Syracuse University
SURFACE

Electrical Engineering and Computer Science Technical Reports

College of Engineering and Computer Science

7-1989

Analysis of a Parallel Mergesort

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

Follow this and additional works at: http://surface.syr.edu/eecs_techreports

Part of the Computer Sciences Commons

Recommended Citation
Hansen, Per Brinch, "Analysis of a Parallel Mergesort" (1989). *Electrical Engineering and Computer Science Technical Reports.* Paper 55.
http://surface.syr.edu/eecs_techreports/55

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Electrical Engineering and Computer Science Technical Reports by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

SU-CIS-89-03

Analysis of a Parallel Mergesort

Per Brinch Hansen

July 1989

School of Computer and Information Science
Syracuse University
Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

ANALYSIS OF A PARALLEL MERGESORT

PER BRINCH HANSEN

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

July 1989

SUMMARY

The paper describes a performance model of a fine-grained, parallel mergesort which sorts N elements in O(N) time using O(logN) processors. The model predicts both the communication time involved in merging the elements and the decomposition time required to activate and terminate the binary tree of processes. The parallel algorithm is written in Joyce and runs on an Encore Multimax.

KEY WORDS Parallel mergesort Multiprocessor algorithm Performance analysis Communication time Decomposition time

INTRODUCTION

The parallel mergesort is an extreme example of fine-grained parallelism. Its performance is limited not only by the speed of process communication, but also by the overhead of process creation.

The algorithm recursively activates a binary tree of parallel processes. The leaf processes input a finite, unordered sequence of elements and split it into sequences of length 1. The branch processes gradually merge smaller, ordered sequences into longer, ordered sequences until a single, ordered sequence has been output by the root process. While this is going on, the process tree gradually terminates from the top down.

Knuth attributes the sequential mergesort to John von Neumann [1]. Several researchers have analyzed the running time of the parallel mergesort under the assumption that the overhead of process creation is negligible [2, 3].

We will describe an exact model of the parallel mergesort which predicts both the communication time involved in merging the elements and the decomposition time required to activate and terminate the binary tree of processes.

Copyright (c) 1989 Per Brinch Hansen

The parallel algorithm is written in the programming language Joyce which supports recursive processes and synchronous communication [4]. The Joyce program runs on an Encore Multimax [5].

The parallel mergesort pushes the multiprocessor to its technological limits in two ways:

1. The number of parallel processes is several orders of magnitude larger than the number of processors.

2. The processes exchange the smallest possible messages (single integers) with minimal computation.

PARALLEL ALGORITHM

The parallel mergesort uses a binary tree of processes to sort a finite number of elements. The processes communicate by messages only. The number of elements to be sorted is known a priori.

Figure 1 shows a binary tree that sorts four integers. The tree consists of seven processes (the nodes) connected by eight communication channels (the edges). The sorting takes place in several stages which are partially overlapped.

First, the tree is created recursively. The environment activates the root which then activates its two children which, in turn, activate their own children, and so on, until the leaves have been activated. All processes in the tree now run in parallel.

At the top of the tree the leaves input an unordered sequence of four integers

(3, 1, 2, 4)

from a common input channel and split it into four sequences of length 1

(3) (1) (2) (4)

This is achieved by letting each leaf input a single integer and output it to its parent in the tree.

At the next level in the tree each process inputs two sequences of length 1 from its children and merges them into a single, ordered sequence of length 2. We have now combined the four elements into two ordered sequences

(1, 3) (2, 4)

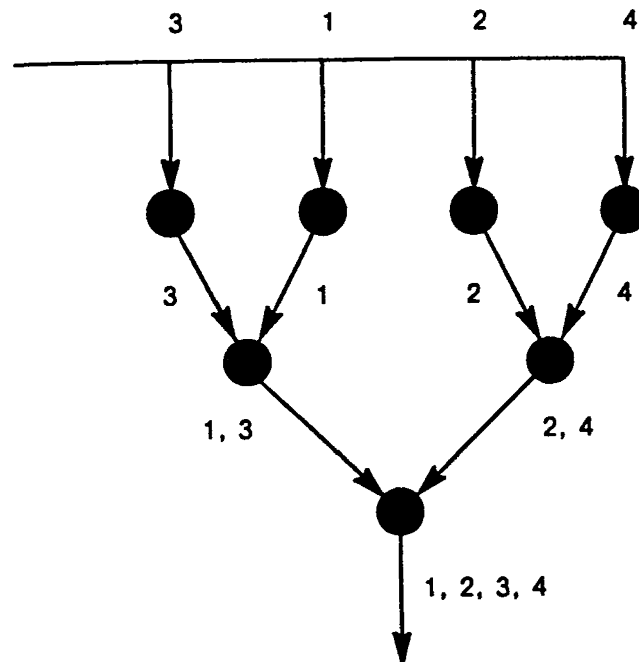

Fig. 1. Parallel merge tree.

At the root of the tree a single process inputs two ordered sequences of length 2 from its children and merges them into a single ordered sequence of length 4

(1, 2, 3, 4)

which is output to the environment. This completes the sorting.

During the sorting each process terminates as soon as it has output the last element to its parent.

JOYCE PROGRAM

The parallel mergesort is written in the programming language Joyce.

A leaf is defined by a procedure that copies a single element from an input channel to an output channel

```
type sequence = [element(integer)];

procedure COPY(input, output: sequence);
var x: integer;
begin input?element(x); output!element(x) end;
```

The two channels

input, output: sequence

are both of type sequence. Each channel can transmit a sequence of messages. Each message is called an element and carries an integer value.

The execution of the statement

input?element(x)

inputs an element from the input channel and assigns its value to the variable x.

The execution of

output!element(x)

outputs the value of the element x through the output channel.

The channels do not buffer messages. When a process is ready to input a message from a channel, it is delayed until another process is ready to output a message from the same channel. When two processes are ready to communicate, a communication takes place between them.

The next procedure merges a left sequence of length L and a right sequence of length R into a single output sequence of length L + R

```
procedure MERGE(left, right,
  output: sequence; L, R: integer);
var x, y: integer;
begin
  { L > 0 and R > 0 }
  left?element(x); right?element(y);
  while (L > 0) and (R > 0) do
    if x < y then
      ADD(x, L, left, output)
    else
      ADD(y, R, right, output);
  { L = 0 or R = 0 }
  while L > 0 do
    ADD(x, L, left, output);
  while R > 0 do
    ADD(y, R, right, output)
  { L = 0 and R = 0 }
end;
```

The procedure inputs the first elements x and y of the left and right sequences. The smaller of these elements, say x, is added to the output sequence and replaced by the next element of the corresponding input sequence (in this case, the left one). The comparison and copying of elements continue until one of the input sequences has been

exhausted. The rest of the other input sequence is then added to the output sequence.

The following procedure adds an element x to the output sequence and replaces it by the next element (if any) from an input sequence of length N

```
procedure ADD(var x, N: integer;
  input, output: sequence);
begin { N > 0 }
  output!element(x); N := N - 1;
  if N > 0 then input?element(x)
end;
```

These sequential procedures are the basis for the parallel mergesort

```
agent MERGESORT(input, output: sequence;
  N: integer);
var left, right: sequence; L, R: integer;
begin
  if N > 1 then
    begin
      L := N div 2; R := N - L;
      +left; +right;
      MERGESORT(input, left, L);
      MERGESORT(input, right, R);
      MERGE(left, right, output, L, R)
    end
  else COPY(input, output)
end;
```

This procedure defines a class of identical processes known as mergesort agents. Each process sorts a sequence of length N, where N is a process parameter.

If N > 1, a mergesort process creates a left and a right channel

```
+left; +right
```

and activates two children

```
MERGESORT(input, left, L);
MERGESORT(input, right, R)
```

which then run in parallel with the parent. Each child sorts half of the N elements and terminates. The parent merges the input from its children into a single output sequence of length N.

If N = 1, a mergesort process behaves as a leaf which copies a single element from an input channel to an output channel.

When a mergesort process reaches the end of its agent

procedure, it is delayed until both of its children have terminated. At this point, the process terminates and the local left and right channels cease to exist.

When these pieces are put together we obtain a single, recursive process (Algorithm 1).

```
type sequence = [element(integer)];

agent MERGESORT(input, output: sequence;
  N: integer);
var left, right: sequence; L, R, x, y: integer;
begin
  if N > 1 then
    begin
      L := N div 2; R := N - L;
      +left; +right;
      MERGESORT(input, left, L);
      MERGESORT(input, right, R);
      left?element(x); right?element(y);
      while (L > 0) and (R > 0) do
        if x < y then
          begin
            output!element(x); L := L - 1;
            if L > 0 then left?element(x)
          end
        else
          begin
            output!element(y); R := R - 1;
            if R > 0 then right?element(y)
          end;
      while L > 0 do
        begin
          output!element(x); L := L - 1;
          if L > 0 then left?element(x)
        end;
      while R > 0 do
        begin
          output!element(y); R := R - 1;
          if R > 0 then right?element(y)
        end
    end
  else
    begin input?element(x); output!element(x) end
end;
```

Algorithm 1. The parallel mergesort.

The branches in the tree do not use the common input channel. They only pass it as a parameter to their children so that it eventually becomes accessible to every leaf. If several leaves attempt to input a single element from the

same channel simultaneously, the channel will interleave these inputs in unpredictable order [4]. However, the order in which the elements are distributed among the leaves is irrelevant since the input sequence is assumed to be unordered.

PERFORMANCE MEASURES

The running time T(P,N) is the time required to sort N elements on a multiprocessor with P processors where

$$1 \leq P \leq N$$

The running time per element is

$$t(P,N) = T(P,N)/N$$

To simplify the analysis we will assume that P and N are powers of two

$$P = 2^p \qquad N = 2^n$$

The speedup

$$S(P,N) = t(1,N)/t(P,N) \qquad (1)$$

defines how much faster the algorithm runs on P processors compared to a single processor.

These performance measures will be expressed in terms of three constants:

- a The average activation time of a process and its output channel.
- b The average termination time of a process and its output channel.
- c The average execution time of two processes for each communication between them.

The running time has two components

$$T(P,N) = Tc(P,N) + Td(P,N)$$

the communication time Tc which is used to transfer and merge elements in the tree, and the decomposition time Td which is used to activate and terminate the processes.

PROCESS COMMUNICATION

A leaf is a sequential process that sorts a single element

by copying it from one channel to another. Its communication time

$$Tc(P,1) = 2c$$

is independent of the number of processors used.

Consider a process tree running on a single processor. The tree consists of a root process and two subtrees. The subtrees sort N/2 elements each. The root inputs and outputs all N elements. According to the definition of c the input time of the root is included in the communication time of the subtrees.

The corresponding recurrence relation

$$Tc(1,N) = 2Tc(1,N/2) + Nc$$

has the solution

$$Tc(1,N) = NTc(1,1) + (N\log N)c$$

where $\log N = n$ is the binary logarithm of N.

By combining these results we obtain

$$tc(1,N) = (\log N + 2)c \qquad (2)$$

which simply says that each element must pass through a channel at each level in the tree.

Now think of a tree running on P processors and assume that the multiprocessor automatically divides the computational load evenly among the processors [5]. The two subtrees of the root are identical processes running in parallel. Each subtree uses the equivalent of P/2 processors to sort N/2 elements in time Tc(P/2,N/2). In addition, the root must output N elements sequentially, that is

$$Tc(P,N) = Tc(P/2,N/2) + Nc$$

This recurrence has the solution

$$Tc(P,N) = Tc(1,N/P) + 2N(1 - 1/P)c$$

From Eq. (2) we have

$$Tc(1,N/P) = (N/P)(\log(N/P) + 2)c$$

Consequently

$$tc(P,N) = \left(2 + \frac{\log(N/P)}{P}\right)c \qquad (3)$$

This shows that the communication time of a process tree

always exceeds the communication time of the root (which is 2c per element). The remaining communications are speeded up by a factor of P (approximately).

We have implicitly assumed that the use of a common input channel for the leaves does not affect the parallel execution of subtrees. If several processors attempt to access this channel simultaneously, each of them is delayed until it obtains exclusive access to the channel for a single communication. If processor conflicts are significant, we must use queuing theory to predict the communication time.

We can make a rough estimate of the probability of such conflicts by viewing the shared channel as a single server with random arrivals and exponential service times [6]. The proportion of the communication time, during which the shared channel is busy, is c/tc(P,N). The probability of more than one processor being ready to use this channel simultaneously is

$$Pdelay(P,N) = (c/tc(P,N))^2$$

In practice the probability of conflict is fairly small as illustrated by the following examples

$$Pdelay(10, 8192) = 0.11 \qquad Pdelay(N,N) = 0.25$$

We will therefore assume that the processors interleave their inputs from the common channel with communications on other channels without experiencing noticeable delays. The performance measurements presented later show that this assumption is realistic for a multiprocessor.

The absence of conflicts has the surprising consequence that the algorithm cannot be speeded up by letting the leaves input the elements in parallel from N channels (instead of one).

The communication speedup is

$$Sc(P,N) = tc(1,N)/tc(P,N)$$

that is

$$Sc(P,N) = \frac{2 + \log N}{2 + \frac{\log(N/P)}{P}} \qquad (4)$$

The upper bound on the speedup is

$$Sc(N,N) = \frac{\log N}{2} + 1 \qquad (5)$$

For N = 8192, we have Sc(N,N) = 7.5. Since Sc(P,N) $\leq$ Sc(N,N) for P $\leq$ N, we conclude that the parallel merge sort is effective only on a multiprocessor with a modest number of processors.

PROCESS DECOMPOSITION

The decomposition time of a (sequential) leaf is the sum of its activation and termination times

$$Td(P,1) = a + b$$

To activate a tree, a single processor must activate and terminate both the root and its two subtrees

$$Td(1,N) = 2Td(1,N/2) + a + b$$

The solution of this equation

$$Td(1,N) = NTd(1,1) + (N - 1)(a + b)$$

shows that the tree consists of N leaves and N - 1 branches.

The above can be rewritten as

$$Td(1,N) = (2N - 1)(a + b)$$

We will use the approximation

$$td(1,N) = 2(a + b) \quad \text{if } N >> 1 \qquad (6)$$

The situation is more subtle when multiple processors are used. We assume that the processors share the memory (although they may use local caches to speed up memory acccess). A lock ensures that a processor has exclusive access to memory while allocating (or reclaiming) space for a process [5].

We will deliberately oversimplify the model by assuming that the tree is activated and terminated in two non-overlapping phases

$$Td(P,N) = Te(P,N) + Tf(P,N)$$

1. In the first phase the processors activate the N - 1 parents (but not the leaves). No communication takes place since the leaves do not exist yet. Since the memory lock serializes the activations we have

$$Te(P,N) = (N - 1)a$$

which can be approximated by

$$te(P,N) = a \quad \text{if } N >> 1$$

2. In the second phase the N leaves are activated and the 2N - 1 processes terminate one by one. So far we have assumed that it is possible to discuss process decomposition and process communication as unrelated events. However at this point we need to make an assumption about how communication influences decomposition.

Process communication begins to take place as soon as the first leaf has been activated. Process activations and terminations are now relatively rare events compared to communications. Only one activation and two terminations take place for every logN + 2 communications in phase two. So it is not very likely that several processors will attempt to lock the memory simultaneously. Since leaf activations and process terminations seldom are delayed by locking conflicts we can analyze them as if they were operations that can be speeded up by using more than one processor. This is the only assumption we need to make about the effect of communication on decomposition.

We will therefore assume that the decomposition of two subtrees is distributed evenly among the processors. When both subtrees have terminated, the corresponding root also terminates. So the decomposition time in phase two is

$$Tf(P,N) = Tf(P/2,N/2) + b$$

or

$$Tf(P,N) = Tf(1,N/P) + (\log P)b$$

Tf(1,N/P) is the overhead of activating N/P leaves and terminating 2N/P - 1 processes sequentially

$$Tf(1,N/P) = (N/P)a + (2N/P - 1)b$$

From these equations we obtain

$$Tf(P,N) = (\log P - 1)b + (N/P)(a + 2b)$$

or approximately

$$tf(P,N) = (a + 2b)/P \quad \text{if } N >> 1$$

Finally we combine these results to obtain the decomposition time per element

$$td(P,N) = a + \frac{a + 2b}{P} \quad \text{if } N >> 1 \qquad (7)$$

which shows that half of the activations and all the terminations are speeded up by a factor of P.

The decomposition speedup is

$$Sd(P,N) = td(1,N)/td(P,N)$$

or

$$Sd(P,N) = \frac{2(a + b)}{a + \frac{a + 2b}{P}} \quad \text{if } N >> 1 \qquad (8)$$

The upper bound on the decomposition speedup is

$$Sd(N,N) = 2(1 + b/a) \quad \text{if } N >> 1 \qquad (9)$$

If we make the reasonable assumption that the termination time of a process does not exceed its activation time, we have

$$Sd(P,N) \leq Sd(N,N) \leq 4 \quad \text{if } b \leq a$$

which is not too encouraging.

COMPLETE MODEL

The running time of the algorithm is the sum of the communication and decomposition times

$$t(P,N) = tc(P,N) + td(P,N)$$

According to Eqs. (2) and (6) the single processor time per element is

$$t(1,N) = 2(a + b) + (\log N + 2)c \qquad (10)$$

From Eqs. (3) and (7) we derive the parallel running time

$$t(P,N) = a + 2c + \frac{a + 2b + \log(N/P)c}{P} \quad \text{if } N >> 1 \qquad (11)$$

The speedup S(P,N) is defined by Eq. (1). The upper bound on the speedup is

$$S(N,N) = \frac{2(a + b) + (\log N + 2)c}{a + 2c} \quad \text{if } N >> 1 \qquad (12)$$

By using Eqs. (5) and (9) we can rewrite this as

$$S(N,N) = \frac{aSd(N,N) + 2cSc(N,N)}{a + 2c}$$

If $\log N \geq 2 + 4b/a$ then $Sd(N,N) \leq Sc(N,N)$ and

$$Sd(N,N) \leq S(N,N) \leq Sc(N,N)$$

that is

$$2(1 + b/a) \leq S(N,N) \leq (\log N)/2 + 1 \qquad (13)$$

THEORY AND PRACTICE

The parallel mergesort written in Joyce runs on an Encore Multimax 320 at Syracuse University. This multiprocessor has 18 processors and a shared memory of 128 Mb. The Joyce compiler generates portable code which is interpreted by a kernel written in assembly language. An elementary operation, such as pushing an integer on a process stack, takes about 5 us.

The activation and termination times of a mergesort process

$$a = 270 \text{ us} \qquad b = 190 \text{ us}$$

were estimated from the program text using the elementary execution times listed in [5].

Most (but not all) of the activation time is serialized by a memory lock. This small error is corrected by replacing the first _a_ in Eq. (11) by the locked part of the activation time

$$a' = 210 \text{ us}$$

The communication time of a mergesort process

$$c = 290 + 10P \text{ us}$$

increases slightly with the number of processors P. This is due to the load balancing mechanism. After each communication the Joyce kernel scans a table of length P and enters the process blocked by the communication in the shortest processor queue.

The curve in Fig. 2 shows the execution times predicted by Eq. (11) for 8192 elements sorted on 1 to 10 processors. The plotted points are execution times measured on the Multimax.

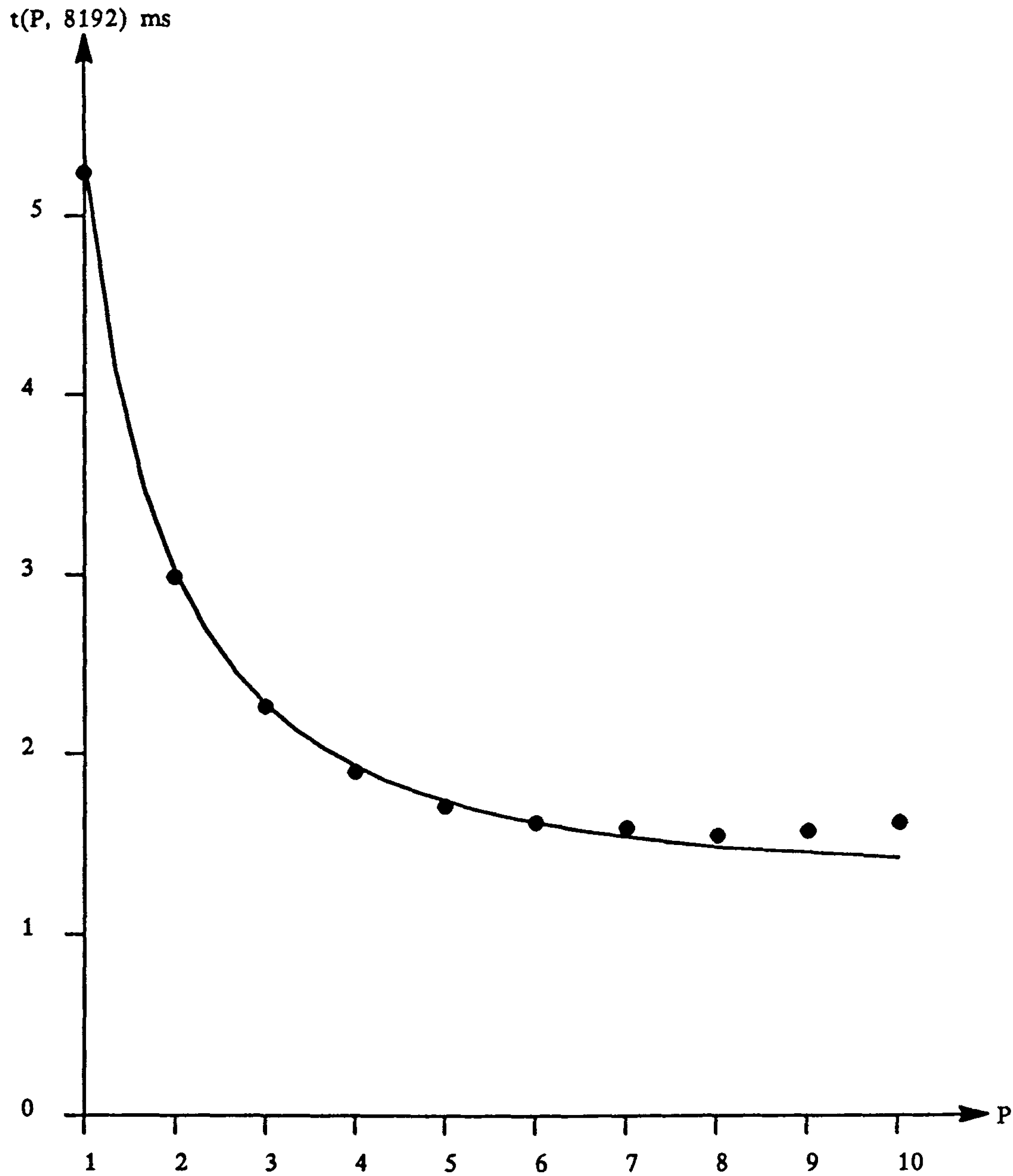

Fig. 2. Parallel running times.

Figure 3 shows the corresponding speedup.

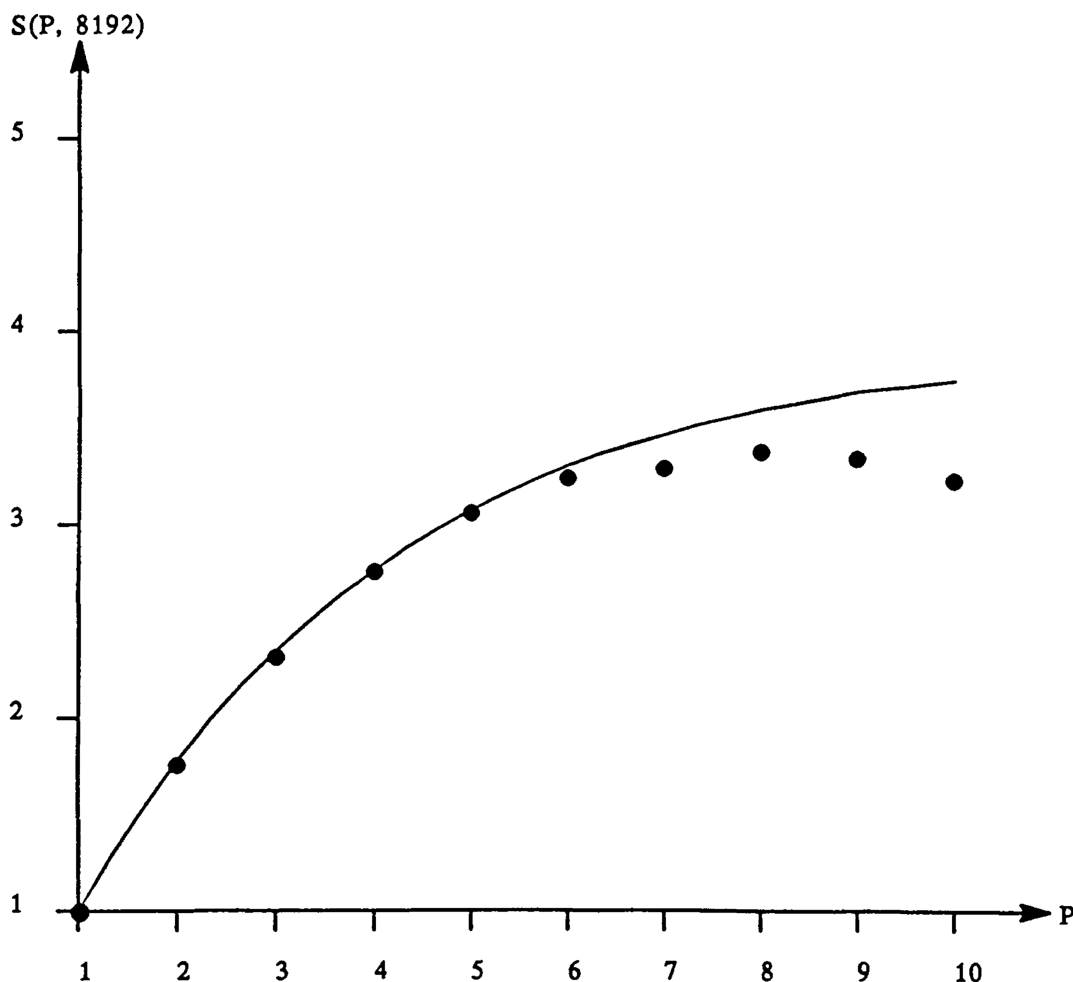

Fig. 3. Parallel speedup.

On a single processor the Joyce program activates 16,383 processes, which exchange 122,880 messages in 43 s. On 10 processors the speedup is 3.2 and the running time is 13 s (or 1.6 ms/element).

Similar agreement between theory and practice was obtained for 1024, 2048 and 4192 elements.

CONCLUSION

We have described an exact performance model of the parallel mergesort. This algorithm sorts N elements by spawning almost 2N processes which perform roughly NlogN communications. The analytical model accurately predicts the running time of the algorithm on a multiprocessor.

The performance of this highly parallel algorithm is

dominated by the overhead of process activation, termination, and communication. On a multiprocessor with shared memory these operations take about the same amounts of time. In this final argument we will assume that they take exactly one unit of time each, that is

$$a = b = c = 1$$

In that case the minimum running time of the mergesort is approximately

$$T(N,N) = 3N \quad \text{if } N >> 1$$

according to Eq. (11). This result suggests that it takes N processors to sort N elements in O(N) time. However this view of the parallel performance is too pessimistic.

If we use

$$P = (\log N + 3)/3$$

processors only, we obtain the following approximate running time from Eq. (11)

$$T(P,N) = 6N \quad \text{if } N >> P/8$$

In other words the parallel mergesort needs only O(logN) processors to sort N elements in O(N) time. (The exact sorting time is about twice the minimum time.)

For example a multicomputer that sorts a million elements on a million processors runs only twice as fast as a multiprocessor that sorts the same elements on eight processors.

We conclude that the parallel mergesort performs well on a small multiprocessor but is unable to utilize a large multicomputer efficiently.

The main purpose of the paper has been to illustrate the analysis of an algorithm with dynamic parallelism.

ACKNOWLEDGEMENTS

The paper has been improved by valuable comments from Nawal Copty, Jonathan Greenfield, Anand Rangachari, Sanjay Ranka and Tapas Som.

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] D. E. Knuth, _The Art of Computer Programming. Vol. 3. Sorting and Searching_. Addison-Wesley, Reading, MA, 1973.

[2] J. T. Robinson, "Some analysis techniques for asynchronous multiprocessor algorithms," _IEEE Trans. on Software Engineering_, vol. 5, pp. 24-31, 1979.

[3] D. J. Evans and N. Y. Yousif, "The parallel neighbour sort and 2-way merge algorithm," _Parallel Computing_, vol. 3, pp. 85-90, 1986.

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

[5] P. Brinch Hansen, "A multiprocessor implementation of Joyce," _Software - Practice and Experience_, vol. 19, pp. 579-592, 1989.

[6] L. Kleinrock, _Queueing Systems. Vol. 1. Theory_. John Wiley & Sons, New York, NY, 1976.