An Interactive Parallel Programming Environment Applied in Atmospheric Science*

Gregor von Laszewski
Mathematics and Computer Science Division, Argonne National Laboratory
gregor@mcs.anl.gov

Proceedings of the Seventh ECMWF Workshop on the Use of Parallel Processors in Meterology, Reading, UK, Nov. 2–6, 1996, pages 311–325. Bookchapter in: Making its Mark, Editors Geerd-R Hoffman and Norbert Kreitz, World Scientific, 1997.

Abstract

This article introduces an *interactive parallel programming environment*t (IPPE) that simplifies the generation and execution of parallel programs. One of the tasks of the environment is to generate message-passing parallel programs for homogeneous and heterogeneous computing platforms. The parallel programs are represented by using visual objects. This is accomplished with the help of a graphical programming editor that is implemented in Java and enables portability to a wide variety of computer platforms. In contrast to other graphical programming systems, reusable parts of the programs can be stored in a program library to support rapid prototyping. In addition, runtime performance data on different computing platforms is collected in a database. A selection process determines dynamically the software and the hardware platform to be used to solve the problem in minimal wall-clock time. The environment is currently being tested on a Grand Challenge problem, the NASA *four-dimensional data assimilation* system.

Keywords: Graphical program design, visual programming, metacomputing, Java, data assimilation.

1 Introduction

During the development of programs to solve Grand Challenge problems, many diverse computing environments usually are used [13,15]. These environments include different hardware platforms and software packages and are used to solve various smaller subproblems in order to derive a solution for the overall problem.

*This material is based upon work conducted at Northeast Parallel Architectures Center at Syracuse University, supported by the National Aeronautics and Space Administration under Cooperative Agreement No. NCCW-0027. This work was continued under support by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Because of frequent changes in hardware and software, high overhead costs arise in the program design, maintenance, and execution on the ever-increasing number of computing platforms. Unfortunately, only limited time is available for scientific researchers to obtain detailed knowledge of the diverse software environments and hardware platforms. Therefore, in order to simplify the programming task and to reduce the development costs for the design of parallel programs, a graphical programming environment has been developed. The environment, called the *interactive parallel programming environment* (IPPE), allows scientists to embed their favorite software library while providing enough structural guidance to support parallel programming on the basis of a task-parallel approach.

This article describes some design issues of IPPE and its application on the Grand Challenge problem of *four-dimensional data assimilation* (FDDA) system used in atmospheric science studies.

The next section describes the requirements placed on the design of the environment for parallelizing scientific codes. Section 3 describes the environment developed to fulfill these requirements. Then, the usefulness of the environment is shown while applying it to the Grand Challenge FDDA problem (Sections 4–6). Section 7 points out similar research efforts, and Section 8 summarizes the conclusions and outlines future research directions.

2 Motivating Analysis

Many different approaches for parallelizing sequential programs exist. For example, during the parallelization of the FDDA, a top-down approach is used. A typical top-down approach includes the determination of major program blocks and the decision about which parallel programming paradigm is best suited for the parallelization of the problem.

In this article, a task-parallel programming paradigm is used. Several steps are necessary when converting a sequential program to a task-parallel program:

1. Determine the structure of the sequential program, and divide the program in smaller subproblems.
2. Based on the structure analysis, transfer the program into a task graph and include data dependencies.
3. Maximize the parallelism of the task graph.
4. Determine the tasks that are sequential, and reuse as much of the original sequential code as possible.
5. Determine and rewrite the parts of the task graph that are concurrent.
6. Generate a mapping from the task graph onto a virtual computing platform, which will be mapped onto a real hardware platform.
7. Execute the parallel program, and observe the performance.

To reduce the amount of effort needed to parallelize the serial code, the IPPE supports all steps of the parallel program design. Although the software developer still has to determine which parts should be parallelized, the IPPE supports the visualization of the task graph, its mapping onto a real machine, and the animation of the execution of the final parallel program. During use of the visual representation of the program (a task graph), the final parallel program is augmented automatically with the necessary parallel language constructs. Depending on the available hardware and software, these can be message-passing library calls (e.g., MPI or PVM) or additional language constructs (Fortran M, CC++, etc.).

Besides simplifying the coding effort, the IPPE programming environment takes care of the resource management. Therefore, the scientist can concentrate on the problem to be solved, rather than having to decide what resources are available and on which machine the program should be executed. Resources are selected in such a way that the wall-clock time for solving the problem is minimized, as it is necessary for many Grand Challenge problems. The next section summarizes some of the requirements arising from a particular application analysis that have influenced the design of the interactive parallel programming environment.

3 Application Analysis

The well-known method of *optimal interpolation* (0I) serves as a test application throughout this article. The OI algorithm, as used in four-dimensional data assimilation, interpolates from irregularly distributed observations into a three-dimensional grid. Later, this grid is used as input for a climate model. A detailed description of the sample application and its parallelization is beyond the scope of this article. More information about the scientific problem can be found in the literature. The parallelization of different versions of the optimal interpolation algorithm is described elsewhere[15-17].

Of major importance are the properties of the code, as shown in Table 1, which directly influence the design of the parallel software environment. This code analysis is typical for many scientific codes.

The sequential 0I code was developed by many programmers over the past decade. It has been transferred from very early computers to state-of-the-art vector supercomputers. Unfortunately, the documentation and code quality has suffered during this process. Because of the size of the program and its structural complexity, however, a complete redesign of the program is too expensive. Thus, any attempt at parallelizing the code should reuse as many parts of the original program as possible. This restriction also implies that, since the original program was written in Fortran, the core of the program should be kept in this language. Doing so facilitates maintenance by current and future atmospheric scientists who are most likely to know Fortran. Keeping the program in Fortran also provides portability of the code to other machines. Using the IPPE parallel programming environment improves the program documentation and the overall structure of the program because supporting features for both are embedded in the design of the environment. The environment is able to generate a message-passing parallel code that runs on a wide variety of parallel computing platforms, including heterogeneous computing platforms. The generated programs are executable in batch

mode for MIMD supercomputers, as well as interactively on distributed computing platforms performing in time-sharing mode. Currently, scientists are developing several other modules that are similar in functionality to the optimal interpolation algorithm. With the help of the IPPE parallel computing environment, these modules can be easily incorporated in the overall structure of the data assimilation system. After inclusion of the programs in a module library, an atmospheric scientist can use the module best suited for her/his individual goal (i.e., minimal wall-clock time and/or increased accuracy of the calculation). Use of IPPE and its expansion with user-defined modules will ultimately lead to a wide variety of predefined computational modules. The availability of these predefined modules will dramatically reduce the programming effort for building new programs. Moreover, the modules can be reused in other projects, thus enabling rapid prototyping.

Many of the decisions dealing with the resource management of the program will be hidden from the user and taken care of by the IPPE. The next section describes part of the actual metacomputing environment in more detail.

4 Parallel Programming Environment

The following list summarizes the principal requirements for the parallel programming environment. The environment should

- be easy to use for experts and nonexperts in the field of parallel programming;
- simplify the parallel program design;
- regulate the software development while providing support for documentation and structure;
- regulate and supervise the program execution;
- be expandable;
- have native language support for Fortran and other high-level languages;
- support heterogeneous computing environments; and
- be expandable to include WWW resources in the environment.

To fulfill some of these requirements, the IPPE environment follows a dataflow programming concept used by other similar approaches.3 To provide portability and to enable program execution on many MIMD machines, the message-passing paradigm is used to translate the visual program specification of the task graph into a program by using message-passing routines. After the generation of the message-passing program, the executable is executed on a statically or dynamically selected set of computers or processors.

Figure 1 shows the multiple functionality of the IPPE. The IPPE is used to simplify generating parallel programs or transferring a sequential program into a parallel program. Additionally, it can be used to supervise the execution of a sequential as well as a parallel program.

Internally, the IPPE comprises two main layers. First is a *message-passing interface*, intended to enable support of different computing platforms. Currently, the IPPE provides an interface to MPI. On top of this communication library a *parallel support library* provides the necessary functionality to simplify parallel programming. An example is a set of parallel vector routines necessary for the optimal interpolation algorithm or other scientific codes.5 Code generated with the help of the programming environment can be added to the pool of available modules.

Computationally intensive and time-critical applications are supported, while integrating high-level programming languages often used in scientific programming, for example, Fortran 77, Fortran 90, C, and C++.

While the original prototype of the parallel computing environment was developed in Tcl/Tk, Pen, CGI and Python, the current implementation of the interface is based on Java. This simplifies expansion toward the WWW-driven usage of the interface as suggested in the MetaWeb project [6]. Using Java also reduces the number of software packages involved in the core implementation of the computing environment.

5 Practical Application

After this relatively abstract introduction of the environment, some snapshots will illustrate the usefulness of the parallel programming environment. In Figures 2- 7, snapshots of the software environment are displayed as used during the parallelization and execution of the 0I algorithm.

Figure 2 shows the logical division of the sequential program. Data is generally shown in rectangles, while processes working on data are displayed in circles. Dependencies between data and tasks/processes are displayed with the help of directed edges. For better visualization, colors are used in addition to the obvious form distinction.

Once the processes with concurrent nature are defined, they are introduced into the process graph as shown in Figure 3. A parallel process is visualized with multiple circles, while distributed data is visualized with multiple rectangles. An example for such distributed data is the block distribution as known from HPF.

Data objects *flowing* between process objects are defined via a simple interface. For example, in the optimal interpolation algorithm, a simplified data structure for the observations and their location is used (Figure 4). The definition of data objects is similar to a record, as known from several Fortran 77 extensions and Fortran 90, and the struct command in C. The definition of the data type generates the necessary message-passing routines, allowing communication between the data objects and the process objects. Figure 5 shows the use of a data flow object in a procedure definition. The indirection of the data flow is marked with the special keywords IN DATA and OUT DATA. Thus, with only a very limited extension to the original sequential programming language (in this case Fortran), task-parallel programs can be defined easily and naturally. More interesting problems can be considered while sending dynamic data structures, as found in irregular problems. For future research, we point out that the extension of the data flow concept, with actual programs as data, will enable the distribution of programs similarly to the distribution of data. Special care has to be taken, however, in order to solve security issues. After the processes and the data objects

are defined, they have to be mapped onto a real computer to be executed (Figure 6). Restrictions during the code development (e.g., the code can be compiled only on one machine and is not portable) may limit the number of choices for the mapping. To minimize the overall wall-clock time of the program execution, dynamic load balancing is used to map the problem on the different processors and/or computers, based on their current load.

To support this strategy, a process monitor keeps track of the status and use of the machines. Figure 7 shows an example of some system variables (here CPU load, load average, and swap load) monitored to support the mapping strategy.

The load monitor helps to display performance bottlenecks of the parallel program during its execution on the machines, while collecting a time-space diagram.

In the example depicted, all processes are mapped to an IBM SP2, and the graphical display is viewed on a SPARC workstation. If the processes are written in a portable way, mapping onto other machines is possible too. Hence, one can execute a parallel program in a heterogeneous (super)computing environment.

6 Dynamic Selection of Software and Compute Resources

The dynamic execution of a program is driven by two factors. These are the software modules and the available computational nodes (hardware). Figure 8 illustrates the process responsible for making the selection of the hardware and software used to solve a problem. Often, a scientific problem is solved many times for similar instances of data. If the runtime is not significantly different between the instances, it can be used for performance prediction. In other cases, a performance prediction function can be used. Once the suspected execution time for a particular machine configuration is stored in the database, the information about the current utilization of the machine is used to predict the real-time performance of the program. If several choices of software and hardware mappings are available, the one with the shortest execution time is chosen. Hence, the selection not only includes a hardware mapping but also can include the usage of completely different algorithms that are best suited for the selected computer to solve the (meta)problem.

7 Related Research

Considerable research has been done in the field of visual programming in respect to parallel computing. The environment described in this article uses an approach similar to that introduced in HeNCE and CODE. A more detailed description of visual programming and their applications in parallel computing can be found in the literature.34 Unlike these other systems, however, the IPPE extends the usage toward a realistic metacomputer while providing a database of performance predictions that guides the selection and mapping of programming tasks to selectable resources.

8 Conclusion and Future Research

This article describes an interactive parallel programming environment. The environment makes it possible to view the available resources as a metacomputer and to reduce the development time for parallel programs. Dynamic process assignment is used to assist the execution of the parallel programs on diverse computing resources. This includes not only the selection of the best suited hardware platform but also the appropriate software for solving the problem.6 Many extensions are planned. One of the most striking will be the inclusion of a message-passing layer for the WWW. This will allow access to resources available via the WWW. Integration of a Fortran interpreter or language tools to simplify the distribution of programs is desirable. For mathematical problems, scripting languages like Matlab, Scilab, or Mathematica could be a viable alternative to interpreted Fortran or Java.

9 Acknowledgments

I thank Geoffrey C. Fox and Miloje Macivic for their support an valuable comments. I am grateful to the research team at the Data Assimilation Office, Goddard Space Flight Center, Greenbelt, MD, for they support during the project. I especially thank Peter Lyster, David Lamich, Jim Stobie, and Mike Seablom for their discussions in helping to understand the dark secrets of OI code. I also thank Richard B. Rood for his support and hospitality during several visits at NASA Goddard Space Flight Center as part of the Universities Space Research Association (USRA). Finally, I thank Gail Pieper and Warren Smith from Argonne National Laboratory for their comments on improving the paper.

References

1. MPI: A Message Passing Interface. Technical report, University of Tennessee, Knoxville, TN, 1994.

2. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HeNCE: Tools for heterogeneous network computing. In J. S. Kowalik and L. Grandinetti, editors, Software for Parallel Computation: Proceedings of the NATO Advanced Workshop on Software for Parallel Computation, held at Cetraro, Cosenza, Italy, June 2226, 1992, volume 106 of NATO ASI Series F, pages ix + 363, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1993. Springer-Verlag.

3. James C. Browne, Jack Dongarra, Syed I. Hyder, Keith Moore, and Peter Newton. Visual Programming and Parallel Computing. Technical report, University of Texas Austin and University of Tennessee at Knoxville, 1996.

4. James C. Browne, Syed I. Hyder, Jack Dongarra, Keith Moore, and Peter Newton. Visual programming and debugging for parallel computing. IEEE parallel and distributed technology: systems and applications, 3(1):75ff, 1995.

5. Roger Daley. Atmospheric Data Analysis. Cambridge Atmospheric and Space Science Series, Cambridge University Press, 1991.

6. Geoffrey C. Fox, Wojtek Furmanski, Marina Chen, Claudio Rebbi, and James H. Cowie. Web-Work: Integrated Programming Environment Tools for National and Grand Challenges. Joint Boston-CSC-NPAC Project Plan to Develop Web-Work, Northeast Parallel Architectures Center at Syracuse University, Syracuse, NY, 1996.

7. Geoffry C. Fox and Wojtek Furmanski. Neat Tools Overview. Technical Report unpublished, Northeast Parallel Architectures Center at Syracuse University, Syracuse, NY, 1996.

8. A. A. Khokhar, V. K. Prasanna, and M. E. Shaaban. Heterogenous Computing: Challenges and Opportunities. IEEE Computer, pages ppl8–27, 1993.

9. Andrew Lumsdaine, Jeffrey M Squyres, and Brian C. McCandless. Object Oriented MPI (OOMPI): A C++ Class Library for MPI. Technical report, Department of Computer Science and Engeneering, University of Notre Dame, Notre Dame, IN, 1996. http://www.cse.nd.edu/ lsc/research/oompi.

10. James Pfaendtner, Stephen Bloom, David Lamich, Michael Seablom, Meta Sienkiewicz, James Stobie, and Arlindo da Silva. Documentation of the Goddard Earth Observing System (GEOS), Data Assimilation System - Version 1. NASA Technical Memorandum 104606, Vol.4, NASA GSFC Data Assimilation Office, Greenbelt, Maryland, Jan. 1995.

11. Adrian Simmons. High Performance Computing Requirements for Medium Weather Forecasting. ECMWF Newsletter, Reading, UK, (69):814, Spring 1995.

12. Lawrence L. Takacs, Andrea Molod, and Tina Wang. Documentation of the Goddard Earth Observing System (GEOS), General Circulation Model - Version 1. NASA Technical Memorandum 104606, Vol.1, NASA GSFC Data Assimilation Office, Greenbelt, Maryland, Sep. 1994.

13. Gregor von Laszewski. The Parallelization of a Weather Prediction Model. Technical Report SCCS 533, Northeast Parallel Architectures Center at Syracuse University, July 1993.

14. Gregor von Laszewski. Minimal Requirements for a Graphical User Interface for Parallel Computing Applications. Technical Report SCCS 602, Northeast Parallel Architectures Center at Syracuse University, February 1994.

15. Gregor von Laszewski. Preliminary Performance of a Parallel Interpolation Algorithm. Technical Report SCCS 713, Northeast Parallel Architectures Center at Syracuse University, December 1995. first edition in June 1995.

16. Gregor von Laszewski. A Parallel Data Assimilation System and its Implications on a Metacomputing Environment. PhD thesis, Syracuse University, December 1996.

17. Gregor von Laszewski and et al. Design Issues for the Parallelization of an Optimal Interpolation Algorithm. In G-R. Hoffman and N. Kreitz, editors, Coming of Age: Proceedings of the 4th Workshop on the Use of Parallel Processing in Atmospheric Science, European Centre for Medium Weather Forecast, Reading, UK, pages 290–302. World Scientific, November 1994.

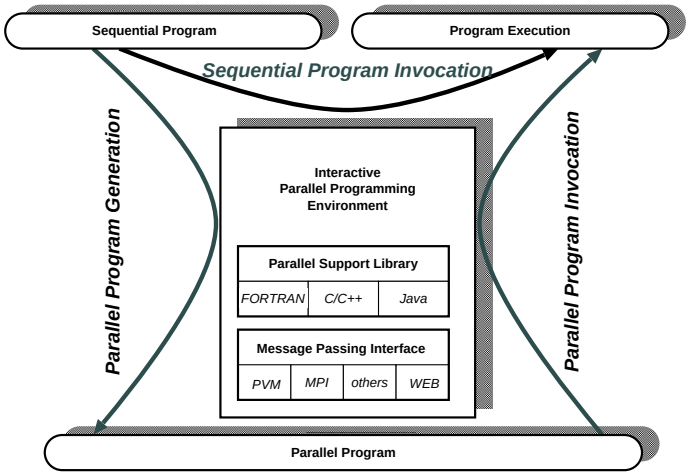

Figure 1: The multiple purpose of the parallel programming environment while creating and executing parallel programs.

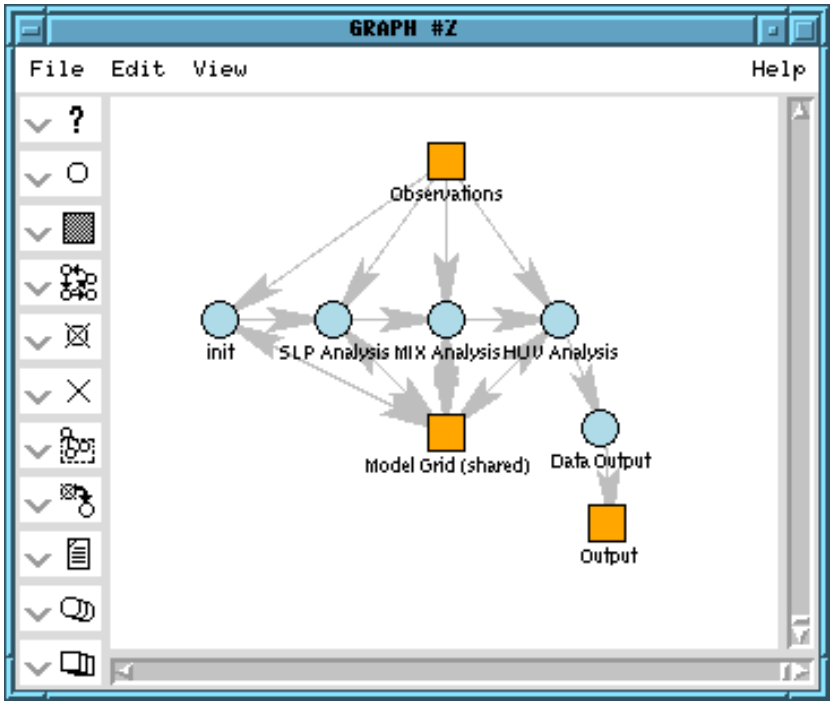

Figure 2: The window shows the building blocks used in the global program structure (tightly coupled metacomputing program).

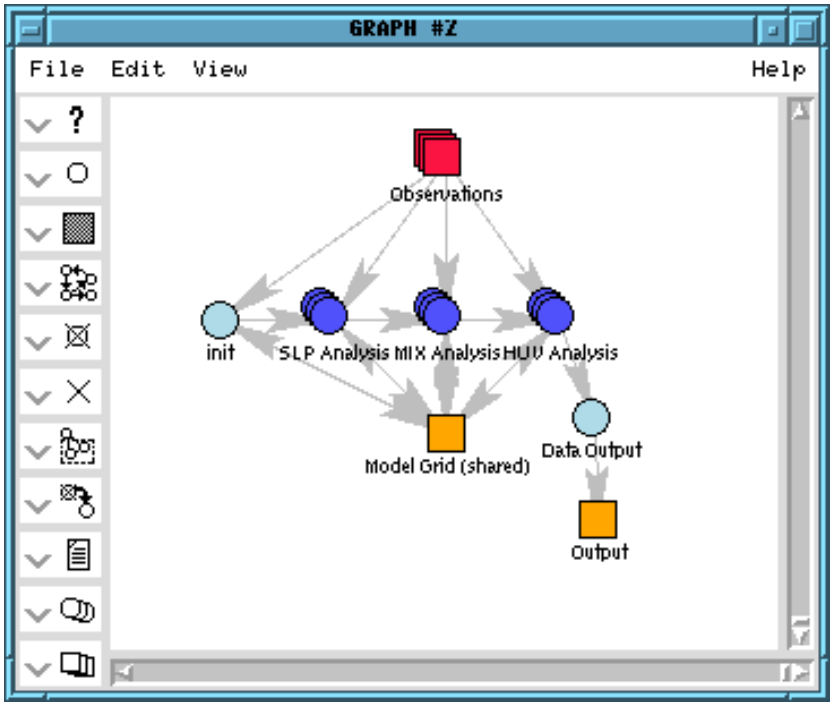

Figure 3: The window shows how the program is represented after the parallel program blocks have been introduced.

```
DATA OBJECT OBSERVATIONS
  INTEGER NoOfObservations;
  REAL x(NoOfObservations);
  REAL y(NoOfObservations);

  REAL temp(NoOfObservations);
  REAL pressure(NoOfObservations);
END DATA OBJECT
```

Figure 4: Definition of data able to flow between process objects. The data object is a simplified data object as used in the NASA project.

```
PROCESS  huv (IN_DATA OBSERVATIONS,
              IN_DATA MODEL_in,
              OUT_ DATA MODEL_out)

      !-- Quality Control
      do i=1,NoOfObservations
         call buddy_check
         call gross_check
      end do
      !-- Matrix Solve
      do i=1,NoOfGridpoints
         call set_up_the_matrix
         call solve_the_equations
      end do
END PROCESS huv
```

Figure 5: Definition of a process object using data objects on its inputs.

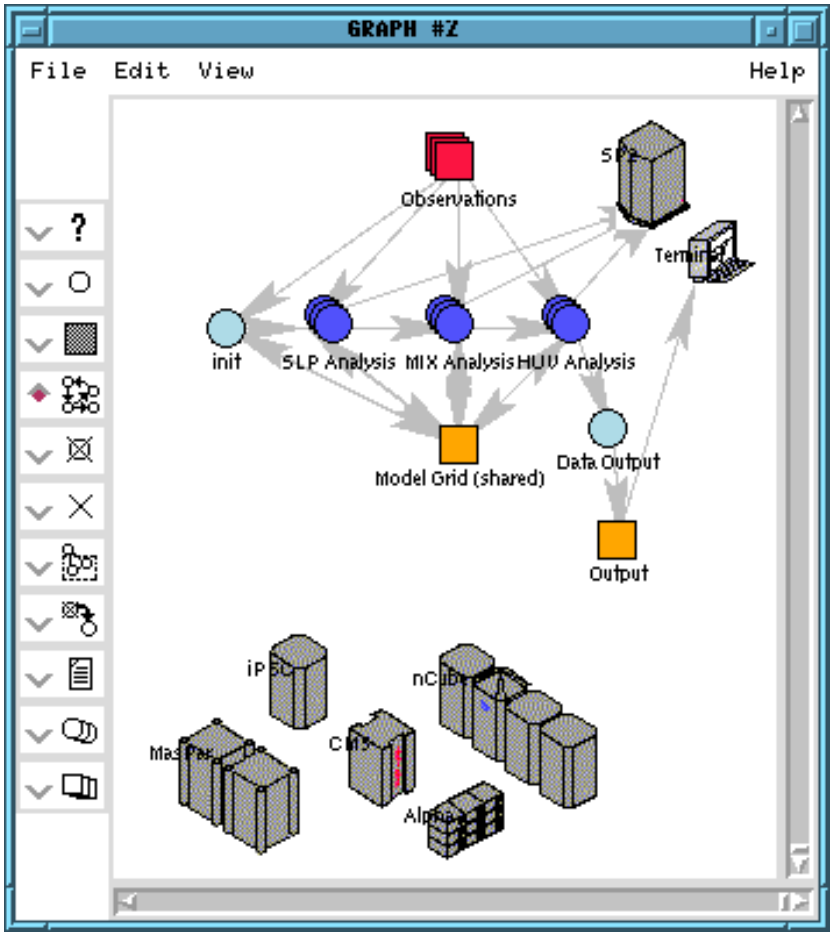

Figure 6: The window shows the selection of the machines participating in the execution of the program.

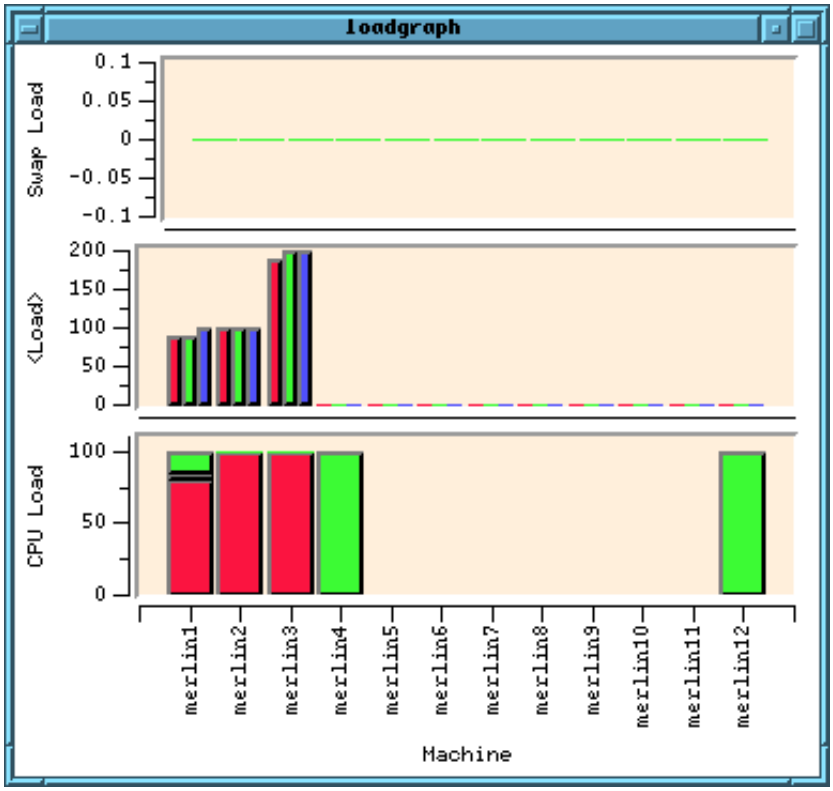

Figure 7: The window shows the load meter to control dynamic load balancing while executing the code.

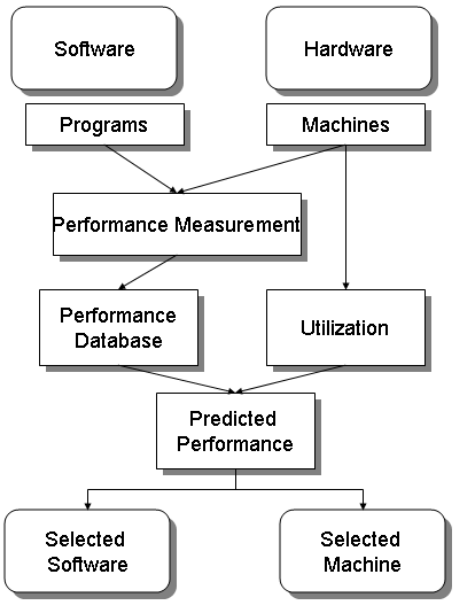

Figure 8: Dynamical selection process during program execution