessing Graph Ports and to the starting and stopping of the PGM Graph, must occur after differentiation, i.e. after evaluation of the switch expression. The value of the switch expression is a function of the rank of the PGM process. This is because the starting of a PGM Graph must be done after differentiation. The processes that will be used to run the graph (from here on referred to as the PEP processes) must be identified when the graph is created (before differentiation) and after differentiation (i.e. in the appropriate case statement of the run method) the graph must be started on these processes. The burden is on the Command Program writer to make sure that the graph is indeed started on and only on those processes identified as PEP processes during the creation of the graph.

The communication between PEP processes is accomplished using the MPI message passing libraries. There is no reason that processes not under control of the

PEP, i.e. the Command Program processes, cannot use MPI as well for communicating between processes and indeed this is what was done for the GramGraph application.

## **Graph Creation**

The PGM Graph is created by calling the constructor for the desired graph. If the PGMT GUI is used to generate the C++ for the PGM Graph, the definition of the Graph's constructor will be found in the generated header file for the Graph. The arguments for the constructor for the Graph will include the definition of the processes assigned to the PEP (the graphRanks) and the Command Program processes that will communicate with the PEP processes (the cpRanks). A call to the constructor will be of the following form:

```
theGraph = new MyGraph( [other arguments,]
                        const CPRanks& cpRanks,
                        const GraphRanks& graphRanks,
                        const GraphPortAssignment& portAssignment,
                        double maxLatency );
```

**Figure 2 Graph constructor invocation**

Any process in the cpRanks vector of processes can interact with the Graph, but only those with Graph Ports assigned to them can write or read from the Graph. Graph Port Assignments will be discussed in the section on Writing and Reading from the Graph. maxLatency is used by the PEP scheduler in calculating assignments of Transitions to processes. When no solution has a latency less than maxLatency, the best assignment is considered to be the one that minimizes latency. When assignments can be found that satisfy the maxLatency constraint, the best solution is the assignment that gives maximum throughput of data through the graph.

## **Starting and Stopping a Graph**

As indicated in the Command Program code fragment in figure 1, PGMT::startGraph() is called from each PEP process. In this example, the graph does not stop running until the call to PGMT::stopGraph() from the only Command Program process (the rank 0 process in this case). In the Command Program code fragment from the GramGraph application, figure 3, we see that there are four separate Command Program processes that have different behaviors. One process (INHAND) writes data to the input port of the graph, one (OUTHAND) reads data from the output port of the graph, one (FRONTEND) interprets the interactions of the user with the GUI, and one (EXECUTOR) manages the running of the graph. (Note: several classes inherit from CommandProgram in the GramGraph application and many of the methods in the run method are virtual methods defined in the subclasses.)

```

void CommandProgram::run( int argc, char* argv[] ){

    GCL::log( BEGIN, "void CommandProgram::run (int argc, char* argv[])" );

    /*****
    /* Attach the Graph ports */
    *****/

    DBG( "In CommandProgram::run" );

    vector< int > depth0;
    vector< int > depth1( 1 );
    GraphPortAssignment portAssignment;
    for (int i=0;i < 1 ;i++) {
        depth1[0]=i;
        portAssignment.insert( GraphPortPair( GraphPortID( shadingGP,depth1),
                                                EXECUTOR ) );
    }
    portAssignment.insert( GraphPortPair( GraphPortID( windowsizeGP,depth0),
                                                EXECUTOR ) );
    portAssignment.insert( GraphPortPair( GraphPortID( lofreqGP,depth0),
                                                EXECUTOR ) );
    portAssignment.insert( GraphPortPair( GraphPortID( angleGP,depth0 ),
                                                EXECUTOR ) );
    portAssignment.insert( GraphPortPair( GraphPortID( integrationGP,depth0),
                                                EXECUTOR ) );
    portAssignment.insert( GraphPortPair( GraphPortID( outputdataGP,depth0),
                                                EXECUTOR ) );
    portAssignment.insert( GraphPortPair( GraphPortID( inputdataGP,depth0),
                                                INHAND ) );

    /*****
    /* Define the Command processes (cpRanks) on which the graph will */
    /* be built and the graph processes ( granks). */
    *****/

    get_cpRanks().push_back( EXECUTOR );
    get_cpRanks().push_back( INHAND );
    for( int i=4; i < Machine::getNumProcesses(); i++ )
        get_gRanks().push_back(i);

    /*****
    /* Build the graph on the appropriate processes */
    *****/

    bool isInCPRanksVector = find( get_cpRanks().begin(), get_cpRanks().end(),
                                   rank() ) != get_cpRanks().end();
    bool isInGRanksVector = find( get_gRanks().begin(), get_gRanks().end(),
                                   rank() ) != get_gRanks().end();
    if ( isInCPRanksVector || isInGRanksVector ) {
        GCL::buildmsg( "Calling graph constructor with numPhones = ", numPhones );
        double start = Machine::PGM2_wall_clock(0);
        set_graph( new Build3_gramGraph( numPhones,get_cpRanks(), get_gRanks(),
                                         portAssignment ) );
        double duration = Machine::PGM2_wall_clock( start )/1000000000.;
        DBG( Machine::getCurrentProcessID() << " time to build graph "
            << duration << " secs" );
    }

    /*****
    /* Define what happens on each process */
    *****/

    switch( rank() ){

```

```

case EXECUTOR:
/*****
/* The Executor process is the main control process for the Demo */
/* Graph. */
*****/
{
    DBG( "EXECUTOR is running on " << Machine::getCurrentProcessID() );

/*****
/* The process now loops through */
/* reading and handling messages from the FrontEnd process */
/* reading data from the output port of the graph and sending */
/* it to the OutHand process */
/* until quit is pressed on the FrontEnd GUI. */
*****/

    bool autoReassign = false; // TEST
    DBG( "Calling stopAutoReassignment()" );
    PGMT::stopAutoReassignment();
    while( quitNotPressed() ){
        rcvFrontEndToExecutorMessage();
        if( ! graphSuspended() ){
            if( readFromOutputPort() ){
                sendExecutorToOutHandMessage();
            }
        }
    }
    DBG( "Done with while loop" );

/*****
/* This process waits two seconds and then removes all tokens from */
/* output port and throws them away before finally exiting. */
*****/
    sleep( 2 );
    while( readFromOutputPort() ){} // Let data fall on the floor
}
break;

case FRONTEND:
/*****
/* The Frontend process is the control process for the Demograph GUI */
*****/
{
/*****
/* Start the java GUI running and establish socket with the process. */
/* The socket number is given by the second parameter , if present, */
/* passed on the command line and otherwise defaulted to */
/* DEFAULT_GUI_PORT (3007) */
*****/

    DBG( "FRONTEND is running on " << Machine::getCurrentProcessID() );
    int socketPort;
    if( argc > 2 ) {
        socketPort = atoi( argv[2] );
    } else {
        socketPort = DEFAULT_GUI_PORT;
    }
    char socketPortInt[10];
    sprintf( socketPortInt, "%d", socketPort );
    string str;
    GCL::buildmsg( "socketPort = ", socketPort );
    GCL::buildmsg( "DISPLAY environment variable set to ",

```

```

        Machine::getDisplayProcessorName() );
str = str + "( PATH=$PATH:/usr/java/bin; export PATH; "
+ "DISPLAY=" + Machine::getDisplayProcessorName()
+ ":0.0 ; export DISPLAY; cd java; java JFrontEnd "
+ Machine::getProcessorName() + " " + socketPortInt + " & )";
GCL::buildmsg( "Command = ", str );
system( str.c_str() );
init_server( socketPort );
connection = accept_client();
if( connection ){
    while( quitNotPressed() ){
        checkGUI();
    }
} else {
    sendFrontEndToExecutorMessage( QUIT );
}
}
break;

case INHAND:
/*****
/* The InHand process controls the reading of raw phone data from the */
/* disk and writing it to the Demograph input port. */
*****/
{
/*****
/* The process now loops through */
/* reading and handling messages from the Executor process */
/* reading and handling messages from the FrontEnd processes */
/* reading data from the disk */
/* sending messages to the executor */
/* until quit is pressed on the FrontEnd GUI. */
*****/
DBG( "INHAND is running on " << Machine::getCurrentProcessID() );
while( quitNotPressed() ){
    rcvExecutorToInHandMessage();
    rcvFrontEndToInHandMessage();
    readDataFromDisk();
    sendInHandToExecutorMessage();
}
DBG( "Exiting inhand" );
}
break;

case OUTHAND:
/*****
/* The OutHand process controls the displaying of a beam spectrum in */
/* an X-Window display. The display is first opened and then messages */
/* to update the X-Window display are received from the Executor and */
/* processed until quit is pressed on the GUI FrontEnd */
*****/
{
DBG( "OUTHAND is running on " << Machine::getCurrentProcessID() );
openXWindow();
while(quitNotPressed()){
    rcvExecutorToOutHandMessage();
}
DBG( "Exiting outhand" );
}
break;

default:
/*****

```

```

/* All other processes are PEP processes. While quit is not pressed */
/* these processes wait for control messages from the Executor and */
/* them when they are received. The processes exit when quit is */
/* pressed on the GUI Frontend. */
/*****/
{
  DBG( "GraphWrapper(default) is running on "
        << Machine::getCurrentProcessID() );
  while( quitNotPressed() ){
    usleep( SLEEP );
    rcvExecutorToGraphWrapperMessage();
  }
}

DBG( "Exiting run loop" );
GCL::log( END, "void CommandProgram::run (int argc, char* argv[])" );
}

```

**Figure 3 CommandProgram::run() for GramGraph**

All other processes reach the default statement at the end of the switch where they alternately sleep and check for MPI messages sent by the the EXECUTOR Command Program process. They are referred to as GraphWrapper instances because they are Command Program processes until the user presses the “Start” button in the GUI, at which point they all receive MPI messages from the EXECUTOR which trigger them to call PGMT::startgraph(). From that point until the graph is stopped by a call to PGMT::stopGraph(), they are Graph processes.

```

void GraphWrapper::rcvExecutorToGraphWrapperMessage (){
/*****/
/* Look for a message from the Executor. There are only two types of */
/* messages that we expect. GO and QUIT. */
/*****/

while( Machine::nbRcvCPMessage( controlMessage, 1, PGMT_INT, EXECUTOR,
    PGMT_ANY_TAG, &returned_tag, &returned_source ) ){

  switch( controlMessage[0] ){

  case GO:

    /*****/
    /* When a GO message arrives, send an acknowledgement back to the */
    /* Executor and start the graph */
    /*****/

    GCL::log( MSG, "Received GO from Executor" );
    // First send back acknowledgement
    ackGraphWrapperToExecutorMessage( GO );
    // startGraph returns when a CP Process calls PGMT::stopGraph
    DBG( "Calling startGraph in GWrapper" );
    PGMT::startGraph( *get_graph() );
    break;

  case QUIT:

    /*****/

```

```

/* When a QUIT message arrives, send an acknowledgement back to the */
/* Executor and indicate that the quit button has been pressed      */
/*****
GCL::log( MSG, "Received QUIT from Executor" );
// First send back acknowledgement
ackGraphWrapperToExecutorMessage( QUIT );
// graph must be suspended if GraphWrapper functions are being called
quitNotPressed( false );
break;

default:

/*****
/* Any other message is invalid so exit                            */
/*****

GCL::buildmsg( "Bad message received by GWrapper", returned_tag );
exit( 1 );
}
}
}

```

**Figure 4 GraphWrapper::rcvExecutorToGraphWrapperMessage()**

The rcvExecutorToGraphWrapperMessage() function shown in figure 4 shows the call to start the graph when the GO message is received. This method, specifically the call to PGMT::startGraph(), does not return until the graph is stopped by a call to PGMT::stopGraph() made in a non-Graph process. The startGraph method takes one argument, a reference to the GCL\_Graph to be started.

In the case of GramGraph, the EXECUTOR calls PGMT::stopGraph() when it receives an MPI message from the FRONTEND which is monitoring the GUI. This occurs in the rcvFrontEndToExecutorMessage method shown in figure 5.

```

void Executor::rcvFrontEndToExecutorMessage() {

while( Machine::nbRcvCPMessage( controlMessage, 1, PGMT_INT, FRONTEND,
                               PGMT_DEFAULT_TAG, &returned_tag,
                               &returned_source ) ){

switch( controlMessage[0] ){

case GO:

/*****
/* GO was pushed. If the graph was suspended send GO messages to  */
/* and the Pep processes. Set graph to unsuspended state before  */
/* returning.                                                       */
/*****

GCL::log( MSG, "GO" );
if( graphSuspended() ){
sendExecutorToInHandMessage( GO );
DBG( "Executor sending GO message to GraphWrappers" );
sendExecutorToGraphWrapperMessage( GO );
graphSuspended( false );
}
break;

```

```

case SUSPEND:

    /*****
    /* SUSPEND was pushed. Send Suspend message to InHand, stop Pep */
    /* and set graph to suspended state before returning. */
    /*****

    GCL::log( MSG, "SUSPEND" );
    if ( ! graphSuspended() ){
        sendExecutorToInHandMessage( SUSPEND );
        DBG( "Executor suspending graph from GUI" );
        PGMT::stopGraph( *get_graph() );
        graphSuspended( true );
    }
    break;

case QUIT:

    /*****
    /* QUIT was pushed. Then */
    /* Send suspend message to InHand */
    /* Stop the Pep */
    /* Set graph to suspended state */
    /* Send QUIT acknowledgement to InHand, wait for acknowledgement */
    /* Send QUIT message to OutHand and Peps */
    /* Set quitnotPressed flag */
    /*****

    GCL::log( MSG, "QUIT" );
    PGMT::saveAssignment( "assignment_save_file" );
    if ( ! graphSuspended() ){
        DBG( "Executor suspending graph before QUIT" );
        sendExecutorToInHandMessage( SUSPEND );
        PGMT::stopGraph( *get_graph() );
        graphSuspended( true );
    }
    sendExecutorToInHandMessage( QUIT );
    sleep(1);
    sendExecutorToOutHandMessage();
    sendExecutorToOutHandMessage( QUIT );
    sendExecutorToGraphWrapperMessage( QUIT );
    quitNotPressed( false );
    break;

case UPDATE:

    /* Code to reconfigure the graph by writing to ports not shown here */

    } /* end switch */
} /* end while() */
}

```

**Figure 5 Portion of Executor::rcvFrontEndToExecutorMessage()**

In GramGraph the user at the GUI can start and stop the Graph as often as desired. There is no need for the graph to be reconstructed by calling the constructor. Pressing “Quit” in the GUI exits the application entirely by causing PGMT::stopGraph to be called in the EXECUTOR and MPI messages to be sent between all Command Program processes to exit their run loops.

## **Reading From and Writing to the Graph**

An instance of GraphPortAssignment is passed to the constructor of the graph that is to be constructed. This structure is built up by the Command Program writer by a succession of calls to insert instances of the GraphPortPair class as shown in figure 3. The GraphPortPair's constructor is invoked with an instance of the GraphPortID class and the rank of the process to which this Graph Port is being assigned. The GraphPortID constructor is passed the name of the Graph Port variable and a vector of integers representing the dimensionality of the graph port (see the PGM User's Manual).

An example of writing to a Graph Port is shown below in figure 6. The Command Program process INHAND checks to see that the graph is not suspended and that the last read from the data source was successful. If this is the case and there are less than ten tokens presently being held by the Graph Port, a token of rank 2 (of dimensionality 2) is written to the Graph Port using the GCL\_GraphInport\_T method putMatrix.

```
void InHand::sendInHandToExecutorMessage () {  
  
    if( !graphSuspended() && !readFailed ) {  
        try {  
            GCL_GraphInport_T<short> *inputdataPort;  
            const string inputdataGP( "INPUTDATA_GP" );  
            const unsigned int inputDataSize = 1024;  
            inputdataPort = (GCL_GraphInport_T<short> * )  
                get_graph()->getInPort( inputdataGP );  
            if ( inputdataPort == 0 ) {  
                throw PGM_MissingGraphInport( inputdataGP );  
            }  
            DBG( "content of inputdataPort = " << inputdataPort->getContent() );  
            if( inputdataPort->getContent() < 10 &&  
                inputdataPort->getAvailCapacity() > 0 ) {  
                readNeeded = true;  
                unsigned int numPhones = 16;  
                unsigned int inHeight = numPhones;  
                unsigned int inWidth = inputDataSize/inHeight;  
                bool inRetVal = inputdataPort->putMatrix( inHeight, inWidth, *data );  
                DBG( "inRetVal = " << inRetVal );  
            } else {  
                DBG( "No read needed" );  
                readNeeded = false;  
            }  
        } catch (GCL_Exception &e) {  
            e.handleWarning( e.printName(), e.printMessage() );  
            GCL::log( END, "Executor::writeToInputPort ()" );  
            throw;  
        }  
    }  
}
```

**Figure 6 Writing to a Graph Port**

## **Controlling the Scheduling of the Graph**

The Command Program writer has the ability to turn dynamic scheduling of the Graph on or off. By default the PEP will adjust the assignment of PGM Transitions to Graph processes dynamically while the Graph is running. The PEP monitors the performance of the graph and tries to determine whether or not there exists another assignment that is anticipated to produce better performance. If it finds such an assignment, it issues the new assignment to all Graph processes. The GCL carries out the change to the assignment and any movement of tokens between PGM Places that this necessitates.

The Command Program may block the PEP from changing the current assignment by calling `PGMT::stopAutoReassignment()` and remove the block by calling `PGMT::startAutoReassignment()`.

Additionally, the current assignment can be saved to a file by calling `PGMT::saveAssignment("file_name")`. An assignment, saved in this fashion, can be effected by the PEP by a call to `PGMT::commandAssign("file_name")` if certain requirements are met. For an assignment to be implemented by the PEP, the Graph must be the same Graph that was running when the assignment was saved (the same constructor was called to create the Graph) and the number of Processes running the Graph must be the same. If these requirements are not met or if the file does not exist, the behavior of the PEP is not defined.

The code fragment in figure 7 shows an example of starting and stopping dynamic scheduling as well as saving and restoring assignments.

```
DBG( "Calling stopAutoReassignment()" );
PGMT::stopAutoReassignment();
int iter = 0;
while( quitNotPressed() ){
    if( !( ++iter ) % 1500000 ) {
        if( autoReassign ) {
            DBG( "Calling startAutoReassignment()" );
            PGMT::startAutoReassignment();
            autoReassign = false;
            cerr << "AutoReassignment starting" << endl;
            DBG( "AutoReassignment starting" );
        } else {
            DBG( "Calling stopAutoReassignment()" );
            PGMT::stopAutoReassignment();
            autoReassign = true;
            DBG( "AutoReassignment stopping" );
            cerr << "AutoReassignment stopping" << endl;
        }
    }
    if ( !( iter % 200000000 ) ){
        PGMT::saveAssignment( "assignment_save_file" );
    }
    if ( !( iter % 250000000 ) ){
        PGMT::commandAssign( "assignment_save_file" );
    }
}

/* snip */
```

```
}
```

**Figure 7 Code fragment demonstrating controlling the scheduling of a Graph**