A Dataflow-based Methodology for
Coarse-Grain Multiprocessing on Distributed
Systems

J. Wrabetz, R. Jagannathan, and A. Downing
SRI International,
Menlo Park, California 94025

(Extended Abstract)

1 Introduction

Distributed systems applications such as database query processing, process
control, and multi-function processing have requirements for multiprocessing. Although distributed systems provide the necessary aggregate computing power and communications bandwidth to support multiprocessing, they are often under utilized because the necessary tools and methodologies to facilitate multiprocessing are not available.

We are developing an integrated methodology that allows users to describe and execute applications on distributed systems that achieves increased performance by implicitly exploiting the parallelism inherent in these applications. The integrated methodology for multiprocessing provides for the following.

1. a graph-based programming model for describing applications such that parallelism is implicit in the application description,
2. a computing model that exploits useful parallelism inherent in an application and
3. an execution system that takes advantage of the aggregate computing power of the distributed system architecture.

We briefly describe the three parts of the methodology below.
2 Programming Model

Dataflow programming models are appropriate for multiprocessing applications on distributed systems because they are based on mechanisms of interaction that can be efficiently implemented on such systems. In dataflow programming models, an application is described as a data-dependency graph whose vertices can be thought of as processes and whose edges can be viewed as communication channels between processes. Thus a data-dependency graph description of an application naturally defines an abstract architecture of a distributed system.

An application should be viewed as consisting of processes (denoting vertices) that communicate with each other via messages (denoting data items) that travel on the connecting communication medium (denoting edges). The ratio of inter-process communication delay to the processing speed in a distributed system dictates that the amount of time spent processing between inter-process communication should be large. In other words, the granularity of the vertex denoting the process should be coarse. We have developed a coarse-grained dataflow language that allows procedures within vertices of the graph to be described in an imperative language and sequentially executed [4].

3 Computing Model

We have adopted the tagged lazy-dataflow computing model as the basis of our implementation [1]. Unlike the tagged eager-dataflow computing model which has formed the basis of several dataflow implementations [1], this model avoids any superfluous computing. This is particularly important in our context because the vertices may cause side-effects which are undesirable when the associated evaluation is superfluous.

We assume that the computing model is implemented exclusively using demand-driven execution (or eduction) [3]. It is possible to intersperse data-driven execution which would reduce the overhead associated with propagation of demands [1]. In the demand-driven implementation of the tagged lazy-dataflow model, only the output of a program instance that is needed is demanded. This causes further demands to be propagated – demand for an input data item (from the external world) is satisfied when the data item is available. Execution of a vertex which is suspended pending outstanding demands for required data items is resumed when these data items become
available. By using the tagged lazy-dataflow computing model to evaluate the graph, both static parallelism and dynamic parallelism are implicitly exploited. Static parallelism is parallelism inherent in the structure of the data-dependency graph whereas dynamic parallelism is parallelism inherent in the concurrent evaluation of several instances of a given vertex.

4 Execution System

We have designed an execution system that implements the tagged lazy-dataflow computing model on a distributed system. The execution system implements the vertices of the dependency graph as processes and the interaction between vertices as messages. Because a distributed system lacks global state information and has no central point of control, it is inappropriate for the developers of applications to attempt to define allocation and scheduling of processes across the system. Thus, the execution system is designed to perform dynamic runtime allocation and scheduling of processes denoting vertices in an efficient manner.

![Diagram of the Execution System]

**Legend**
- SDM: Shared Data Manager
- SA: Site Allocator
- GM: Graph Manager
- SM: Session Manager
- VM: Vertex Manager

**Figure 1: Structure of the Execution System**

The execution system consists of five components that run on each participating site of the distributed system. The *vertex manager* propagates de-
mands, executes processes associated with vertices, and returns results. The vertex manager can manage multiple vertices simultaneously. The shared-data manager is responsible for managing shared data items, satisfying outstanding demands, and "garbage collection." A unique mapping from data items to a shared-data manager evenly distributes the processing and storage requirements associated with shared data. The site allocator which makes runtime decisions about the execution locations of vertices, is used for load balancing and for fulfilling special processing requirements. The session manager provides the user interface to aid the user in programming, debugging, and executing applications. The graph manager stores a static graph representation for shared use by the other components. Figure 1 shows the structure of the execution system.

5 Current Status

We have developed the compiler for a prototype graph-based description language that incorporates C code for tasks with an equational graph description language. The prototype execution system has been designed, and implementation will initially be based on a local-area network of Sun workstations running Sun UNIX.

References


