OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Architecture of the Parallel Programming Support Environment

T. G. Lewis
W. G. Rudd
Oregon Advanced Computing Institute (OACIS)
&
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

90-80-2

Architecture of the Parallel Programming Support Environment

T. G. Lewis and W. G. Rudd

Oregon Advanced Computing Institute
and
Department of Computer Science
Oregon State University
Corvallis, Oregon 97333
lewis@cs.orst.edu
rudd@cs.orst.edu

Abstract

The Parallel Programming Support Environment (PPSE) is an experimental integrated set of tools for the design and construction of large software systems to run on parallel computers. The tools include a graphical design editor, a graphical target machine description system, a task mapper/scheduler tool, parallel code generator, and graphical aids for performance analysis. The objective is, to the extent possible, to design and develop parallel software with little regard for the details of the architecture of the target machine, programming language, or parallel computing paradigm that the program is to use.

1. Introduction

This paper provides a top-level view of the architecture of the Parallel Programming Support Environment, which is an experimental set of tools for use in the design and implementation of software systems for parallel computing systems. Our research in this area is an attempt to close the gap between what the world of research in computer science knows about programming languages and paradigms, scheduling, code generation, and applications, and what is available as tools in the development of software for parallel computers. We want to build a dynamic framework into which we can insert new modules that incorporate usable results as they arise. We want to use the resulting system as a testbed for these tools and for our ideas on the overall process of the design and implementation of parallel programs.

Using the PPSE Parallax graphical design editor, one can use a hierarchical graphical description language called ELGDF, Extended Large Grain Dataflow, to describe data and control dependencies between tasks. One then uses the Target Machine Editor to provide a graphical description of the architecture of the machine on which the program is to run, together with some performance parameters that indicate processor speeds, communications delays, and bus rates. The Task Grapher module takes the output from the Parallax editor and from the Target Machine Editor and, using several heuristics, generates Gantt chart descriptions of the mapping of tasks onto processors and their scheduling. One can use the resulting performance estimates as guides in modifying the software or hardware designs or both to improve performance. As an alternative, one can use the Superglue module to generate source code for the target machine, using code fragments for the lowest level tasks supplied through the Parallax editor or from other sources.

We have a prototype of the environment running on the Macintosh and we have used it to parallelize a small image processing application and several simple programs. Our current version of Superglue generates C-Linda code. Continuing research includes efforts to define a formal interface between the output from design tools and the PPSE tools, increased emphasis on data partitioning in the tools, expanding the list of target machines and languages, expanding the range of the schedulers, and "test drives" of the system on several significant applications.

We begin with a description of the design objectives and architecture of the system. We then present a report on the status of the system as of this writing, and conclude with a summary of what we have learned so far in this research and a description of plans for further work. A more detailed discussion of the components of the system is to be found in [RLEJ89] and [Lewi89].

2. Design Objectives

Currently, parallel computing architectures and software development systems are in a rapid state of flux. To avoid building a system that rapidly becomes obsolete and to try to make the system usable over a broad range of hardware and software environments, we have adopted two major design objectives for the PPSE. First, we want the system to be as independent of the target machine architecture, configuration, programming language, and operating system as we can make it. In other words, in the design and analysis process, we want to delay the binding to these target system-specific details as long as we can.

Our other major objective is that we want the system to accept software system design specifications from a wide variety of software development tools, including analyzers of existing programs, CASE tools, graphical design aids, and high-level design languages. This requires a clean and well-defined interface between the outside world and PPSE.

While there is nothing inherent in our approach that restricts the level of granularity one can select to use with our tools, the current implementations are best suited for a large-grained approach, in which the lowest level nodes are code modules or functions. The intent is that one can use tools and techniques that are suited for working on finer

grained approaches, such as vectorizing compilers, if the target machine has vector processors, loop unrolling strategies, and data parallel approaches to improve the performance of the code in the lowest level nodes.

Experience indicates that, for the foreseeable future, the development of parallel programs will be an iterative process, in which one must repeatedly go through a cycle of decomposing data and functions, perhaps altering the target machine description, doing the recoding that is necessary as a result of design changes, doing simulations to predict performance, and then running, testing, and doing performance analyses on the results. The PPSE is designed to make this process as easy as is possible.

We intend for the PPSE to serve as a framework into which we can incorporate new design tools, schedulers, code generators, and other components as they become available. Here again, the intent is to avoid binding to particular environments and technologies as much as we can in order to support the broadest range of options that we can.

3. Structure of the PPSE

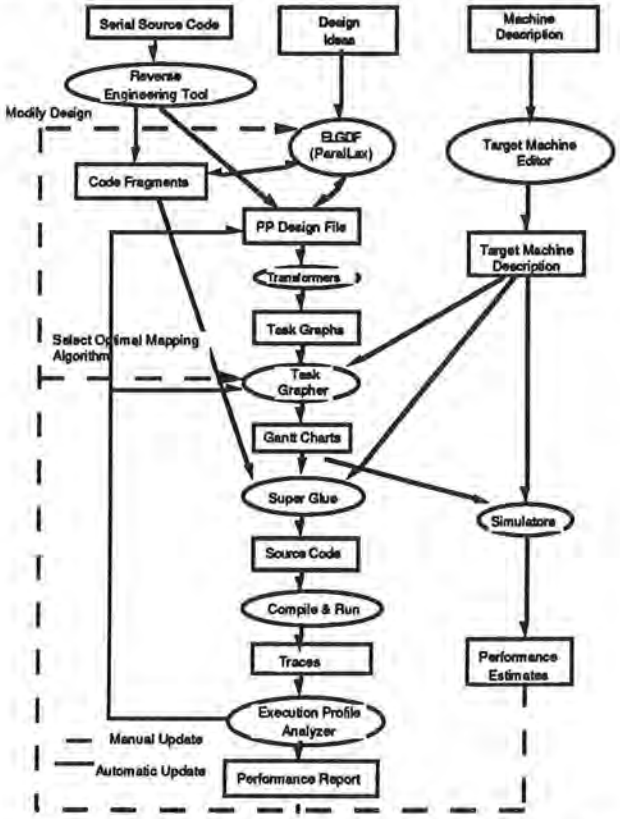

Figure 1. The top-level architecture of the Parallel Programming Support Environment. Ovals represent processes and rectangles represent information.

As can be seen from the overfiew of the system in Figure 1, we are concerned with both of the problems of software engineering for parallel computing systems. We want the system to be of use in the re-engineering process in which we want to parallelize existing "dusty deck" programs. Secondly, we want our tools to be applicable to the problem of developing new systems of parallel programs. Therefore, at the top of Figure 1, we indicate two sources of input for the initial program design and code. On the left side we indicate that a reverse engineering tool will accept existing programs and convert them into a form that our tools can process. In the center, we indicate that our Parallax graphical editor [Kim89] can also be used to input the original design for the software. In either case, the design - including data and control flow information - is captured in the PP Design File. The Parallax editor can then be used to alter the design, no matter what its original source was.

The Parallax editor works with a graphical design language called Extended Large Grain Dataflow, or ELGDF [ElLe89a]. This language arises from the same impetus as do LGDF2 [DiBa88], CODES [BASo89], and other high-level graphical design aids. It is a hierarchical language that offers several graphical constructs, such as pipes, loops, repeated nodes, and special dataflow arcs to indicate mutually exclusive access to data.

In order to predict performance later in the process, the user enters into the initial design description estimates of how long it will take to execute the tasks at the lowest-level nodes in the design specification and how much data will be transmitted on each dataflow arc. The schedulers and mappers and performance estimators use these estimates in performing their tasks. These data can be updated automatically with real values from the performance measurement tools once the program has been run on the target machine.

The examples we show in Figures 2 through 6 are from the use of the system in parallelizing a small (3,000 lines of Fortran) image processing application [JuRu89]. Figure 2 shows the top-level ELGDF diagram for the image processing example.

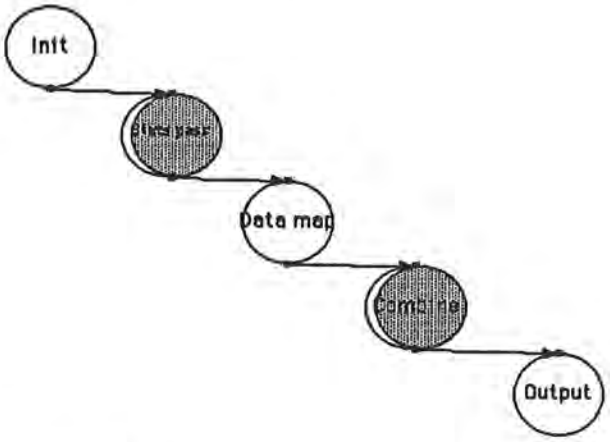

Figure 2. A top-level ELGDF diagram for the image processing example.

An important feature of the PPSE is that the user can obtain a great deal of experimentation with software and data decomposition and with matches of software designs to target hardware without actually writing any source code for the target hardware. Thus, the actual coding can be delayed until most of the design details have been tested through the use of the scheduling and simulation tools. Once the initial design has been developed and tried with the schedulers and simulators, one can use the Macintosh text editor from the Parallax editor, or any other editor, to generate the code fragment files. The code fragments are the software modules that carry out the processes indicated in the lowest-level nodes or tasks in the ELGDF design or in the PP Design File. As noted above, in practice these correspond to code modules or functions.

On the hardware side, we use our Target Machine Editor to enter the necessary description of the architecture of the target machine or network and its configuration. This editor allows the user to use a PMS-style graphical language [SBNe82] to specify arbitrary systems. The user enters data that indicate processor speeds and data transfer rates on communications paths. These data are combined to form a Target Machine Description File, which is used by the scheduling and mapping tools and by the code generator tool. Figure 3 shows a possible hardware configuration for our image processing example.

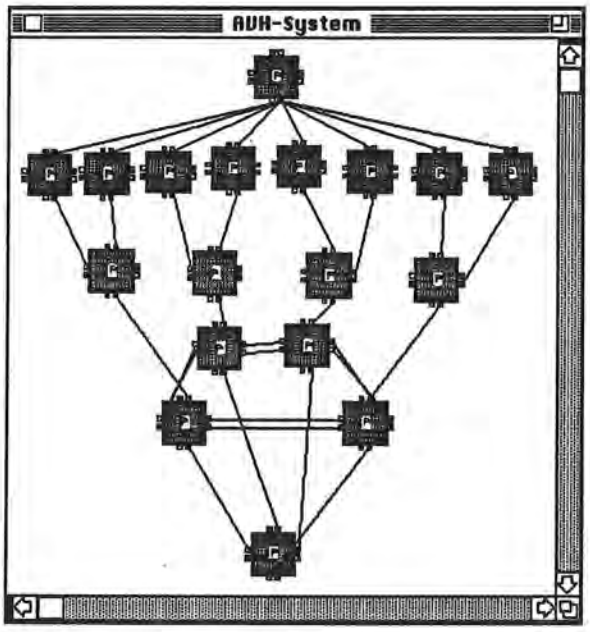

Figure 3. A proposed target machine architecture for the image processing problem.

Once the design specifications and target machine description have been created, or modified if this is not the first time through the software implementation cycle, the PP Design File is transformed into a task graph, which is simply a precedence graph with nodes weighted with estimated or measured execution times and arcs weighted with quantities of data to be transmitted. A Task Graph for the image processing example is shown in Figure 4.

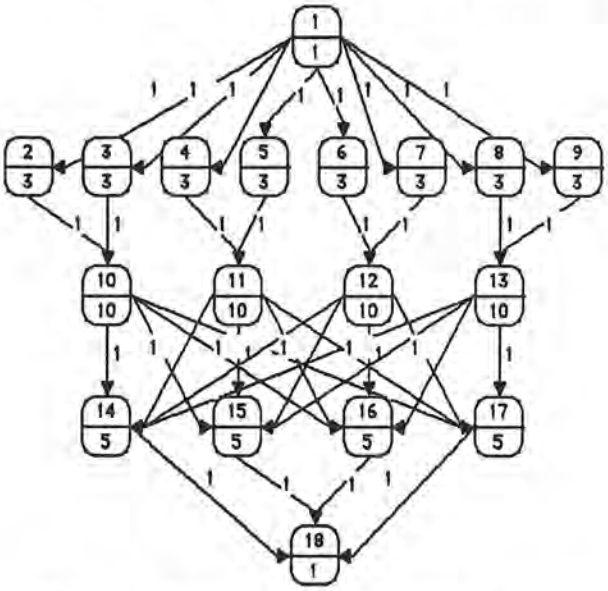

Figure 4. A task graph for the image processing example.

The Task Grapher uses these and the target machine description to apply (currently) seven different scheduling and mapping heuristics. These heuristics include a simple level approach [Hu61] and six new heuristics developed at OSU [ElLe89], [El89], [Kru87]. The latter techniques take communications delays between processors into account. One of the outputs from the scheduling and mapping tools is a set of Gantt charts that indicate the which processes are to be run on which processors at what times. Figure 5 shows a Gantt chart for the image processing application.

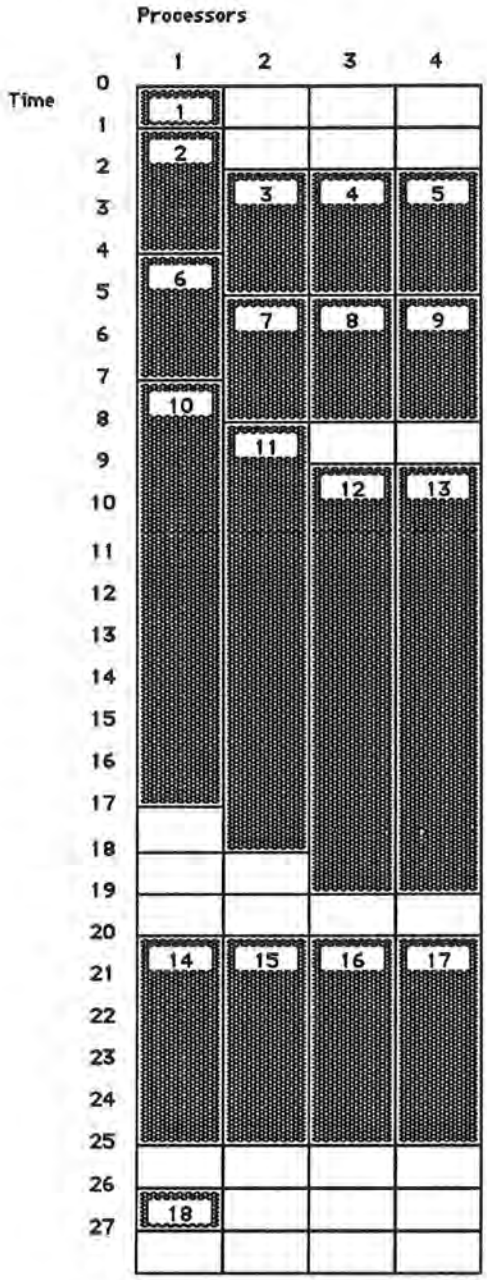

Figure 5. A Gantt chart schedule for the image processing example.

These schedules are themselves performance estimates, and we have several tools that can be used at this stage to analyze the simulated performance further. One can also use the editor that is built in to the Task Grapher to alter the original task graph, and the Task Grapher also functions as a stand-alone design tool [Fort89].

The Task Grapher produces as outputs bar charts showing processor utilization and efficiency, a animated simulation of the execution of the program, speedup curves, and other data. In Figure 6, we show a speedup curve for the image processing example.

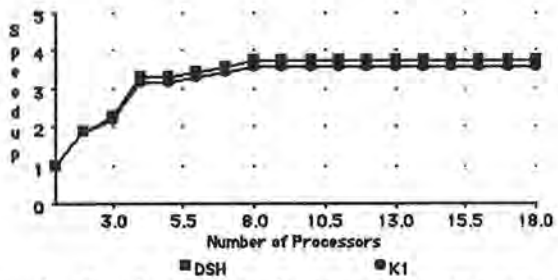

Figure 6. Speedup curve for the image processing example for two different scheduling heuristics.

If real code is to be developed, the Superglue tool [Hand90] takes source code from the code fragments, dataflow information from the PP Design File, the target machine description, and a Gantt chart schedule and creates source code ready for compilation. In other words, it is at this stage that we make the final binding to a programming language and parallel computing control method. Superglue can automatically insert timing function calls into the code so that real performance data can be recorded for subsequent analysis. An option that we have not yet implemented is to have Superglue insert calls to animation and debugging routines.

4. Progress to Date

We are currently in the early prototype stage in this project. While the system has gaps in it that must be manually overcome, the tools are not yet well-integrated, and we cannot yet deal with a diverse set of target languages and systems, we have nevertheless had enough experience with it to have proved the concepts involved.

The tools now run on large Macintosh computers with a Hypercard top level for control, file management, and help. To date, Superglue generates code only for parallel computing systems that use C-Linda for control of and communication between parallel processes. We have run codes generated by the system and collected performance data Sequent Balance and Symmetry and Intel iPSC2 computer systems to date.

Our experience with several applications shows that the tools described thus far really are powerful aids in the development of parallel program systems. While we have not tested their use in a course on parallel programming, we suspect that they will be fine teaching aids, as well.

In the development of the image processing application, we found that, while the lack of integration between the components is a definite hindrance, the tools really did help us develop a code that performs at levels that would have been difficult to achieve without the system. The Fortran code fragments were written by hand in this experiment.

In the process, we discovered that a valuable product of the Task Grapher is that it can indicate possible paths that are not worth taking early in the design process. For example, one might conclude from the task graph in Figure 4 that an 18-processor system like that shown in Figure 3 might improve performance on this problem. But the speedup curve in Figure 6 indicates that there is not much use in putting more than 4 processors to work on this computation.

We have also developed small programs that have improved significantly in performance after each of several cycles around the design and implementation loop. We have explored several cases in which actual performance measurements were fed back into the Task Grapher, with the result that improved schedules resulted in further improvements in performance.

One interesting fact that we have observed in monitoring the use of the PPSE as it stands is that there are at least two different top-level approaches to parallel program design. In one approach, for which our current ELGDF is well suited, the user emphasizes the functions to be performed and their decomposition. In this approach, the user distributes processes onto processors, and then partitions the data to fit onto the distributed processes. In the other approach we have observed, the user begins by partitioning the data in some logical way that fits with the algorithm, and then, in a sense distributes processes to carry out the computations onto the data partitions. The scheduling and mapping tools then map the processes onto processors. Our current ELGDF language is not well suited to the latter approach.

In this paper we emphasize the architecture of the PPSE itself. Some of our co-workers in this large team project are constructing new components that have been or soon will be added to the environment. Harrison and Gifford [HaGi89] have developed a prototype of a system that automatically converts Fortran source code into a Prolog fact base that is then converted into an ELGDF equivalent of the program. Others in the project, including Bella Bose and his students [AlBo88, 89a,89b,89c], have developed algorithms for the efficient partitioning of hypercubes into subcubes, while still others, including Virginia Lo and Sanjay Rajopadhye and their students [LoRa89a, b, LRGK90], have developed powerful techniques for describing and analyzing problems with high levels of regularity to run on regular architectures.

5. Work in Progress

We are now beginning the second year of effort on this project. Our experience indicates that we should focus our attention on the following issues:

Data partitioning. As noted above, ELGDF has a process-oriented flavor to it, with essentially no explicit means for describing or restructuring data between processes. Since we do not bind to an architecture or language at the design stages, we need to implement means in ELGDF to allow the user to explicitly indicate where data partitioning or views change in the dataflow. We plan to do this by using the notion of constrained dataflow [Stee89], in which, if needed, one can specify logical input and output channels for a node, and can then apply a data mapping function to partition, project, merge, construct a new view, or do other similar transformations on data structures.

Formal definition of ELGDF. This graphical design notation was originally conceived as a back-of-the-envelop design aid, and thus the original intent was to allow the user to define details of the meanings of the objects of the language as she saw fit. As the project has evolved, it has become clear that we are coming close to compiling ELGDF. Therefore, we need to produce a more formal definition of the language features than was heretofore necessary.

Entry interface to the core of the system. As we indicated above, ELGDF is just one of a host of possible sources for high-level design specifications that our system could process. In order for our system to work with other design tools, we need to construct a well-defined general interface to our system, so that we can write filters that convert output from other tools into a form that our system can process. We see no reason to go farther in this regard than to extend our current internal dataflow notation to a simple more general guarded command notation.

Scheduling loops and branches. Our scheduling and mapping tools produce static schedules that can be converted into source code to be compiled. This means that, although ELGDF diagrams (and real programs!) include conditional branches and loops that cannot really be scheduled statically, our system cannot presently handle these dynamic control structures. Therefore, for now all loops and branches must be hidden in low-level nodes; we cannot now handle branches and loops in the task graphs that our tools process. We plan to explore several ways in which to circumvent this current limitation of our system, including scheduling the longest critical path and using *a priori* estimates of probabilities of execution paths to try to generate reasonable schedules.

Enhanced Superglue. Superglue now produces C-Linda code, and we soon will be able to generate message-passing code for NCUBE hypercube systems. Immediate plans are to extend Superglue to produce code for the Cogent parallel computer and to generate Strand code, which will extend the utility of our tools to include several other kinds of parallel computing systems beyond those we can now support. We also plan to incorporate the SCHEDULE package [DoSo87] into the system soon.

Test drives. A difficult practical aspect in research in tools for the development of large-scale software systems is that it is expensive and time-consuming to test one's ideas, for, in order to do so, one must use the tools, ideally in a controlled experimental setting, to develop significant software packages. While we cannot afford to do real experiments, we do have several projects on the drawing board in which experts from other domains will work with software engineers to create parallel software systems for applications in those domains. We plan such projects in genetic sequencing, factoring large integers, and finite element modelling.

6. Conclusions

We have embarked on a significant experiment in parallel software engineering in which we are developing an integrated framework into which tools can be inserted to aid in the process all the way from initial top level design to performance measurement and for additional iterations through this process. Our preliminary results using the tools indicate that this approach is a sound one. We suggest that, until we have a much firmer idea of how the entire process might be more fully automated, perhaps through the development of "genius" compilers, ours and similar systems will be the preferred environments for the foreseeable future.

The PPSE is available for research use through the authors. Also, we would like to establish collaborative relationships with other groups that have tools available that might fit into our framework.

While the system is designed to cover a wide range of architectures and languages, the techniques and the system can be specialized easily to aid in developing software for single architectures and languages. We encourage computer manufacturers and software developers to consider the PPSE as a way to make parallel programming easier for their customers.

7. Acknowledgments

We are particularly grateful to Sequent Computer Systems and to Apple Computer, Inc., for their support of this work.

References

[AlBo88] A. Al-Dhelaan and B. Bose, "Incomplete cube-connected cycles," Proc., Canadian Conference on Electrical and Computer Engineering, pp. 573-576, Nov. 1988.

[AlBo89a] A. Al-Dhelaan and B. Bose, "Efficient fault tolerant broadcasting algorithms for hypercube," Proc., Fourth Conference on HypercubeConcurrent Computers and Applications, March 1989.

[AlBo89b] A. Al-Dhelaan and B. Bose, "New strategy for processors allocation in an N-Cube multiprocessors," Proc., Int. Phoenix Conference on Computers and Communications, pp. 114-118, March 1989.

[AlBo89c] A. Al-Dhelaan and B. Bose, "Efficient fault tolerant broadcasting algorithm for the cube-connected cycles network," Proc., The IEEEPacific Rim Conference on Communications, Computers, and Signal Processing, June 1989.

[BASo89] Brown, J. C., M. Azam, and S. Sobek, "CODE: A Unified Approach to Parallel Programming," IEEE *Software* , pp. 10-18, 1989.

[BaDi87] Babb, R. and D. DiNucci, "Design and Implementation of Parallel Programs with Large-grain Dataflow," in *Characteristics of Parallel Algorithms,* Jamieson, Leah H., Dennis B. Gannon, and Robert J. Douglass, eds., MIT Press, Cambridge, MA, pp. 335-349, 1987.

[DoSo87] Dongarra, J. and D. Sorensen, "SCHEDULE: A Tool for Developing and Analyzing Parallel Fortran Programs," in Characteristics *of Parallel Algorithms,* Jamieson, Leah H., Dennis B. Gannon, and Robert J. Douglass, eds., MIT Press, Cambridge, MA, pp. 363-394, 1987.

[ElRe89] El-Rewini, H. "Task Partitioning and Scheduling on Arbitrary Parallel Processing Systems." unpublished PhD dissertation, Dept. of Computer Science, Oregon State University, 1989.

[ElLe88] El-Rewini, H. and T. G. Lewis, "Software Development in Parallax: The ELGDF Language," Technical Report (88-60-17), Dept. of Computer Science, Oregon State University, 1988.

[ElLe89] El-Rewini, H. and T. G. Lewis, "Static Mapping of Task Graphs with Communications onto Arbitrary Target Machines - Case Study: Hypercube," Proc BISYCP '89, China, 1989.

[Fort89] Fortner, P. "MacSchedule: Tool for Scheduling Parallel Tasks," unpublished Master's thesis, Dept. of Computer Science, Oregon State University, 1989.

[Hand90] Handley, S. "Superglue: Tool for Automatic Code Generation," unpublished Master's thesis, Dept. of Computer Science, Oregon State University, (in preparation).

[Hu61] Hu, T. "Parallel Sequencing and Assembly Line Problems," *Operations Research* **9**, pp 841-848, 1961.

[JuRu89] Judge, D. and W. G. Rudd, "A Test Case for the Parallel Programming Support Environment: Parallelizing the Analysis of Satellite Imagery Data," Technical Report (89-80-2), Dept. of Computer Science, Oregon State University, 1989.

[Kim89] Kim, I. "Parallax: An Implementation of ELGDF," unpublished Master's thesis, Dept. of Computer Science, Oregon State University, 1989.

[Krua87] Kruatrachue, B. "Static Task Scheduling and Grain Packing in Parallel Processing Systems," unpublished PhD dissertation, Dept. of Computer Science, Oregon State University, 1987

[Lewi89] Lewis, T. G., "Parallel Programming Support Environment Research, Technical Report TR Lewis 89-1, Oregon Advanced Computing Institute, 1989.

[LoRa89a] Lo, V. M. and S. Rajopadhye, "LaRCS: Language for Regular Communication Structures", unpublished manuscript, 1989.

[LoRa89b] Lo, V. M. and S. Rajopadhye, "Mapping Distributed Divide-and-Conquer Algorithms onto Parallel Architectures", unpublished manuscript, 1989.

[LRGK90] Lo, V. M., S. Rajopadhye, S. Gupta, D. Keldsen, M. Moataz, and J. Telle, "OREGAMI: Software Tools for Mapping Parallel Algorithms to Parallel Architectures", unpublished manuscript, 1990.

[RLEJ89] Rudd, W. G., T. G. Lewis, H. El-Rewini, D. Judge, S. Handley, and I. Kim, "Status Report: Parallel Programming Support Environment Research at Oregon State University," Technical Report (89-80-1), Dept. of Computer Science, Oregon State University, 1989.

[SBNe82] Sieworek, Daniel P., C. Gordon Bell, and Allen Newell, *Computer Structures: Principles and Examples,* McGraw-Hill, New, York, NY, pp 18-22, 1982.

[Stee89] Steer, K., private communication