9-1988

Joyce Performance on a Multiprocessor

Per Brinch Hansen
Syracuse University, School of Computer and Information Science, pbh@top.cis.syr.edu

Rangachari Anand
Syracuse University, School of Computer and Information Science

CIS-88-5

JOYCE PERFORMANCE ON A MULTIPROCESSOR

Per Brinch Hansen and Anand Rangachari

School of Computer & Information Science
Syracuse University
Syracuse, New York 13244, U.S.A.

JOYCE PERFORMANCE ON A MULTIPROCESSOR

PER BRINCH HANSEN AND ANAND RANGACHARI

School of Computer and Information Science
Syracuse University
Syracuse, New York 13244

September 1988

Abstract - Joyce is a parallel programming language based on CSP and Pascal. The language has been moved from the IBM PC to the Encore Multimax. The paper explains how the multiprocessor implementation of Joyce was guided by performance evaluation. The measurements show that the speed-up of Joyce programs follows Amdahl's law.

Index Terms - Programming languages, concurrent programming, communicating agents, multiprocessors, language implementation, performance evaluation, Joyce.

I. INTRODUCTION

Joyce is a parallel programming language based on CSP and Pascal [1, 2, 3]. The language has been moved from the IBM PC to the Encore Multimax [4, 5, 6]. This paper explains how the multiprocessor implementation of Joyce was guided by performance evaluation. The measurements show that the speed-up of Joyce programs is determined by Amdahl's law [7].

II. BENCHMARK

The purpose of the performance experiments was to enable us to make a meaningful choice between different implementations of Joyce. Initially, we learned two lessons the hard way.

Our first benchmark was a parallel prime sieve [1]. This turned out to be a poor benchmark, since its performance is limited not only by the Joyce implementation but also by the sieve algorithm.

On a parallel computer, performance measurements are often distorted by unpredictable factors. These include operating system overhead, the presence of other users and incorrect measurement procedures.

Our initial efforts were inconclusive until we realized these problems and established two guidelines:

1. A benchmark must be utterly simple to reveal only the performance limitations of the language implementation.

2. Performance measurements should be trusted only if they agree with an analytic model of the phenomena observed.

Joyce is well-suited for highly parallel computations in which a large number of processes exchange short messages. To find out if this programming style is practical on a multiprocessor, we used a benchmark with 200 independent pairs of processes as shown in Fig. 1. Each pair consists of a sender, which outputs a fixed number of messages through an unbuffered channel, and a receiver, which inputs the messages from the channel. The messages each occupy one word.

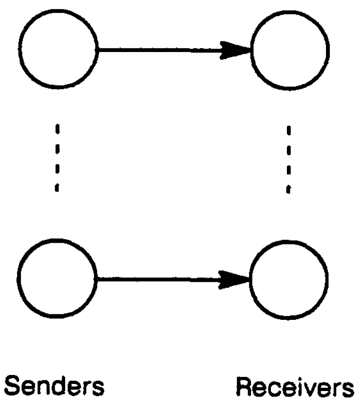

Fig. 1. Benchmark.

The benchmark pushes the multiprocessor to its practical limits in three ways:

1. The number of processes executed simultaneously is an order of magnitude larger than the number of processors available.

2. To make the relative overhead of communication as large as possible, the processes exchange the smallest

possible messages and perform minimal computations.

3. After each communication, a process migrates from one processor to another in order to distribute the load evenly among the processors.

In the following, we will also use the benchmark to illustrate the programming concepts of Joyce.

A Joyce program consists of nested procedures. Each procedure defines a class of identical processes known as agents.

```
agent BM(io: iosym);
const q = 200 {agents};
  m = 3500 {messages per agent};
  n = 4 {iterations per message};
type stream = [data(integer)];

agent SENDER(c: stream; m, n: integer);
var i, k: integer;
begin SEND end;

agent RECEIVER(c: stream; m: integer);
var j, k, n: integer;
begin RECEIVE end;

var c: stream; i: integer;
begin ACTIVATE end;
```

Each pair of agents communicates through a channel c of type stream. Through this channel, the agents can transmit a sequence of named symbols. The benchmark uses only one kind of symbol (named data). This symbol carries a message of type integer.

When program execution begins, a single agent defined by the outermost procedure BM is automatically activated. This initial agent activates the senders and receivers

```
ACTIVATE: i := 0;
          while i < q do
          begin
            +c;
            SENDER(c, m, n);
            RECEIVER(c, m);
            i := i + 2
          end
```

The statement

```
+c
```

creates a new channel and assigns a reference to the channel to the variable c. Strictly speaking, it should be called "the channel denoted by c". However, we will often refer to it simply as "the channel c".

The agent statements

```
SENDER(c, m, n);
RECEIVER(c, m)
```

activate a new pair of agents with access to the same channel c. These agents run in parallel with all other agents (including the initial agent).

A sender produces and outputs m messages

```
SEND: i := 1;
      while i <= m do
      begin
        COMPUTE;
        c!data(n);
        i := i + 1
      end
```

A receiver inputs and consumes m messages

```
RECEIVE: j := 1;
         while j <= m do
         begin
           c?data(n);
           COMPUTE;
           j := j + 1
         end
```

A communication between two agents takes place when a sender is ready to output on a channel c

```
c!data(n)
```

and a receiver is ready to input from the same channel

```
c?data(n)
```

The communication assigns the value of the parameter n of the sender to the local variable n of the receiver. (In general, the effect of a communication is to assign the value of an output expression e to an input variable x.)

For each communication, an agent performs a local computation simulated by a loop

```
COMPUTE: k := 1;
         while k <= n do k := k + 1
```

The amount of computation can be varied by changing the number of iterations n.

When a sender (or receiver) reaches the end of its procedure, it terminates. When all senders and receivers have terminated, the initial agent terminates. This completes the program execution.

III. THE ENCORE MULTIMAX

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 Mb. Each processor has a local cache of 64 kb which maintains local copies of memory locations accessed by the processor. When a processor writes a value into a memory location, the value is stored in both the local cache and the memory location. 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 a processor waits on a closed lock, it reads the lock into its cache once and continues to fetch it from the cache until another processor changes the lock by opening it.

IV. THE MULTIPROCESSOR KERNEL

The Joyce compiler generates portable code which is interpreted by a kernel of 2300 lines written in assembly language.

The benchmark creates a fixed number of agents which exchange thousands of messages. The performance of such a program is limited by the speed of communication (but not by the initial creation of channels and agents). Consequently, we will consider only how the processors execute agents and synchronize communications. For a detailed explanation of the multiprocessor kernel, see [5].

Every channel and agent is represented by a memory segment of fixed length called an activation record. Agent records can be chained together to form queues of agents.

The agents that are ready to run wait in queues known as ready queues (Fig. 2). Every processor has its own ready queue. When a processor is idle, it selects an agent from its ready queue and executes it until the agent either terminates or waits for a communication to take place.

A channel that can transfer several different kinds of symbols has a separate agent queue for each symbol. Since the benchmark agents exchange one kind of symbol only, each channel has just one queue.

When a running agent p is ready to communicate on a channel, its processor examines the channel queue to see if a matching agent q is waiting to communicate on the same channel. In that case, the processor retrieves the output value e and the address of the input variable x from the activation records of p and q, assigns e to x, and moves q from the channel queue to the shortest ready queue. However, if no other agent is ready to communicate on the channel, the processor enters p in the channel queue and selects another running agent (if any) from its ready queue.

Communicating agents circulate between ready queues and channel queues until they terminate.

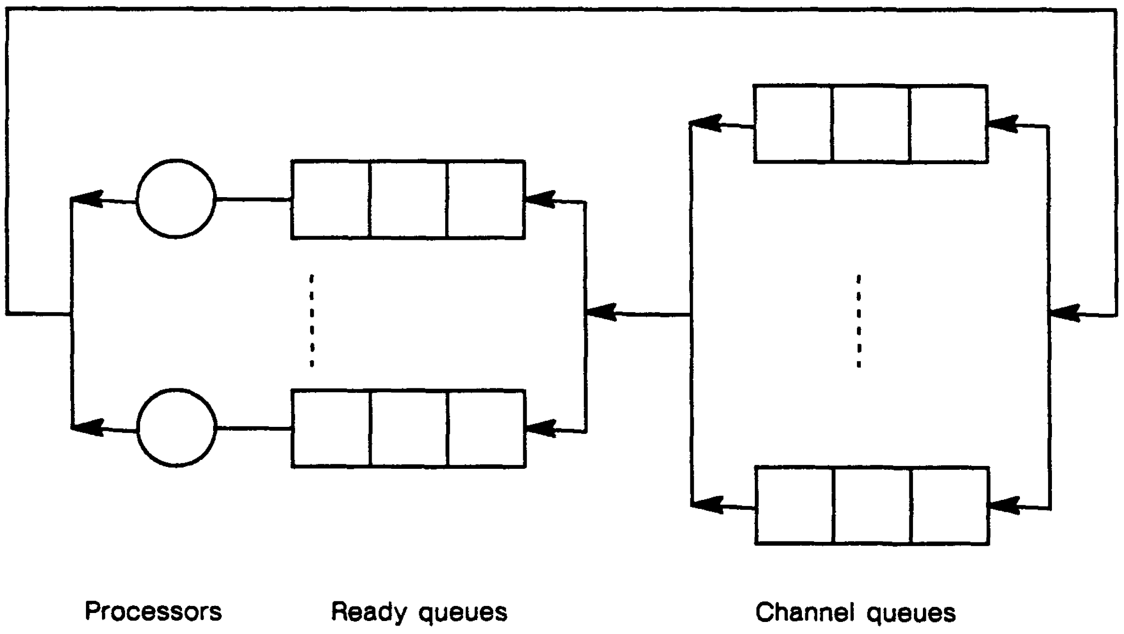

Fig. 2. Queuing network.

V. EXECUTION TIMES

The execution times of Joyce programs running on an Encore Multimax 320 are expressed in terms of

n	the number of iterations.
p	the number of processors.

The following execution times (in us) apply to operands of simple types

Constant	4.1
Variable	4.8
:=	4.7
<=	4.7
+	5.5
c!data(n)	93 + 5.2p
c?data(n)	93 + 5.2p
while B do S	5.3 + B + (8.5 + B + S)n

The execution times of other operations, including the creation of channels and agents, are listed in [5].

Communication times increase with the number of processors p. If a Joyce program runs on 10 processors, the input or output of a single integer takes

$$93 + 5.2 \times 10 = 145 \text{ us}$$

The most likely explanation of this phenomenon is the following: When a processor inputs or outputs a message, it scans a shared table of length p to find the shortest ready queue. The processor then increments the length of that queue. Since the processors share the ready queues, this change eventually forces every processor to refetch the updated value from memory. Later, we will show that the scaled overhead of 5.2 us per processor effectively limits the possible speed-up of Joyce programs when the number of processors is increased.

The benchmark consists of cyclical agents. In each cycle, an agent participates in a single communication and performs a local computation. The cycle time of a sender is determined by adding the following execution times:

```
while i <= m do          8.5 + 4.8 + 4.7 + 4.8
begin
  k := 1;                4.8 + 4.7 + 4.1
  while k <= n do        5.3 + 14.3 + (8.5 + 14.3)n
    k := k + 1;         (4.8 + 4.7 + 4.8 + 5.5 + 4.1)n
  c!data(n);              93 + 5.2p
  i := i + 1             4.8 + 4.7 + 4.8 + 5.5 + 4.1
end
```

The cycle time

$$t(p) = 173 + 47n + 5.2p$$

increases with the amount of computation n and the number of processors p. A receiver has the same cycle time.

VI. AMDAHL'S LAW

Consider the execution of q cyclical agents each of which communicates m times. The agents have identical cycle times of the form

$$t(p) = a + bp$$

where _a_ and _b_ are constants.

A single processor can obviously execute such a program in time

$$T(1) = q\ m\ t(1) = q\ m\ (a + b)$$

We will show that p processors can execute the same program in time

$$T(p) = q\ m\ t(p)/p = q\ m\ (a/p + b)$$

T(1) and T(p) are called the sequential and parallel execution times of the program.

The parallel speed-up

$$S(p) = T(1)/T(p)$$

defines how many times faster the program runs on p processors compared to a single processor.

The speed-up can be rewritten as follows

$$S(p) = \frac{p}{1 + (p - 1)f}$$

where

$$f = b/(a + b)$$

is the fraction of time each agent spends on scaled overhead. This is also known as Amdahl's law [7].

For our benchmark with q = 200 agents, we have

$$T(p) = 200m\ \left(\frac{173 + 47n}{p} + 5.2\right)\ us$$

We used three variants of the benchmark called BM1, BM2, and BM3. The execution times of these benchmarks were measured on 1-10 processors. Each benchmark used a different value of n to obtain a different speed-up. The number of messages m was selected to make a benchmark run for approximately 30 s on ten processors.

Table 1 shows the predicted performance of these benchmarks. The speed-up is limited by minute fractions of scaled overhead (f = 0.001 - 0.029).

BM	m	n	T(p) s	f
1	6500	0	224.9/p + 6.8	0.029
2	3500	4	252.7/p + 3.6	0.014
3	300	104	303.7/p + 0.3	0.001

Table 1. Predicted performance of benchmarks.

The curves and plotted points in Fig. 3 represent the predicted and measured execution times of BM1. The model also accurately predicts the run times of BM2 and BM3.

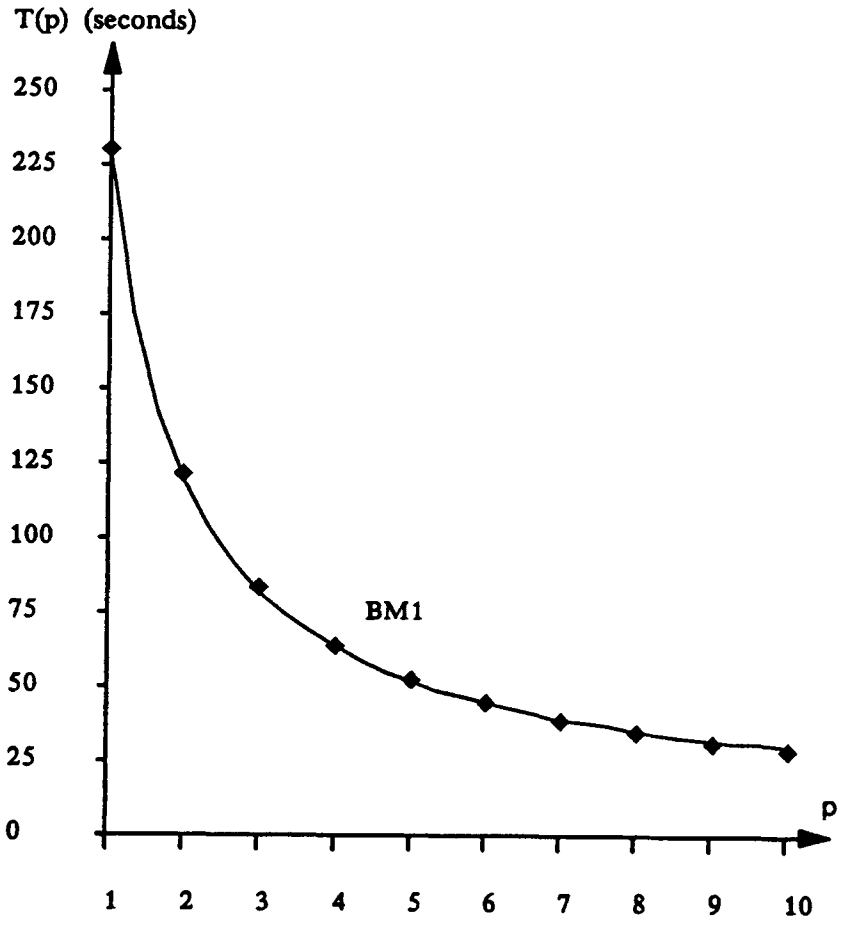

Fig. 3. Execution times of BM1.

Figure 4 shows excellent agreement between the predicted and measured speed-up of the benchmarks. BM1 is an extreme example of a computation in which parallel agents exchange very short messages with minimal processing of each message. This benchmark defines a lower bound on speed-up. Most programs will perform better! It is encouraging that ten processors can speed this demanding benchmark up by a factor of almost eight. BM2 and BM3 show that one can get arbitrarily close to linear speed-up by increasing the amount of computation per communication.

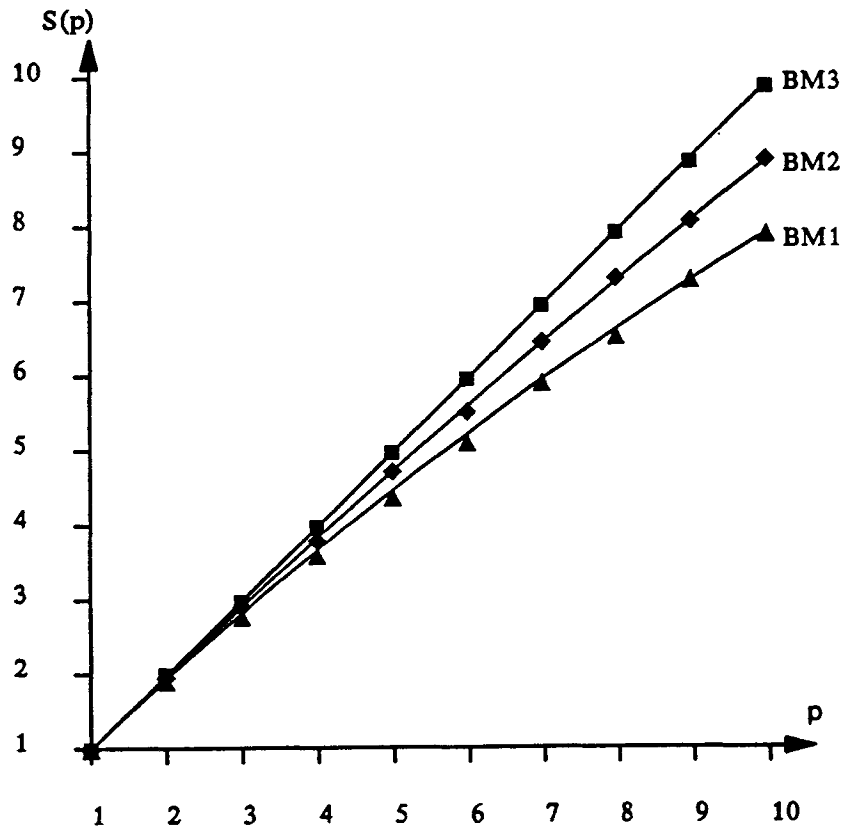

Fig. 4. Speed-up of benchmarks.

VII. SCALED BENCHMARK

Several researchers have pointed out that linear speed-up can be achieved by scaling a computation up as the number of processors increases. See, for example, [8].

From the previous benchmarks, we derived a scaled benchmark that demonstrates this principle. We assume that each agent communicates a fixed number of words m and performs a fixed computation for every word. The idea is to reduce the communication time by sending a block of n integers in each message.

```
type block = array [1..n] of integer;
  stream = [data(block)];
var x: block;
```

The input (or output) of n integers takes

```
c?data(x)     96 + 1.7n + 5.2p us
```

A fixed computation per word is simulated by letting the agents examine every integer in a message. To account for this, we need the average execution time of the following statements in senders and receivers, respectively:

```
x[k] := y     y := x[k]     35 us
```

In the scaled benchmark, the receivers repeat the following cycle b times, where b = m/n is the number of blocks received

```
b := m div n;
while j <= b do
  begin
    c?data(x);
    k := 1;
    while k <= n do
      begin
        y := x[k];
        k := k + 1
      end;
    j := j + 1
  end
```

The cycle time of a receiver (or sender) is

$$t(p) = 176 + 83n(p) + 5.2p$$

As a result of the scaling, the message length n(p) is a function of the number of processors p.

The cycle time is of the form

$$t(p) = a + bn(p) + cp$$

where *a*, *b* and *c* are constants.

A linear speed-up

$$S(p) = p$$

is obtained if each processor uses the same amount of time per message word independent of the number of processors used, that is

$$t(p)/n(p) = t(1)/n(1)$$

or

$$\frac{n(p)}{n(1)} = 1 + \frac{c}{a + c}(p - 1)$$

Since the message length can be increased by multiples of one word only, we chose

$$n(1) = (a + c)/c = a/c + 1$$

Consequently,

$$n(p) = a/c + p$$

and

$$t(p)/n(p) = b + c$$

For the scaled benchmark, we have

$$n(p) = 176/5.2 + p = 34 + p \text{ words/message}$$

and

$$t(p)/n(p) = 83 + 5.2 = 88.2 \text{ us/word}$$

If 40 pairs of agents each communicate 42500 integers, we have

$$T(p) = T(1)/p = 80 \times 42500 \times 88.2/p \text{ us} = 299.9/p \text{ s}$$

Figure 5 shows the predicted and measured linear speed-up of the scaled benchmark.

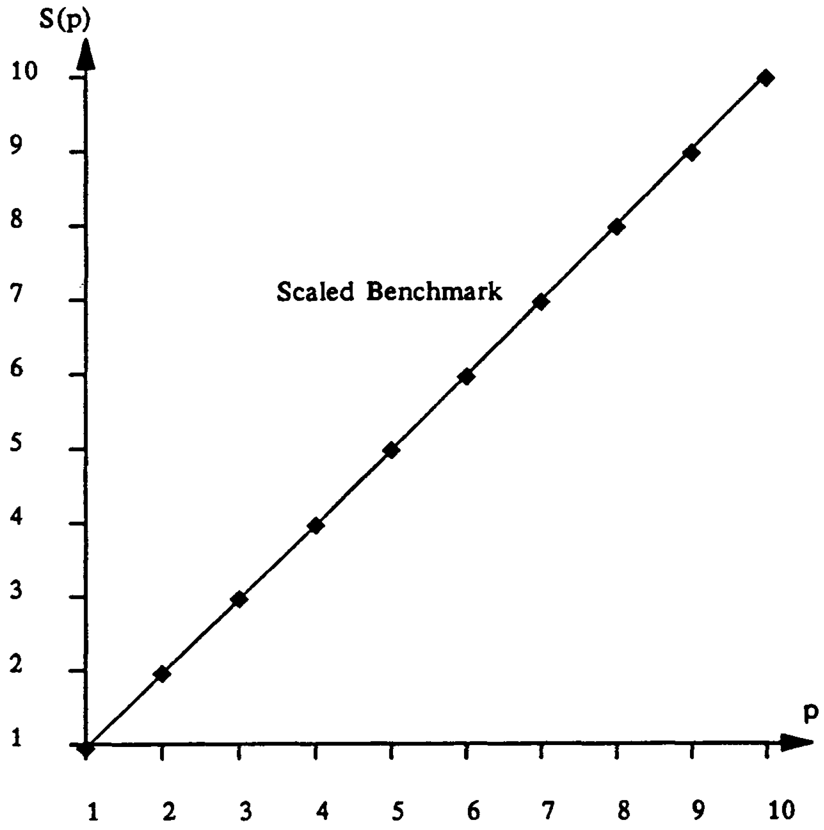

Fig. 5. Speed-up of scaled benchmark.

VIII. THE COST OF POLLING

If a receiver does not know in advance how long a stream of messages is, the sender can output a (possibly empty) sequence of data symbols followed by an eos symbol which signals the end of the stream. In that case, the channels must be of the following type

```
type stream = [data(integer), eos];
```

A sender now behaves as follows

```
SEND;
c!eos
```

while a receiver executes the algorithm

```
more := true;
while more do
  poll
    c?data(n) -> COMPUTE |
    c?eos -> more := false
  end
```

The polling statement delays the receiver until the sender is ready to output one of the two possible symbols on the channel c:

1. If the sender outputs a data symbol, the receiver inputs the symbol, performs a local computation, and repeats the execution of the polling statement (since more remains true).

2. If the sender outputs eos, the receiver inputs the symbol and terminates the loop (by setting more to false).

The channel c has an agent queue for each of the two symbols it can transmit. The polling is implemented by examining these queues one at a time to determine if another agent is ready to output one of the symbols. If the examination is unsuccessful, the corresponding processor reenters the receiver in its own ready queue and selects a running agent from the same queue.

If an agent is unable to complete simple input (or output), it waits in a channel queue until the communication can take place. Polling is a more expensive form of communication which may waste processor time by examining the same channel queues repeatedly.

In Fig. 6, the curve represents the speed-up of the simple benchmark BM1. The plotted points show the measured speed-up of the corresponding polling benchmark.

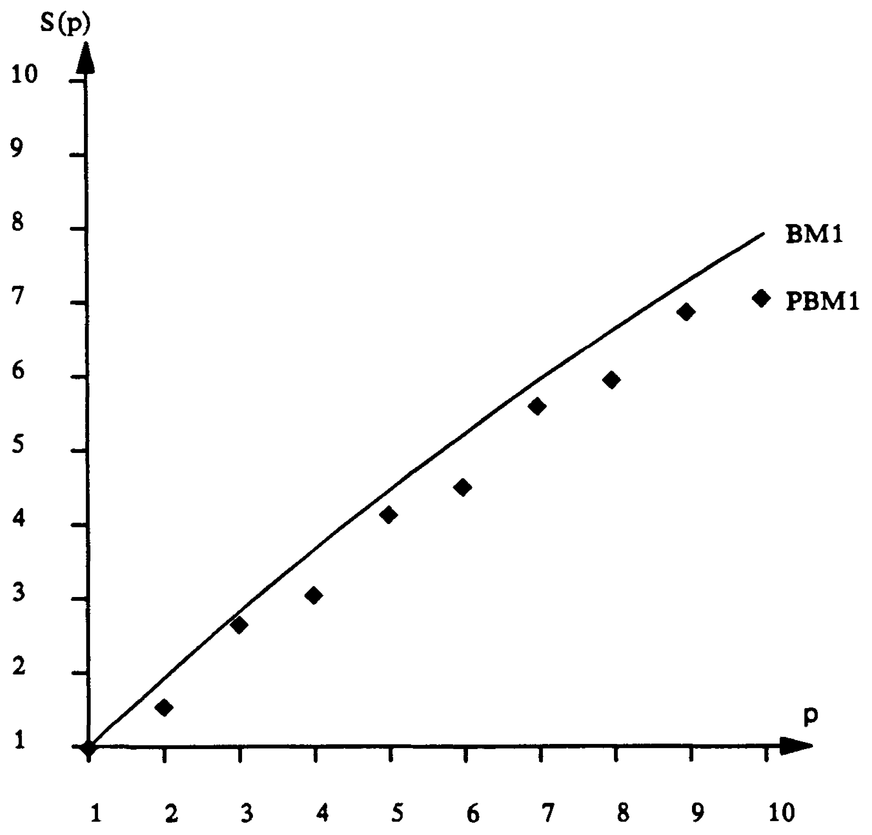

Fig. 6. Speed-up with and without polling.

IX. DESIGN DECISIONS

The most crucial design decisions are the number of ready queues used, the number of locks required, and the selection of the ready queue in which an agent is entered after a communication. The challenge is to balance the work load evenly among the processors with minimal loss of efficiency. Several possibilities were evaluated and rejected.

1. Load balancing is trivial if the processors share a single ready queue. A lock associated with the queue ensures that the processors never attempt to access the queue simultaneously. We did not really expect this simple idea to work well for agents that communicate frequently, but were curious to find out how poor it is. As you might expect, the common lock becomes a bottleneck which forces the processors to work sequentially when they select or reactivate agents. With ten processors, the speed-up of a

benchmark similar to BM1 was 2.7 only.

2. From then on we used a separate ready queue for each processor. A processor always selects a running agent from its own ready queue. In order to balance the load among the processors, a processor must be able to enter agents in the ready queues of other processors. So each ready queue must have its own lock. The processors refer most often to their own ready queues. Rarely will several processor compete for access to the same queue. Consequently, idle processor time caused by locking is largely eliminated. (This was verified by experiment.)

However, when several ready queues are used, it is more difficult to keep the load evenly balanced between them.

3. At one point, it seemed reasonable to let each processor reactivate agents cyclically among the ready queues starting with its own ready queue. Unfortunately, this simple algorithms turned out to be unstable. Due to random fluctuations, one of the ready queues will always at some point be somewhat longer than the other ready queues. When an agent from the longer queue communicates, it will eventually enter a channel queue and, later, join one of the shorter ready queues. Since that ready queue is short, the agent is soon resumed, and moves via a channel queue to the next short queue, and so on. After a few more communications, the agent is right back where it came from - in the long queue. This unstability makes some processors work overtime, while others are underutilized.

4. Our next idea was to maintain a table of the lengths of the ready queues. When a processor reactivates an agent, it scans the table and enters the agent in the shortest ready queue. This algorithm might increase the chance of processor delays, if the agent had to lock and unlock every ready queue during a search. We avoid this problem by scanning the table without locking the queues. After finding the shortest queue, a processor locks that queue only before entering an agent. Occasionally, several processors may select the same queue simultaneously and extend it. However, this will only temporarily make a queue slightly longer than it should be. The imbalance will be corrected as soon as some of the agents communicate and move to other ready queues.

5. When the length of a ready queue does not include the running agent, another anomaly can occur. Consider a Joyce program with two agents only running on two processors. If an agent p running on one of the processors communicates with the other agent q, while q is waiting in a channel

queue, the processor may enter q in its own (empty) ready queue. Both agents now run on the same processor, while the other processor is idle. (This phenomenon was also demonstrated by experiment.) In the final kernel, the queue length defines the number of agents currently served by a processor. This is the number of agents waiting in the ready queue plus the running agent (if any).

6. The previous algorithm can be improved further by measuring the amount of processor time used by an agent from the moment it is selected as a running agent until it enters a channel queue. When an agent is reactivated, its previous time slice is used as an estimate of its next time slice. The length of a ready queue is replaced by the estimated amount of processing time needed to allocate another time slice to each of the running and waiting agents. Although one can construct unusual Joyce programs that run faster under this scheduler, it does not improve the performance of ordinary programs, such as the benchmarks. So our present choice is the simpler scheduler described previously.

Every channel has its own lock. If a channel connects two agents only, at most two processors can attempt to access it simultaneously. Even that is a rare event which can occur only if the two agents happen to be running simultaneously. So we do not expect channel locks to reduce processor performance.

The memory allocation of activation records takes place in a stack as described in [5]. A single lock ensures that processors create and remove activation records one at a time in this program stack. Since activations and terminations of agents and channels are rare events compared to communications, we do not anticipate that this lock will influence the performance of the multiprocessor significantly.

X. FINAL REMARKS

We have moved the parallel programming language Joyce from a single processor to a multiprocessor. The main design decisions were the number of scheduling queues used, the number of locks required, and the implementation of load balancing. We have treated these decisions as performance issues and settled them by benchmark experiments. The performance of the final product is predicted accurately by a simple deterministic model.

ACKNOWLEDGEMENTS

The paper was improved by the suggestions and comments of Birger Andersen and Gideon Frieder. 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), Griffiss AFB, NY.

REFERENCES

[1] P. Brinch Hansen, "Joyce - A programming language for distributed systems," Software - Practice and Experience, vol. 17, pp.29-50, 1987.

[2] C. A. R. Hoare, "Communicating sequential processes," Comm. ACM, vol. 21, pp. 666-677, 1978.

[3] N. Wirth, "The programming language Pascal," Acta Informatica, vol. 1, pp.35-63, 1971.

[4] P. Brinch Hansen, "A Joyce implementation," Software - Practice and Experience, vol. 17, pp. 267-276, 1987.

[5] P. Brinch Hansen, A Multiprocessor Implementation of Joyce. School of Computer and Information Science, Syracuse University, Syracuse, NY, 1988.

[6] Encore Corp., Multimax Technical Summary. Encore Computer Corp., Marlboro, MA, 1987.

[7] G. Amdahl, "Validity of the single-processor approach to achieving large-scale computer capabilities," Proc. AFIPS Conf., vol. 30, pp. 483-485, 1967.

[8] J. L. Gustafson, "Reevaluating Amdahl's law," Comm. ACM, vol. 31, pp. 532-533, 1988.

Per Brinch Hansen (F'85) is Distinguished Professor of Computer Science at Syracuse University. He is the designer of the programming languages Concurrent Pascal, Edison and Joyce. Dr. Brinch Hansen's text books on Operating System Principles (1973) and The Architecture of Concurrent Programs (1977) have been published in six languages.

Anand Rangachari received the B.S. degree in Chemistry from Indian Institute of Technology, New Delhi, in 1984 and the M.S. degree in Computer Science from Syracuse University in 1986. He is currently pursuing the Ph.D. degree in Computer Science at Syracuse University.