# DPOS : A Metalanguage and Programming Environment for Parallel Processors

Robert R. Kessler and John D. Evans

UUCS-90-019

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

October 9, 1990

## Abstract

*The complexity and diversity of parallel programming languages and computer architectures hinders programmers in developing programs and greatly limits program portability. All MIMD parallel programming systems, however, address common requirements for process creation, process management, and interprocess communication. This paper describes and illustrates a structured programming system (DPOS) and graphical programming environment for generating and debugging high-level MIND parallel programs. DPOS is a metalanguage for defining parallel program networks based on the common requirements of distributed parallel computing that is portable across languages, modular, and highly flexible. The system uses the concept of stratification to separate process network creation and the control of parallelism form computational work. Individual processes are defined within the process object layer as traditional single threaded programs without parallel language constructs. Process networks and communication are defined graphically within the system layer at a high level of abstraction as recursive graphs. Communication is facilitated in DPOS by extending message passing semantics in several ways to implement highly flexible message passing constructs. DPOS processes exchange messages through bi-directional channel objects using guarded, buffered, synchronous and asynchronous communication semantics. The DPOS environment also generates source code and provides a simulation system for graphical debugging and animation of the programs in graph form.*

# DPOS: A Metalanguage and Programming Environment for Parallel Processors

Robert R. Kessler and John D. Evans

Department of Computer Science
University of Utah
Salt Lake City, Utah

**Abstract** - The complexity and diversity of parallel programming languages and computer architectures hinders programmers in developing programs and greatly limits program portability. All MIMD parallel programming systems, however, address common requirements for process creation, process management, and interprocess communication. This paper describes and illustrates a structured programming system (DPOS) and graphical programming environment for generating and debugging high-level MIMD parallel programs. DPOS is a metalanguage for defining parallel program networks based on the common requirements of distributed parallel computing that is portable across languages, modular, and highly flexible. The system uses the concept of **stratification** to separate process network creation and the control of parallelism from computational work. Individual processes are defined within the **process object** layer as traditional single threaded programs without parallel language constructs. Process networks and communication are defined graphically within the **system** layer at a high level of abstraction as recursive graphs. Communication is facilitated in DPOS by extending message passing semantics in several ways to implement highly flexible message passing constructs. DPOS processes exchange messages through bi-directional **channel** objects using guarded, buffered, synchronous and asynchronous communication semantics. The DPOS environment also generates source code and provides a simulation system for graphical debugging and animation of the programs in graph form.

## 1 Introduction

Within the research area of distributed parallel computer systems the technology to develop software has not kept up with the advances in hardware development. In addition to the computational issues of imperative sequential programs, parallel programs must address issues that are strictly related to parallel programming models such as creation and management of parallelism, synchronization controls, etc. Programmers must also resolve architecture related problems. Machine specific primitives for process creation and synchronization must be incorporated into programming models and architectural issues such as process mapping and load balancing must be addressed. Also, the topologies of process networks are generally complex and irregular graph structures. Single threaded languages may successfully capture the topology of a single threaded program. Representing a process graph structure in single threaded languages, however, is often difficult and obscure. The ability to understand and define parallel programs and the portability of programs that are machine specific and model specific at a low level is severely compromised.

All MIMD parallel programming systems, however, address common general requirements for process creation and management and interprocess communication. Also, due to the impact of distributed parallel computer architectures on algorithm design, distributed parallel algorithm structures are often groupable into common structure types.

We propose the following solution to some of these problems of parallel computing:

1. The separation of parallel program structure and synchronization from strictly computational (in the sequential programming sense) issues.
2. The representation of the parallel program structure and synchronization in a form that is language independent, that supports common parallel program structures, and that accurately and understandably presents program topology.
3. A method of communication that does not require programmers to resolve low level synchronization problems.
4. Allow programmers to develop processes as purely sequential blocks of code that can be developed and debugged as encapsulated units.
5. Individual processes and clusters of processes are encapsulated. This supports the modular reuse of clusters. Also, parallel program development is 'evolutionary' in the sense that programs gradually evolve from individual sequential blocks to large networks by combining processes and clusters of processes.

CSP[3] is a basis model for many distributed parallel programming languages and shares some features with DPOS. It uses common sequential language constructs and incorporates several parallel features. Many distributed systems use a subset of the communication features of CSP and a similar notion of sequential processes. In CSP processes are statically allocated. CSP communication is direct via channels (a specific receiver and sender are specified for each channel). CSP requires special language constructs to implement message-passing semantics within individual processes. CSP specifies a single message-passing semantics (synchronous guarded). Unlike CSP, most distributed parallel programming systems and languages allow some form of dynamic process creation. In most systems, process creation is similar to function calling. Most systems use direct communication which means that senders indicate a specific receiver. Most systems use a single message-passing semantics and most require programmers to use new language constructs.

DPOS differs in several ways from CSP and most other distributed parallel programming systems:

1. DPOS allows dynamic process allocation. DPOS process creation is similar to abstract data type definition rather than function calling. DPOS process subnetworks are defined as graph structured units.
2. DPOS message-passing semantics are an attribute of the communication channel and not a language construct.
3. DPOS channels allow indirect communication. This means that multiple sender and receiver processes may use the same channel.
4. DPOS incorporates several commonly used message-passing protocols. Also, because DPOS semantics are encapsulated within channel objects, multiple communication protocols may coexist within the same program.

The ramifications of these differences are discussed in the following sections.

## 2 DPOS

DPOS[2] brings together the concepts of object-oriented programming, graphical programming, and aspects of modern functional languages. A DPOS program is defined as a network of active processes called Process Objects (POs) and communication lines called Channels that are grouped into subnetworks called Network Modules (NMs). In DPOS a communicating process network model is used.

## 2.1 Process Objects

Process Objects (PO) are single threaded program functions with calling parameters identical to traditional sequential program functions. These active objects employ much of the modularity, encapsulation of function, and encapsulation of data found in sequential object-oriented programming. Sequential objects and many object-like parallel systems terminate execution between receipt of messages. These systems must save state information before termination and test state information upon receipt of every message. DPOS process objects do not terminate execution between receipt of messages. Instead, process objects block while attempting to receive messages from channels leaving the runtime stack intact. The state of computation for the object is defined by the runtime stack as in non object-oriented programming. Because of this, the need for state variables is reduced and consequentially the amount and complexity of code that the programmer must write to explicitly maintain state variables is reduced. DPOS objects enjoy the encapsulation of functionality and data of sequential object oriented programming without the cost in terms of explicit state variable maintanance.

The connection links (Channels) between Process Objects appear as variables passed in as calling parameters similar to files or streams in traditional programming. Process Objects communicate with each other via channel accessor functions like **send** or **receive**. No additional syntax or language extensions are required, since simple function call syntax is used. The control flow of Process Objects is internal. The progress of computation, however, may be controlled by regulating message traffic into and out of the Process Object causing the PO to block waiting for communication to proceed. The synchronization required for communication is controlled by the communication channel. Because of this a PO may follow a bounded sequential computation or may be an unbounded cyclical computation (like an operating system process) that is I/O driven via its communication channels.

The creation of Process Objects is specified within the parent Network Module. The termination of process objects occurs when the execution of the code segment for the process object terminates. The rules governing the specification and creation of process objects are similar to those for network modules (see Subsection 2.3).

The sequential nature of Process Objects allows them to be developed and debugged individually as separate programs before integration into a network module. Simple terminal input and output is substituted for channel communication during sequential debugging. A library of channel definitions in the base language has been developed for sequential debugging.

Global data structures are not defined within the DPOS model. The use of channel objects to implement data that is shared between many processes is discussed below (see subsections 2.2 and 5.2).

## 2.2 Channels

Communication and synchronization between Process Objects is accomplished by message-passing. The concept of message-passing is not new. Simulation systems have long used message queues for interactions between concurrent processes in simulated parallelism. In true distributed parallel environments like OCCAM[7,5] and CSP channels are the primary mode of communication. In these and most other channel message-passing systems, the channel represents a simple communication relationship between a sender and a receiver process. In these systems communication is tightly synchronized. This means that both sender and receiver must block while the exchange of data is made. A process may select (guard) the type of message it receives by means of a language construct that nondeterministically chooses from available incoming messages.

In DPOS a channel is treated as a separate object in the object-oriented sense and not just as a communication relationship. This allows the semantics of channels to be extended in several ways. The semantics of channel communication is an attribute of the channel. DPOS channels encapsulate both functional semantics and in some cases data storage. This encapsulation has several advantages:

1. It allows channels to be accessed by multiple sender and receiver processes. The arbitration for access is handled within the channel object. This considerably reduces both the number of channels required by a program and the complexity of managing channels. For example, the merging and splitting streams of data is trivially implemented using shared channels. Multiple process access to channels allows channels to be used to implement shared data values. This usage is shown in example 5.2.

2. It removes the need for language constructs to implement communication semantics. Because of this, the PO definitions of a DPOS program need no extensions beyond traditional single threaded programming constructs.

3. It allows bi-directional communication. Allowing bi-directional communication may significantly reduce the number of channels required by a program as illustrated in subsection 5.1.

4. Communication is indirect. In direct communication systems the sender names a single receiver or link dedicated to the receiver explicitly and possibly visa versa. This form of communication is adequate for parent child communication but it hampers general dynamic process creation because it requires that senders and receivers be notified whenever a new potential receiver (or sender) process is created. Many process network programs require siblings or cousins to communicate and require substantial propogation code to be added to programs. In DPOS, many processes may use the same channels so dynamic process creation is unhampered. New processes may be added that use existing channels without notification (see Example 5.2).

5. It allows multiple types of channels. In DPOS the type of a channel specifies the semantics of communication for that particular channel. Supporting multiple channel types allows greater programmer flexibility.

In contrast most programming systems support only one communication protocol. The DPOS guarded input channel type is roughly equivalent of OCCAM's guarded communication system. Guarded input is appropriate for some programs, however, it requires strict synchronization of potential senders with the receiver and requires the additional overhead of guard resolution. In many programs another type of communication such as buffered communication is more appropriate, however, neither strict synchronization nor guard resolution is necessary for buffered communication. Many programs may appropriately use more than one communication protocol and they must be implemented with whatever semantics are available. Paying for the added protocols with the overhead of the existing channel types and the added complexity of implementing additional constructs.

DPOS channel types include: synchronized guarded input, synchronized guarded output, asynchronous, buffered, and synchronous.

### 2.3 Network Modules

Network Modules (NMs) are abstractions used for defining subnetworks of DPOS programs. Network Modules are composed of Process Objects, Channels and other nested Network Modules. Network Module types are defined as graph structures (see Figures 1, 2, 4, and 5). Network Module definitions have local environment and have formal parameters that correspond to data values and channel instances. Invoking an NM requires actual parameter arguments to be provided. The arguments are the actual access channels that connect to the NM as well as any required values computed within the scope of the invoking environment. Network Modules may be nested and recursively or mutually recursively defined. In the traditional object-oriented framework, a Network Module definition constitutes a class definition and may be instanced numerous times in the definition of a process network.

Global values are not defined within DPOS. The only exterior environment visible from within a Network Module consists of the actual arguments passed in at the instantiation of the NM. A Network Module then is completely encapsulated by the actual arguments and peripheral channels and may be analyzed as a unit. This modularity allows a Network Module to be developed and debugged as a unit by instantiating it and its peripheral channels and without creating the outlying program network.

A Network Module instance might not be instantiated when the parent NM is instantiated. The instantiation of the NM may be delayed until a demand is made for its creation. Delayed Network Modules are instantiated whenever data flow occurs in one of its peripheral channels. Alternatively the instantiation may be conditional in which case a constraint condition is evaluated within the parent NM environment to determine whether or not the instance is to be instantiated. Constraint conditions and instantiation delays are specified within the parent NM definition. These properties allow process networks to be specified in a manner similar to abstract data types such as trees and graphs in high-level languages. For example, a generic binary tree Network Module may be defined which is then used to define a specific, irregular extended tree process network (see Figure 2).

## 3 Portability

The DPOS metalanguage is portable across programming languages. The process networks defined using DPOS may be implemented in any target language supported. The currently supported target languages all have a common base language, sequential Scheme. Three target languages derived from Scheme are supported: Butterfly Scheme[8,4], Concurrent Utah Scheme[6], and DPOS Scheme. Butterfly Scheme is a shared memory language using 'locks' for synchronization and 'futures' for process creation. Concurrent Utah Scheme is a distributed parallel language using remote function calls to create processes and a specialized form of monitor for synchronization control. DPOS Scheme is a distributed memory language which uses remote process creation and DPOS channels for communication. DPOS Scheme was designed and implemented specifically to use DPOS semantics as a part of this project.

The individual process definitions are traditional sequential programs developed outside of DPOS. Individual process definitions are language dependent and are implemented in the base language. Process Objects using only Scheme constructs and DPOS function calls are portable across the three target languages supported.

The implementation of a DPOS derived base language such as DPOS Scheme requires the ability to implement explicit process creation and destruction, and the ability to implement simple message-passing. The implementation of DPOS constructs in an existing base language requires the ability to construct the same basic features in the base language.

Shared memory languages supporting semaphores, locks, monitors or similar synchronization and some form of process creation such as futures are generally adequate for this. Many distributed programming languages that are message oriented meet the implementation requirements.

## 4 DPOS Programming Environment

The design of the system layer of DPOS was intended to be implemented graphically from its inception and to encapsulate the parallel programming issues of network topology, synchronization control and dynamic process creation. The DPOS Programming Environment incorporates a graphical program editor/animator and simulator program. The environment provides the capability to define Network Modules, specify process object interfaces, to generate source code in any of the target languages, to simulate program execution and to interactively animate and debug DPOS programs using the simulation output.

The graphical editor is a window-oriented block diagram manipulating system. It also incorporates text editing for specification of NM and PO class names, instance names, formal parameters and arguments. Class definitions of Network Modules and Process Objects are defined by editing templates (see Figures 1 and 2). Instances of these classes may then be placed in other templates or added (recursively) into the original template. Blocks represent Network Module, Process Object and channel instances. External 'ports' on templates correspond to ports on the instance blocks. The accessibility of a channel instance from a Network Module or process object instance is represented by a connection line from a 'port' on the NM or PO instance to the channel instance.

## 5 Example Programs

The programs presented were selected to demonstrate features of DPOS and its application to common parallel program structures. The examples refer to Figures 1 thru 6. Figures 1, 2, 4, and 5 show Network Module templates. In the templates the square boxes represent channels with the enclosed letter indicating channel type. Black boxes represent nested Network Modules and white boxes represent Process Objects. Network Modules and Process Objects are labeled **type-name:instance-name**.

### 5.1 Fibonacci Numbers

This program presents the naive recursive algorithm for computing the fibonacci sequence. This example (see Figures 1, 2 and 3) is presented as a simple illustration of a recursive process network programmed with DPOS and not because it is the optimum way to solve this particular problem. The program includes two Process Object types, a controller process of type **f-start** and worker processes of type **fib-unit**. The entire Process Object source listing is given in Figure 3. The graphical Network Module definitions are shown in Figures 1 and 2.

Process objects use interface functions defined by the user. Process objects **f-start** and **fib-unit** use interface functions **f-start-po** and **fib-unit-po** respectively (see Figure 3). Interface files may contain an arbitrary amount of sequential code to support the semantics of the PO.

Figure 1 shows the top-level Network Module. The only argument to the **Fib-test** NM is the value of **size** which determines the fibonacci number to be computed. Figure 1 contains an **f-start** type Process Object called **f-start:st** and a **fib-tree** type Network Module called **fib-tree:tree**. **F-start:st** sends the seed **size** to its channel then reads the result when the computation is complete. The **fib-tree** Network Module is shown in Figure 2. It contains a **fib-unit** PO and two delayed **fib-tree** NMs. **Fib-unit** process objects receive a **size** value from their top channel and if **size** is less than **grainsize** they do the fib calculation for that **size**. If **size** is greater than **grainsize** they send **size**-1 and **size**-2 to their bottom channel which triggers the creation of the delayed **fib-tree** Network Modules who do the work. The child processes then carry out the subcomputations and return the results.

This example demonstrates three DPOS features: The dynamic creation of a process network structure using recursive Network Modules and delayed instantiation (Figure 2). The removal of parallel control mechanisms from the programmer defined source code of process objects (Figure 3). This example also demonstrates the use of guarded input channels for bi-directional communication. The **read-guard** operation specifies the channel and a list of acceptable message types. The **write-guard** operation specifies the channel, message type and data. A received guard message is of the form **(type data)**. The use of guard channels ensure that only appropriate message types are received. If guard channels are not used then a separate input and output channel would be required for both top and bottom connections to ensure that high-level race conditions do not occur. Message types used are **ask** and **reply**.

## 5.2 Matrix Multiplication

This example program multiplies an L x M matrix (A) by a M x N matrix (B). The problem is typical of many numerical computations and other problems with very regular structure. The solution represents the common parallel programming strategy of using a pool of servant processes to carry out similar computations.

Network Modules for the program are shown in Figures 4 and 5. In the program a control process of type **mmcontrol** first sends the B matrix to a channel labeled "B matrix" where it is read each of the **row-mm** type servant processes defined in **row-**

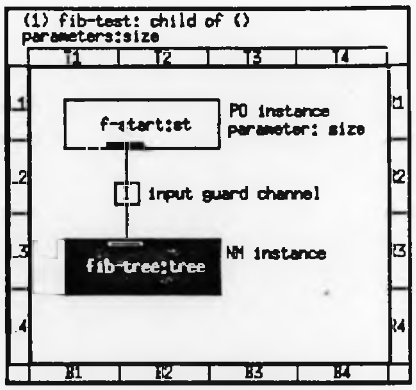

Figure 1: Fib-test Network Module

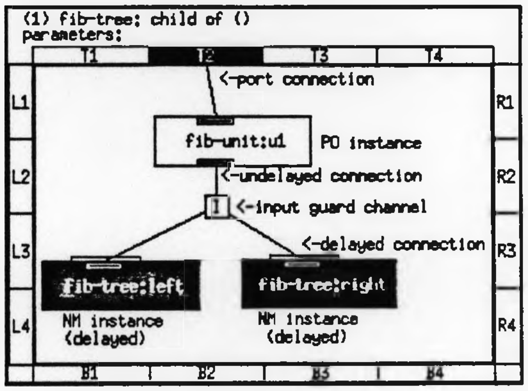

Figure 2: Fib-tree Network Module

```
(define (f-start-po CHAN size)
 (write-guard CHAN ask size)
 (display
   (list 'answer (read-guard CHAN (list reply)))))

(define (fib n)
  (if (< n 3)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))

(define (fib-unit-po TOP BOT)
 (let ((size (cadr (read-guard TOP (list ask))))
       (grainsize 7))
  (if (< size grainsize)
   (write-guard TOP reply (fib size))
   (begin
    (write-guard BOT ask (- size 1))
    (write-guard BOT ask (- size 2))
    (let ((res1 (cadr (read-guard BOT (list reply))))
          (res2 (cadr (read-guard BOT (list reply)))))
     (write-guard TOP REPLY (+ res1 res2)))))))
```

Figure 3: Fibonacci Process Objects

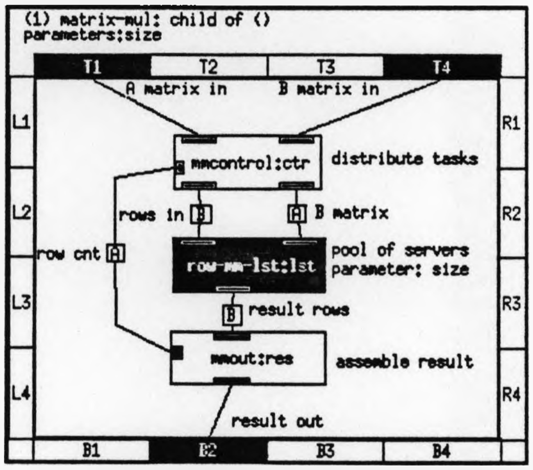

Figure 4: Matrix Multiplication Network Module

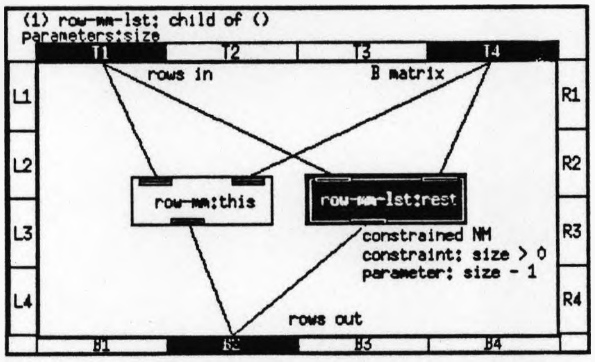

Figure 5: Server Process List Network Module

```
(define (row-mm-po ROWS-IN B-MATRIX RESULT-OUT)
 (let ((matb (chan-read B-MATRIX)))
  (do ((row (receive ROWS-IN) (receive ROWS-IN)))
      ((not row))
   (let ((index (car row))
         (x (cadr row))
         (y (caddr row))
         (rowvec (cadr (cddr row))))
    (send RESULT-OUT
          (list index (row-x-mat x y rowvec matb)))))))

(define
  (mmcontrol-po
    A-MAT-IN B-MAT-IN ROWS-IN B-MATRIX ROW-CNT cnt)
  (let* ((matb (receive B-MAT-IN))
         (mata (receive A-MAT-IN))
         (x (vector-length mata))
         (y (vector-length matb))
         (z (vector-length (vector-ref matb 0))))
    (send B-MATRIX matb))
    (send ROW-CNT x)
    (do ((a 0 (+ a 1)))   ;send out the rows
        ((= a x))
      (send ROWS-IN (list a y z (vector-ref mata a))))
    (do ((a 0 (+ a 1)))   ;send out terminations
        ((= a cnt))
      (send ROWS-IN #f))))
```

Figure 6: Process Object Code Network Module

**mm-lst:lst**. It then distributes rows of matrix A to the servants on a first-come first-served basis through channel "rows in". Computed rows are sent to a collector process **mmout:res** which assembles the resulting matrix. Process Object **mmout** receives the number of rows from **mmcontrol:ctr** via channel "row cnt" to determine when it has received all resultant rows.

The list of servant processes is defined as a recursive Network Module in Figure 5. The individual servant processes are of type **row-mm** and the source code is shown in Figure 6. The number of servants is determined by defining a constraint condition on the nested recursive NM instance **row-mm-lst:rest**. The servant processes all share common access to the channels in the **matrix-mul** Network Module.

The program maintains a balanced workload by taking advantage of shared channel "rows in" to allow the nondeterministic distribution of row vectors to servant processes. The channels marked "B" are buffered to relax the synchronization between processes as much as possible. The program uses asynchronous channel "B matrix" which buffers a single message and allows input operations **receive** with removal or **chan-read** without removal of the channel contents (see Figure 3). Using **chan-read** in this program implements a read only shared variable for the servant processes.

## 6 Debugging and Animation

Network Module definitions may optionally be generated with debugging trace information. DPOS Scheme is implemented in simulated form. The DPOS Scheme simulator optionally executes debugging source code and produces a trace file of program execution. The execution may then be animated using the graphical interface.

DPOS allows a wide range of debugging information to be monitored. The amount and complexity of debugging trace information may be formidable.

The DPOS interface allows the programmer to selectively monitor aspects of the program execution and exclude others. The programmer may open template, parameter and trace windows corresponding to Network Module, channel and Process Object instances and monitor information that pertains only to the selected object.

The graphical animation of Network Module templates shows the creation and termination of processes and Network Modules, message-passing through channels. Information of this type is represented by coloring and marking connection lines, PO, NM and channel blocks. This information is generally adequate to locate deadlocks, infinite process recursions and high-level race conditions.

More detailed information is available through parameter windows as text menus indicating arguments supplied to instantiated process objects and Network Modules. Trace windows display the trace output streams for the object being monitored. Trace window output includes status information such as checkpoints, error messages and i/o generated by processes, and channel status information such as buffer contents and blocked readers and writers.

The user optionally selects to allow free run of the animation, to single step, or to step one process (animate until the next event that affects that process). The user also specifies the speed of the animation.

## 7 Performance Measurements

Preliminary performance measurements are encouraging particularly for more complex program types such as Split Merge Sorting [1] and Branch and Bound Search that require shared variable implementation or complex intercommunication. Further testing is being carried out to test the limits of the programming system on more complex programs. Performance measurements have been taken using Butterfly Scheme on the BBN Butterly and using distributed CUS in a distributed workstation environment. Performance measurements reflect programs written using DPOS in the base languages.

**Performance Measurements**

| Time(sec) | Processors | Speedup | Efficiency |
|---|---|---|---|
| **Prime Number Sieve** | | | |
| 155.35 | 1 | - | 1.0 |
| 13.65 | 12 | 11.38 | 0.948 |
| **Matrix Multiplication** | | | |
| 196.000 | 1 | - | 1.00 |
| 62.604 | 4 | 3.118 | 0.78 |
| 36.400 | 8 | 5.385 | 0.67 |
| **Branch and Bound Search** | | | |
| 386.64 | 1 | - | 1.00 |
| 141.34 | 3 | 2.73 | 0.91 |
| 85.38 | 5 | 4.53 | 0.90 |
| **Split Merge Sorting** | | | |
| 58.33 | 1 | - | 1.00 |
| 19.10 | 4 | 3.05 | 0.76 |

## 8 Conclusions and Further Work

DPOS provides a high-level metalanguage and programming/debugging environment. DPOS defines process networks and communication based on fundamental properties of parallel computer systems at a high level of abstraction that is flexible and portable across a variety of architectures and existing parallel languages. The system allows the incremental construction of programs, with a minimum requirement for low-level parallel programming. The DPOS interface provides an integrated set of tools for defining visualizing and debugging that greatly reduces the need for low-level parallel programming and that assists programmers in resolving parallel programming problems with deadlock, high-level race conditions and recursion problems. The interface also helps organize and selectively access sequential debugging information on a module by module basis. Programs defined using DPOS have been shown to execute at high levels of performance.

We feel that DPOS offers significant possibilities in several other areas: 1) Integration of process mapping and load balancing criteria into DPOS. 2) Integration of parallel performance monitoring with the DPOS environment. 3) Extension of DPOS to other base languages. 4) Extension of DPOS programs to include multiple languages given suitable data type coercion. 5) Inclusion of high-level control flow at the process object level. 6) Development of large applications to test the system limits.

Our currently limited experience shows that DPOS works well when experienced sequential programmers try to write parallel programs although it takes time to learn the parallel system semantics and style. We will be teaching a parallel programming class that uses DPOS to gain more insignt into its utility.

## References

[1] Akl, S. G. *Parallel Sorting Algorithms.* Academic Press, 1985.

[2] Evans, J. D. *A Distributed Object-Oriented Graphical Programming System.* Master's thesis, University of Utah, Salt Lake City, Utah, 1990.

[3] Hoare, C. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[4] Jonathan Rees, William Clinger, e. a. Revised report on the algorithmic language scheme. Tech. Rep., Massachusets Institute of Technology, 1984.

[5] Kerridge, J. *Occam Programming: A Practical Approach.* Blackwell Scientific Publications, 1987.

[6] Kessler, R. R., and Swanson, M. *Concurrent Scheme.* Springer-Verlag, 1990.

[7] INMOS Limited. *Occam Programming Manual.* 1984.

[8] BBN Advanced Computers Inc. *Butterfly Scheme Reference.* 1988.