# A Multiprocessor Implementation of Joyce[*]

(1989)

**Joyce is a programming language for parallel computers based on CSP and Pascal. A Joyce program defines concurrent agents which communicate through unbuffered channels. This paper describes a multiprocessor implementation of Joyce.**

## 1 Introduction

Joyce is a programming language for parallel computers based on CSP and Pascal (Wirth 1971; Hoare 1978; Brinch Hansen 1987a). A Joyce program defines concurrent agents which communicate through unbuffered channels. The Joyce compiler is a Pascal program that generates portable code. A small kernel written in assembly language interprets the Joyce code.

The first Joyce kernel was written for a single processor, the IBM PC (Brinch Hansen 1987b). This paper describes the Joyce kernel for a multiprocessor, the Encore Multimax (Encore Corp. 1987). The main difference between the two kernels is the addition of load balancing and locks in the multiprocessor version. This paper is a completely revised version of Brinch Hansen (1987b). The multiprocessor implementation was guided by performance experiments described in Brinch Hansen (1988).

## 2 Language Concepts

The following is a brief summary of the concurrent programming concepts of Joyce.

[*]P. Brinch Hansen, A multiprocessor implementation of Joyce. *Software—Practice and Experience 19*, 6 (June 1989), 579–592. Copyright © 1989, Per Brinch Hansen.

## Agents

A Joyce program consists of nested procedures which define concurrent processes known as agents. When the execution of a program begins, a single, initial agent is automatically activated. Any agent $p$ may activate other agents which are known as the subagents of $p$.

An agent procedure

```
agent Q(formal parameters);
var uninitialized variables;
begin SL end;
```

defines a class of identical agents called $Q$ agents. The formal parameters are value parameters only.

When an agent $p$ executes an agent statement

```
Q(actual parameters)
```

it activates a new subagent $q$ defined by the procedure $Q$. The activation creates a new instance of every variable defined in $Q$. These variable instances are called the own variables of the new agent and are inaccessible to other agents. The formal parameters of $q$ are assigned the values obtained by evaluating the corresponding actual parameters. The other variables of $q$ are uninitialized.

The agent $q$ executes the statement $SL$ defined by its procedure and waits until all its subagents (if any) have terminated. At this point, the own variables of the agent (and all channels created by the agent) cease to exist, and the agent terminates.

An agent procedure $Q$ may be recursive. Every activation of $Q$ creates a new agent with its own variables.

## Channels and ports

Agents exchange symbols through unbuffered channels. Every symbol has a name and may carry a message. A channel can transfer a finite set of symbols only known as the channel alphabet.

The type definition

$$\textbf{type } T = [s_1(T_1),\ s_2(T_2),\ \ldots,\ s_n(T_n)];$$

defines a channel alphabet consisting of symbols named $s_1, s_2, \ldots, s_n$. These symbols carry messages of types $T_1, T_2, \ldots, T_n$, respectively.

An agent accesses a channel through a local variable

$$c\colon T$$

which points to the channel. The variable $c$ is known as a port variable and its type $T$ is called a port type. (The channel itself is not a variable, but a communication device.)

The creation of a new channel is called the activation of the channel. When an agent $p$ executes a port statement

$$+c$$

a new channel with the alphabet given by type $T$ is created and a pointer to the channel is assigned to the port variable $c$. The channel is known as an internal channel of $p$.

If an agent $p$ activates a subagent $q$ using a port variable $c$ as an actual parameter, $c$ denotes a channel which is accessible to both $p$ and $q$. The channel is known as an external channel of $q$.

## Input/output

A communication between two agents takes place when one of them is ready to execute an output statement

$$b!s_i(e)$$

and the other one is ready to execute an input statement

$$c?s_i(v)$$

where

1. $b$ and $c$ are port variables of the same type $T$ which point to the same channel.

2. $s_i$ is the name of a symbol in the alphabet of $T$. The symbol carries a message of type $T_i$.

3. $e$ is an expression and $v$ is a variable of type $T_i$.

When two agents are ready to execute the above statements, the agents are said to match and a communication between them is said to be feasible. The communication consists of assigning the value of $e$ to $v$.

An input/output statement delays an agent until the corresponding communication has taken place.

Two or more agents may share the same channel, but the channel handles only one communication at a time. If agents waiting to communicate can be matched in several different ways, the channel will arbitrarily select two matching agents at a time and let them communicate.

## Polling

An agent may poll one or more channels until it finds that a communication is feasible on one of them. A polling statement

$$\textbf{poll}\ G_1 \mid G_2 \mid \ \ldots\ \mid G_n\ \textbf{end}$$

delays an agent until exactly one of the guarded statements $G_1, G_2, \ldots, G_n$ has been executed. Every guarded statement is either a guarded output statement or a guarded input statement.

A guarded output statement

$$c!s_i(e)\ \&\ B \ {-}{>}\ SL$$

consists of an output statement $c!s_i(e)$, a boolean expression $B$ and a statement $SL$. The guarded statement is executed only if the output is feasible and $B$ denotes true. The execution consists of performing the output followed by execution of $SL$.

Similarly, a guarded input statement

$$c?s_i(v)\ \&\ B \ {-}{>}\ SL$$

is executed only if the input $c?s_i(v)$ is feasible and $B$ denotes true. The execution consists of performing the input and then executing $SL$.

An agent delayed by a polling statement can be matched only by another agent that executes an input/output statement. It cannot be matched by another agent waiting to execute a polling statement.

## 3 Multiprocessor

The Encore Multimax 320 at Syracuse University is a multiprocessor with 18 NS32332 processors. A shared bus connects the processors to a shared memory of 128 Mbytes.

Each processor has a local cache of 64 Kbytes. The cache reduces the load on the shared bus by maintaining local copies of memory locations accessed by the processor.

When a processor needs the value of a memory location, it attempts to fetch the value from its cache. If the cache does not contain a copy of the location, its value is read from the memory and stored in the cache.

When a processor writes a value into a memory location, the value is stored in both the local cache and the memory. If other caches contain previous copies of the same location, these copies are removed.

Any memory location can be used as a spinlock to ensure that processors do not access shared data structures simultaneously. When waiting on a closed lock, a processor first reads the value of the lock into its cache, and then fetches it repeatedly from the cache until another processor changes the lock by opening it.

## 4 Agent Implementation

The multiprocessor system uses the same memory allocation scheme as the single-processor system. Data structures representing agents and channels are kept in memory where they can be accessed by every processor. The most challenging problem of the multiprocessor implementation is to distribute the workload evenly among the processors.

### Program stack

The memory allocation is inspired by the well-known method for implementing recursive, sequential procedures: it uses dynamic allocation of fixed-length segments which are not relocated during their existence.

When an agent (or channel) is activated, it is assigned a fixed segment of memory known as an activation record. An activation record is either an agent record or a channel record. The length of an activation record is determined during compilation.

The creator–subagent relation between agents defines a tree structure in which every agent (and channel) is placed immediately below its creator.

The initial agent is the root of the tree. Since every agent exists at least as long as its subagents (and internal channels), the tree shrinks from the top towards the bottom. More precisely, in every directed branch from a leaf to the root, the agents (and channels) are activated and terminated in LIFO order. Consequently, the activation records form a tree-structured stack. The problem is to allocate this tree-structured stack in a linear memory.

We use a surprisingly simple method for doing this. The method works very well for a large class of interesting programs (but not for all programs). The memory is organized as a single stack of activation records called the program stack.

When an agent (or channel) is activated, its activation record is pushed on top of the program stack and the agent (or channel) is now said to be active. Every agent is linked to its creator and keeps track of how many active subagents it currently has. An agent also keeps a linked list of its internal channels.

When an agent terminates, it becomes passive and all its internal channels are marked as being dead. When all its subagents are dead, a passive agent is also marked as dead. A processor removes a dead activation record only when it is at the top of the program stack.

The program stack has a single lock that prevents processors from removing activation records simultaneously. The processors do not lock the stack during the execution of agents.

## Agent records

An agent record represents an agent $p$ and consists of four parts:

1. An agent state (defined below).
2. The own variables of $p$ (formal parameters and uninitialized variables).
3. The expression stack of $p$.
4. The length of the agent record.

The agent state is defined by the following fields:

1. Phase: an agent is either active, passive or dead.
2. Creator: a pointer to the agent record (if any) which represents the creator of $p$. (The initial agent is the only agent that has no creator.)

3. Subagents: the number of active subagents of $p$.
4. Channel list: a pointer to a channel record (if any) created by $p$. This is the beginning of a linked list of all channel records created by $p$.
5. Queue links: if $p$ is waiting in a queue, the queue links point to the two agents which precede and follow $p$ in the queue.
6. Register values: if $p$ is waiting in a queue, this field holds the current values of the instruction counter and the expression stack pointer of $p$.

During its lifetime, an agent goes through three phases:

1. Active: the agent has not yet reached the end of its procedure.
2. Passive: the agent is waiting for the termination of its subagents.
3. Dead: the agent and its subagents have terminated. The activation record of the agent (and its internal channels) may be removed from the program stack. These records are not removed until they are at the top of the stack. This may occur immediately after the termination of a given agent, or later, when another agent terminates.

## Ready queues

The agent currently being executed by a processor $P$ is called the running agent of $P$ and is represented by a pointer to the corresponding agent record.

Agents that are ready to run wait in FIFO queues called ready queues. Every processor has its own ready queue. The use of multiple ready queues is essential for the performance of parallel programs (Brinch Hansen 1988).

The ready queue of a processor $P$ consists of three parts:

1. An agent queue (described below).
2. The queue length defines the number of agents currently served by $P$. This is the number of agents waiting in the ready queue plus the running agent of $P$ (if any).
3. A lock associated with the queue enables other processors to enter agents in the queue in order to distribute a parallel computation evenly among the processors.

The agent records in a queue form a doubly-linked list. The queue is represented by a single pointer to the agent record (if any) at the head of the queue.

## Processor scheduling

At the beginning of a program execution, one of the processors creates the initial agent and starts executing it. The other processors are initially idle.

An idle processor inspects its own ready queue continuously until the program execution is stopped by another processor or the queue is non-empty. At this point, the processor either selects a running agent from the queue or terminates.

An empty queue becomes non-empty only if another processor enters an agent in the queue in order to balance the computational load of the multiprocessor.

A shared stop signal indicates whether the multiprocessor should stop or continue its execution of agents. When a processor detects termination (or failure) of the program execution, it sets the stop signal.

The scheduling algorithm of an idle processor is called

Select a running agent (if any)

and is defined as follows

```
1. LOCK own ready queue;
2. while not stop and the ready queue is empty do
      begin
         UNLOCK ready queue;
         Delay;
         LOCK ready queue
      end;
3. if not stop then
      begin
         Remove an agent from the ready queue
         and make it the running agent;
         UNLOCK the ready queue
      end
    else
      begin
         UNLOCK the ready queue;
    Terminate the processor
  end
```

The temporary unlocking of the ready queue in step 2 enables another processor to lock the same queue and enter an agent in the queue. To avoid locking conflicts, an idle processor delays itself by executing the following loop without locking its ready queue:

```
Delay:
  while not stop and the ready queue is empty do
    {skip}
```

When a processor enters an agent $q$ in the ready queue, it executes a scheduling algorithm called

Resume the agent $q$

which is defined below:

1. Find the shortest ready queue;
2. LOCK the ready queue;
3. Enter the agent $q$ in the ready queue;
4. UNLOCK the ready queue.

The processors attempt to distribute the workload evenly among themselves by keeping all ready queues of the same length. The abstract algorithms shown here do not include the obvious updating of the ready queue lengths.

While a processor examines all ready queues to find the shortest one, it does not lock and unlock each queue. This prevents unnecessary delays of idle processors that are ready to lock their own queues and select running agents. Occasionally two processors may select the same queue mistakenly, but this will only make the queue slightly longer than it should have been for a short period.

A processor executes a running agent until one of the following events occurs:

1. The agent reaches the end of its procedure.
2. The agent reaches an input/output statement that cannot be completed at the moment.

3. The agent reaches a polling statement that cannot be completed yet.

The scheduling actions taken after each of these events will be explained in the following.

## Agent activation

A running agent $p$ executes an agent statement

$$Q(\text{actual parameters})$$

in seven steps:

```
1.  Evaluate the actual parameters in the expression stack of p;
2.  LOCK the program stack;
3.  Push a new agent record q on the program stack and
     initialize the state fields of q;
4.  Pop the actual parameter values from the expression stack of
     p and assign the values to the formal parameter part of q;
5.  Resume the agent q;
6.  Increment the number of subagents of p by 1;
7.  UNLOCK the program stack.
```

During the activation of an agent, the "uninitialized" variables are set to zero, which represents a nil pointer. This enables the kernel to detect any attempt by the agent to communicate through a non-existing channel.

## Agent termination

When a running agent reaches the end of its procedure, its processor executes the following algorithm:

```
1.  LOCK the program stack;
2.  Let p initially be the running agent and mark p
       as passive;
3.  while p is a passive agent with no subagents do
       begin
         for each internal channel of p do
           begin
             LOCK the channel;
             Mark the channel as dead;
             UNLOCK the channel
```

```
            end;
          Mark $p$ as dead;
          if $p$ has a creator then
            begin {$p$ is not the initial agent}
              $p$ := creator of $p$;
              Decrement the subagents of $p$ by 1
            end
        end;
4.  while there is a dead activation record
        at the top of the program stack do
          Pop the record;
5.  if the program stack is empty then stop := true;
6.  UNLOCK the program stack;
7.  Select a running agent (if any).
```

Every channel has its own lock. The first and last fields of any activation record define the phase and length of an agent (or channel). This convention simplifies the popping of dead records from the program stack.

## 5 Channel Implementation

The implementation of channels also attempts to combine simplicity with efficiency:

1. A message is always copied directly from the activation record of the output agent to the activation record of the input agent without the use of an intermediate buffer.

2. A communication between two matching agents requires only that one of the processors executing the two agents perform a single context switch from one of the agents to another agent. (If one of the agents is polling, several context switches may be necessary.)

These objectives are achieved by allocating a separate queue for each symbol in the alphabet of a channel $x$. When an agent $p$ is ready to output (or input) a symbol $s_i$ through $x$, it examines the corresponding symbol queue to determine if a matching agent $q$ is waiting to participate in the communication. In that case, $p$ obtains the address of the input variable $v$ and the output value $e$ from the expression stacks of $p$ and $q$, assigns $e$ to $v$, and resumes $q$ by moving it to the shortest ready queue. However, if $p$

does not find a matching agent, $p$ enters the symbol queue and waits until a matching agent completes the communication and resumes $p$.

Since a channel may be used by more than two agents, an agent must be able to determine whether the agents in a symbol queue are waiting to input or output the corresponding symbol. Notice that a symbol queue cannot hold both input and output agents simultaneously. It is therefore sufficient to supplement every queue with a boolean indicating whether the queue holds input or output agents when it is non-empty. The solution we have used is to associate both an input and an output queue with every symbol. The amount of memory is the same in both cases: a single word.

Communicating agents circulate between ready queues and channel queues until they terminate. Since agents are resumed in the shortest ready queues, they tend to migrate from one processor to another as the load changes. The migration is trivial to implement because each processor can access any activation record and queue in memory.

## Channel records

A channel record represents a channel $x$ created by an agent $p$. The record consists of five parts:

1. A channel phase indicating whether $x$ is active or dead.
2. A lock associated with the channel.
3. A channel pointer which links the channel record to another channel record (if any) created by $p$. The agent record of $p$ contains the beginning of this chain of internal channels.
4. The channel queues consist of a pair of queues for every symbol $s_i$ in the channel alphabet: a queue of agents waiting to input $s_i$ through $x$ and a queue of agents waiting to output $s_i$ through $x$. The agents waiting in a queue form a doubly linked list. Each queue is represented by a single pointer to the agent (if any) at the head of the queue.
5. The length of the channel record.

## Channel activation

A running agent $p$ executes a port statement

$$+c$$

in six steps:

```
1. Push the address of the port variable $c$ on the
   expression stack of $p$;
2. LOCK the program stack;
3. Push a new channel record on the program stack,
     mark it as active with an open lock, and make
     the channel queues empty;
4. Link the channel record to the agent record of $p$;
5. UNLOCK the program stack;
6. Assign the channel pointer to the variable $c$ and
   pop the address of $c$ from the stack of $p$;
```

## Input/output

When a running agent $p$ reaches an output statement

$$c!s_i(e)$$

its processor does the following:

```
1. Push the channel pointer $c$ and the output
   value $e$ on the expression stack of $p$;
2. LOCK the channel denoted by $c$;
3. **with** the channel queues of symbol $s_i$ **do**
     **if** the input queue of $s_i$ is non-empty **then**
       **begin**
         Remove an agent $q$ from the input queue;
         UNLOCK the channel;
         Pop the address of the input variable $v$
         and $c$ from the stack of $q$;
         Pop the output value $e$ and $c$ from the
         stack of $p$ and assign $e$ to $v$;
         Resume the agent $q$
       **end**
     **else**
       **begin**
         Enter $p$ in the output queue;
```

```
            UNLOCK the channel;
            Select a running agent (if any)
        **end**
```

A processor executes an input statement

$$c?s_i(v)$$

of the running agent $p$ as follows:

```
1.  Push the channel pointer $c$ and the address of
    the input variable $v$ on the expression stack
    of $p$;
2.  LOCK the channel denoted by $c$;
3.  with the channel queues of symbol $s_i$ do
      if the output queue of $s_i$ is non-empty then
        begin
          Remove an agent $q$ from the output queue;
          UNLOCK the channel;
          Pop the address of $v$ and $c$ from the stack
          of $p$;
          Pop the output value $e$ and $c$ from the
          stack of $q$ and assign $e$ to $v$;
          Resume the agent $q$
        end
      else
        begin
          Enter $p$ in the input queue;
          UNLOCK the channel;
          Select a running agent (if any)
        end
```

## Polling

Polling may involve repeated examination of channel queues until a matching agent is found. It is therefore particularly important to reduce the overhead of context switching and channel examination as much as possible.

We use the following method: when an agent $p$ has examined all guarded statements in a polling statement without success, $p$ re-enters the ready queue of its current processor in the hope that a matching agent will enter an appropriate symbol queue before $p$ gets another chance to examine the same guarded statements.

More precisely, when a running agent $p$ reaches a guarded output statement

$$c!s_i(e) \ \& \ B \ -> SL$$

its processor performs the following actions:

```
1.  Push $c$, $e$ and $B$ on the expression stack of $p$;
2.  LOCK the channel denoted by $c$;
3.  **with** the channel queues of symbol $s_i$ **do**
      **if** the input queue of $s_i$ is non-empty **and** $B$ **then**
        **begin**
          Remove an agent $q$ from the input queue;
          UNLOCK the channel;
          Pop the address of the input variable $v$
          and $c$ from the stack of $q$;
          Pop $B$, $e$ and $c$ from the stack of $p$ and
          assign $e$ to $v$;
          Resume the agent $q$;
          $SL$;
          Go to the end of the polling statement
        **end**
      **else**
        **begin**
          UNLOCK the channel;
          Pop $B$, $e$ and $c$ from the stack of $p$;
          Go to another guarded statement
        **end**
```

where

Go to another guarded statement

stands for one of the following actions:

1. If a guarded statement $G_i$ is followed by another guarded statement $G_{i+1}$ in the same polling statement, then go to $G_{i+1}$.

2. Otherwise, set the instruction counter of $p$ at the beginning of the first guarded statement $G_1$ in the polling statement, re-enter $p$ in the ready queue of its current processor, and select a running agent from the ready queue. When the execution of $p$ is resumed, it will repeat the examination of the same guarded statements beginning with $G_1$.

Notice that the examination of a guarded statement has no effect if it cannot be executed.

The execution of a guarded input statement is very similar.

### System channel

A Joyce program communicates with the computer peripherals through a predefined channel known as the system channel. The initial agent has access to this channel through an actual parameter initialized by the kernel. A communication through this channel is implemented by a system procedure which performs the corresponding peripheral operation. The system channel has its own lock.

## 6 Locks

When a processor needs simultaneous access to several shared data structures, it locks them one at a time in fixed order to prevent deadlock (Brinch Hansen 1973):

1. The program stack.
2. A channel activated by an agent.
3. A ready queue.
4. The system channel.

As an example, during the activation of a new agent, a processor locks the program stack before locking the ready queue in which the agent is entered. Similarly, during the termination of an agent, a processor locks the program stack before locking any channel created by the agent.

Since the system channel has the lowest priority, a processor can lock it during any kernel operation and display a trace of a significant event, such as the activation or termination of an agent or channel or the completion of a communication.

The numbers of ready queues and locks used are based on performance experiments described in Brinch Hansen (1988).

## 7 Unix Interface

The Joyce kernel runs under Umax 4.2, a Multimax version of the Berkeley Unix system (Encore Corp. 1987).

Initially a user communicates with a single Unix process called the master process. When the user decides to run a Joyce program on $p$ processors, the master process creates $p$ additional Unix processes known as the Joyce processors. The Joyce processors share two memory segments which contain the kernel, the Joyce program, the program stack and $p$ ready queues. In addition, every Joyce processor has a local memory segment for its own temporaries.

Every Joyce processor has a unique index which identifies its ready queue. To begin with, one of the processors creates the initial agent and starts executing it. The other processors remain idle until agents arrive in their queues (or the program execution is stopped).

A Joyce processor stops the program execution by setting the stop signal and performing an exit operation. This wakes up the master process, which then kills every Joyce processor and terminates itself.

A Joyce program has access to two text files: an input file and an output file. The program accesses these files through the system channel. Communication through the system channel is performed by means of Unix procedures.

## 8 Execution Times

Table 1 illustrates the performance that can be achieved by a portable implementation of a parallel programming language. The execution times of Joyce programs running on an Encore Multimax 320 are expressed in terms of

| | |
|---|---|
| $c$: | the number of channels activated by an agent. |
| $m$: | the number of simple components in a structured operand. |
| $n$: | the number of iterations. |
| $p$: | the number of processors used. |
| $s$: | the number of symbols in a channel alphabet. |

**Table 1** Execution times ($\mu$s).

| | Simple operands (1 word) | Real operands (2 words) | Structured operands ($m$ words) |
|---|---|---|---|
| Constant | 4.1 | 7 | $15 + 2.8m$ |
| Variable address | 4.8 | 5 | 5 |
| Variable value | 4.8 | 8 | $14 + 1.2m$ |
| $v[i]$ address | 23 | 23 | 23 |
| $v[i]$ value | 27 | 30 | $32 + 1.2m$ |
| := | 4.7 | 7 | $9 + 1.2m$ |
| < = > <= <> >= | 4.7 | 9 | $11 + 2.7m$ |
| − sign | 5.1 | 8 | |
| + − | 5.5 | 9 | |
| * | 9.0 | 9 | |
| / **div mod** | 12 | 13 | |
| **and or** | 4.2 | | |
| **not** | 3.8 | | |
| real($i$) | 13 | | |
| integer($i$) | | 15 | |
| $v!s(i)$ $v?s(i)$ | $93 + 5.2p$ | $95 + 5.2p$ | $96 + 1.7m + 5.2p$ |

The input/output times increase with the message length $m$ and the number of processors $p$. When a processor resumes a communicating agent it examines all ready queues to find the shortest one and changes the length of that queue. Since all processors examine the ready queues, any change will eventually force all of them to refetch the updated queue length from memory. So the search and refetching both introduce communication overhead proportional to the number of processors. This overhead effectively limits the speedup of Joyce programs when the number of processors is increased.

Other execution times are:

| | |
|---|---|
| Port statement $+v$ | $44 + 7.6s$ |
| Agent statement $P(e)$ | $160 + e$ |
| Agent termination | $110 + 42c$ |
| **if** $B$ **then** $S_1$ | $8.5 + B + S_1$ or |
| **else** $S_2$ | $5.3 + B + S_2$ |
| **while** $B$ **do** $S$ | $5.3 + B + (8.5 + B + S)n$ |
| Empty program | $10000 + 60000p$ |

The execution time of an agent statement is the activation time of an agent with a small number of own variables, such as the sorting agent defined in Brinch Hansen (1987a).

The execution time of an empty program is mainly the overhead required to activate and terminate $p$ Unix processes.

## 9 Final Remarks

We have moved the parallel programming language Joyce from a single processor to a multiprocessor. We now plan to use the multiprocessor implementation for scientific and engineering computations.

## Acknowledgements

Anand Rangachari reprogrammed the Joyce kernel for the Encore Multimax. It is a pleasure to acknowledge the constructive comments of Birger Andersen, Gideon Frieder and Anand Rangachari. This work was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University, which is funded by DARPA, under contract to Rome Air Development Center (RADC), Griffis AFB, NY.

## References

Brinch Hansen, P. 1973. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. 1987a. Joyce—A programming language for distributed systems. *Software—Practice and Experience 17*, 1 (January), 29–50. *Article 18.*

Brinch Hansen, P. 1987b. A Joyce implementation. *Software—Practice and Experience 17*, 4 (April), 267–276.

Brinch Hansen, P., and Rangachari, A. 1988. Joyce performance on a multiprocessor. School of Computer and Information Science, Syracuse University, Syracuse, NY.

Encore Corp. 1987. *Multimax Technical Summary.* Encore Computer Corporation, Marlboro, MA.

Hoare, C.A.R. 1978. Communicating sequential processes. *Communications of the ACM 21*, 8 (August), 666–677.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 35–63.