PVMbuilder - A Tool for Parallel Programming

Jan B. Pedersen and Alan Wagner

University of British Columbia, Vancouver
matt@cs.ubc.ca, wagner@cs.ubc.ca

Abstract. Message-passing is often used to implement parallel programs to run on workstation clusters. However, writing message-passing programs is a difficult and error prone task. In this paper we describe a graphical interface called PVMbuilder to support the construction of PVM programs. We show the tool produces little overhead in comparison to the hand-written PVM program. PVMbuilder provides a higher level abstract view of the program and supports dynamic process creation along with automatic generation of PVM communication calls.

1 Introduction

Clusters of workstations are a widely available source of computing power. Programs for these systems are typically written in C or Fortran and use libraries such as PVM [3] and MPI [2] to communicate and coordinate the computation. However, it is difficult to reason about processes all executing at the same time, sending and receiving data according to some, often complicated, pattern. Specifying the communication becomes a major difficulty. Communication errors can lead to unpredictable program behaviour such as deadlock, process failure, or incorrect results. Given the inherent problems in debugging message-passing programs, it is important to develop tools that avoid the problem altogether by assisting the programmer to construct correct code in the first place.

PVMbuilder was created to help with the construction of message-passing programs. Our main motivation was to create a tool to assist non-computer scientists to take advantage of cluster computing. The tool supports the following:

1. Automatic generation of communication calls to reduce errors.
2. A high level graph structure that provides a abstract representation of the overall structure of the program.
3. Generation of source code directly from the graph which compiles into C executables, with execution times close to hand-written programs.

PVMbuilder differs from other graphical program builders such as HeNCE [5] and CODE [7] in that it allows explicit message passing thereby supporting a superset of the programs representable in CODE or HeNCE. In addition, PVMbuilder appears to add a problem dependent constant to the runtime of the hand-written code. This constant varies between 6% and 13%.

Graphs are often used by programmers as an aid in constructing a message-passing program, where nodes are used to represent processes (or processors) and

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 108–112, 1999.

arcs denote communication or control flow. A natural extension to this process is to support the creation of these graphs and from there support the automatic generation of the underlying message-passing program.

As a result the programmer no longer has to manually transcribe the program and communication patterns into detailed source code instructions, a tedious and error-prone task. The graph also provides a higher-level specification of the program which can be used for debugging and program evolution.

2 PVMbuilder

A PVMbuilder program is built by constructing a directed acyclic graph that reflects the high level control flow of the program and the communication between the various processes. The graph consists of several different node types, each representing a different task. The resulting graph is then translated into C-code with PVM communication calls. The programs can then be compiled and executed via PVM on a cluster of workstations. Figure 1 shows an annotated example of a PVMbuilder graph which uses the seven different node types that can appear in the graph. The algorithm shown in Fig. 1 uses a master/slave pattern to solve a hyperbolic differential equation. The slaves are organized into a linear array of processes with left and right neighbours (except for the endpoints). The master distributes a one dimensional array to the slaves who compute and communicate the results to their two neighbours within a loop. The master program is shown on the left in Fig. 1 while the slave program appears on the right.
The following node types are defined:

A BEGIN node marks the beginning of a new process while an END node marks the end of the process. A CODE node contains sequential program code. There are three CODE nodes in Fig. 1. There is one in the master, `Init`, used to initialize variables, and get command line arguments. The slaves have two: `Init`, which again declares and initializes variables, and `DoCalculations` to perform the computation.

The SPAWN node is used to create new process groups. The number of new processes spawned is determined by a function bound to the SPAWN arc. In Fig. 1 there is only a single group of slave processes all executing the same program. As a result, the master has a single SPAWN node with only one SPAWN arc.

The COMM node denotes communication between two processes. COMM nodes are connected by a COMM arc that describes exactly what data is sent. In the example shown in Fig. 1 we find four instances of communication.

LOOPSTART and LOOPEND nodes denote loops containing communication. Since the slaves in the example execute a number of steps, each containing communication and computation, a set of LOOPSTART and LOOPEND nodes are declared. In Fig. 1 the loop condition is shown above the LOOPSTART node.
Figure 1 shows the three different kinds of arcs in a PVMbuilder graph.
PROGRAM arcs connect nodes, thereby describing the control flow.
SPAWN arcs are used to connect SPAWN nodes with the BEGIN nodes of the slave groups they create. In Fig. 1 the function which determines the number

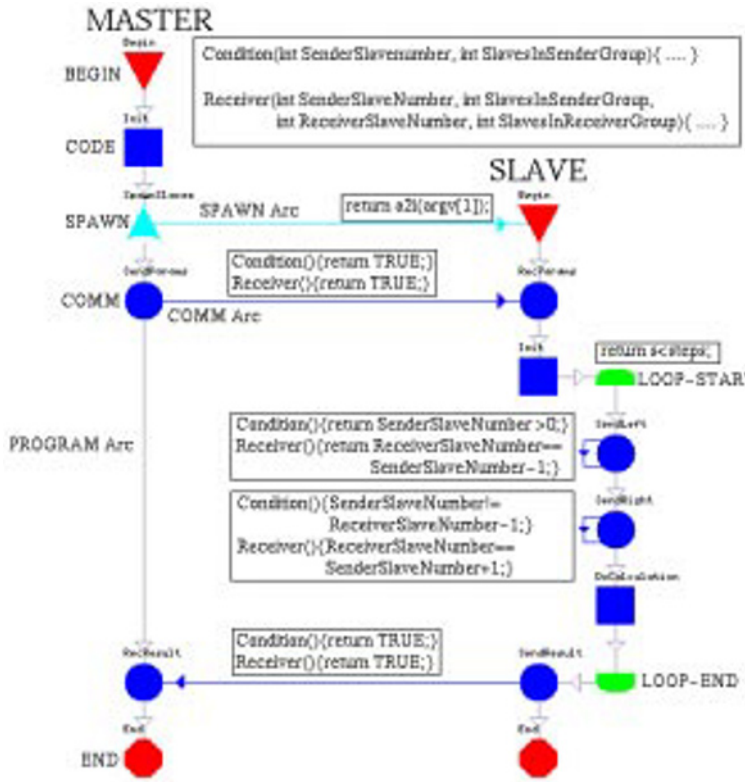

Fig. 1. PVMbuilder graph for the wave equation.

of slave processes to spawn is shown above the SPAWN arc. The COMM arcs connect COMM nodes and denote communication.

It is still necessary to bind program specific information to the appropriate nodes or arcs of a graph. Types, global variables, and functions are bound to BEGIN nodes. Sequential program code is bound to CODE nodes and the loop conditions are bound to LOOPSTART nodes. Figure 1 shows the loop condition above the LOOPSTART node. A runtime evaluable function, which determines the number of slaves to be spawned, is bound to every SPAWN arc.

Bound to every COMM arc are a guard (Condition()) to determine if a send should occur and a predicate (Receiver()) evaluated on every possible sender/ receiver pair (COMM arcs in Fig.1 are annotated with these predicates). The data to be sent is described using our own data description language.

The sender and receiver part of the program must both be able to evaluate the two predicates bound to the COMM arc connecting them. The Receiver() predicate is used by the sender to determine if a message should be sent to a receiver, and by the receiver to determine if there is a message. This restriction ensures that all sends are matched with receives, thus preventing communication deadlocks. In [1] we prove that keeping the PVMbuilder graph acyclic is a sufficient condition to avoid several types of communication deadlocks.

The Condition() and Receiver() predicates must return the same result in both the sender and receiver process. Therefore, the predicates can only depend on their arguments and two globally defined variables (`mySlaveNumber` and `myGroupSize`). In Fig. 1 we have shown function prototypes for the two predicates. As previously mentioned, we first evaluate the Condition() predicate, and if satisfied, the Receiver() predicate is evaluated for each `(SenderSlaveNumber, RecieverSlaveNumber)` pair and a message sent whenever it is satisfied.

Once the graph has been constructed and program specific information bound to the nodes and the arcs, PVMbuilder can generate and compile source files. The source files are generated with respect to the control flow of the graph and the appropriate PVM communication calls are inserted. The programs are then compiled and can be executed either as a stand-alone PVM application or from within PVMbuilder. When compiled in PVMbuilder, compilation errors are matched to the line number in the CODE node that generated them.

PVMbuilder also includes a performance monitor that can be used to track and display idle time, computation time and communication time. In addition trace files can be generated. Once debugged and tuned, source code without monitoring calls can be generated.

3 Patterns

Many algorithms share the same underlying graph structure. This gives rise to the notion of program templates or patterns [6] where the same graph structure can be reused to solve different problems.

In PVMbuilder there are two types of patterns that can be reused, graph patterns and communication predicates. A graph pattern in PVMbuilder is simply a graph structure without the problem specific information bound to the nodes and arcs. Communication predicates implicitly define a communication topology (or pattern) which can often be reused. When a predefined communication pattern is chosen PVMbuilder simply inserts the appropriate pre-defined Condition() and Receiver() predicates. The example shown in Fig. 1 is an instance of a master-slave pattern with an underlying linear array communication topology among the slave processes.

4 Related Work

Two notable tools which influenced the design of PVMbuilder were CODE and HeNCE. However, unlike PVMbuilder, CODE and HeNCE are based on a data flow representation of the program and do not support explicit message-passing. Indeed, PVMbuilder was developed as an attempt to extend the data flow model to encompass message-passing. More recently, the authors of HeNCE and CODE have developed the VPE [8] system which also addresses the shortcomings of CODE and HeNCE by including message-passing.

PVMbuilder differs from the previous system in that communication is more abstract in the sense the user does not have to explicitly specify the send/receive

primitives. As a result it is possible to check for and avoid communication deadlocks.

PVMbuilder includes both the data flow as the communication between processes as well as the control flow within each process. Therefore, it differs from purely process-based tools such as TRAPPER [4] and the higher level abstraction tools such as Enterprise [9] which represents the program as a template or pattern that abstracts away all of the underlying communication.

5 Conclusion

We have presented a new graphical user interface, PVMbuilder, for developing message-passing programs. It is an integrated environment that includes tools for program creation, reuse, tracing and performance monitoring.

The key feature of PVMbuilder is that programs are expressed in a manner close to how humans think of them: as graphs dealing with both code and data flow. The communication model of PVMbuilder, i.e. the ability to specify what to send and receive at the same time, is a significant improvement over the explicit specification of every send and receive.

In addition, we have shown in [1] that PVMbuilder introduces a relatively low overhead that we believe can be further reduced. And, since PVMbuilder generates code and executables that can be run without the presence of the tool, the number of potential users of the tool, or programs generated from it, is large. Although PVMbuilder generates C code and PVM communication calls it can easily be targeted to other similar languages and communication libraries. For more details see http://www.cs.ubc.ca/~matt/pvmbuilder.html

References

[1] B. B. Blendstrup and J. B. Pedersen. PVMbuilder - et grafisk værktøj til parallel programmering. Master's thesis, Aarhus Universitet, Jan. 1997.
[2] J. Dongarra. MPI: A message passing interface standard. *The International Journal of Supercomputers and High Performance Computing*, 8:165–184, 1994.
[3] A. Geist et al. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Prentice Hall, 1994.
[4] F. Heinze et al. Trapper, eliminating performance bottlenecks in a parallel embedded application. *IEEE Concurrency*, pages 28–37, July-September 1997.
[5] J. Dongarra et al. *HeNCE: A Users' Guide. Version 2.0*, http://www.netlib.org/hence edition.
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[7] P. Newton and J. C. Browne. The CODE 2.0 graphical parallel programming language. *Proc. ACM Int. Conf. on Supercomputing*, July 1992.
[8] P. Newton and J. Dongerra. Overview of VPE: A visual environment for message-passing. 1994.
[9] A. Singh, J. Schaeffer, and M. Green. A template-based approach to the generation of distributed applications using a network of workstations. *IEEE Transactions on parallel and distributed systems*, 2(1):52–67, January 1991.