12-9-1994

Methodology for Accurate Speedup Prediction

Aruna Chittor
Portland State University

THESIS APPROVAL

The abstract and thesis of Aruna Chittor for the Master of Science degree in Electrical and Computer Engineering were presented December 9, 1994 and accepted by the thesis committee and the department.

COMMITTEE APPROVALS: [redacted], Chair
Michael A. Driscoll

[redacted]
Marek A. Perkowski

[redacted]
Jingke Li
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL: [illegible]
Rolf Schaumann, Chair
Department of Electrical Engineering

* *

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 21 March 1995

ABSTRACT

An abstract of the thesis of Aruna Chittor for the Master of Science in Electrical and Computer Engineering presented December 9, 1994.

Title : Methodology For Accurate Speedup Prediction

The effective use of computational resources requires a good understanding of parallel architectures and algorithms. The effect of the parallel architecture and also the parallel application on the performance of the parallel systems becomes more complex with increasing numbers of processors. We will address this issue in this thesis, and develop a methodology to predict the overall execution time of a parallel application as a function of the system and problem size by combining simple analysis with a few experimental results. We show that runtimes and speedup can be predicted more accurately by analyzing the functional forms of the sequential and parallel times of critical code segments of a parallel application that affect the speedup of a parallel program. We then combine the functional forms to model the runtime of a parallel application. A small set of experiments are sufficient to get a good estimate of the coefficients for the runtime models obtained. Speedup can then be derived for any case from the runtime model. We also analyze the effect of the I/O on runtimes in memory bounded parallel systems, and how speedup is affected by communication and I/O.

Throughout the thesis we use the bitonic merge sort as a typical realistic parallel application to illustrate our methodology. Several variations of the sorting algorithm (such as problem size greater than or equal to the system size, unlimited or limited buffer size) suitable for a wide range of problem sizes are implemented in two parallel environments and the speedups for them are measured and compared with different speedup predictions. We have conducted numerous experiments using Multi and PVM to empirically study speedup for the different realistic implementations of the bitonic

merge sort. The results show how well the various models predicted speedup, and that our methodology can predict speedup accurately for a given parallel application. One interesting value from a speedup curve is the roll-off point - the system size beyond which speedup actually decreases when the number of processors is increased. Results show that simple theoretic models predicted roll-off point to be higher than the actual values, where as our methodology predicted it to be less than and closer to the actual values. The predictions by our methodology also compare well with the speedup estimates provided by the Multi tool.

METHODOLOGY FOR ACCURATE SPEEDUP PREDICTION

by

ARUNA CHITTOR

A thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1995

ACKNOWLEDGEMENTS

Working on this thesis was challenging, interesting and a great learning experience. Special thanks to my thesis advisor Dr. Michael Driscoll for his constant guidance, encouragement and support throughout the thesis.

I am grateful to Dr. Marek Perkowski and Dr. Jingke Li for serving on my thesis committee, for many useful comments and suggestions that helped me to express my work clearly and concisely.

I would like to thank Chandana and B.J. Porcella for helping me learn the tools during the thesis.

I would also like to thank my husband Suresh for providing the motivational guidance and support. I thank my daughter Achala for cooperating with me to work on this thesis.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

With the enormous progress in the technology of multiprocessor systems, a tremendous potential exists for achieving high gain in speed for applications running on parallel systems. However, high gain in speed can be realized for any parallel application only if the available computational resources are used effectively. The effective use of the computational resources requires a good understanding of the parallel architectures and algorithms. The effect of the parallel architecture and also the parallel application on the performance of the parallel systems becomes more complex with increasing numbers of processors. We will address this issue in this thesis, and develop a methodology that will predict the overall execution time as a function of the system and problem size, thus showing the effect of the number of processors on the performance of parallel applications.

Significant effort has been made in the last decade to understand how the characteristics of parallel architectures and algorithms change with the number of processors used. The effect of the increasing number of processors on the architectural and algorithmic characteristics is often referred to as the scalability of the architecture or application. Models have been proposed that can quantify the scalability of a specific algorithm when it is run on a parallel computer. Such models help in understanding the influence of various factors, such as the inherent parallelism of the application, the computation power, the memory capacity of the parallel computing system and the performance of the communication network, on the performance of the applications. The models also help in predicting and optimizing the gain in computational rates as we scale the system size.

The most commonly used metric to characterize the gain in speed is the ratio of the execution time of the sequential version of a program in a uniprocessor environment to the execution time of its parallel counterpart in a multiprocessing environment (say, with *n* processors). This ratio yields a dimensionless quantity that can be interpreted as the *speedup*.

The actual speedup achieved on a given multiprocessor system depends on the nature of the application being run. In practice, there are several factors that may prevent one from attaining an ideal speedup of *n* on an *n*-processor system. First, most applications contain code segments that are inherently sequential in nature and hence must be run in a serial fashion. Second, the application segments that run in parallel on different processors often have to interact with each other, either to exchange data or to synchronize their executions. These interactions introduce an overhead component into the total runtime of the application. Third, a number of hardware resources such as memory, communication channels, I/O channels, etc. are shared among the processes executing in parallel. Each such shared resource can only be accessed by a single process at a time in a mutually exclusive fashion. Thus, contention for shared resources and the consequent queueing delay introduces yet another overhead component into the total runtime of the application[12]. Finally, the overall computation load has to be evenly shared by the different processors. This load balancing is also difficult to achieve.

Amdahl's law and *Gustafson's scaled speedup* are the two well known speedup models. Both of these speedup formulations use a single parameter, the sequential portion of a parallel algorithm, to characterize an application. They are simple and give much insight into the bounds on speedup when the system size is scaled. *Amdahl's law*[8] assumes a fixed problem size and models the overall execution time as a function of the number of processors. The model shows that the overall execution time is bounded by the sequential time of an application, which is assumed to be a constant. Speedup is thus bounded by 1/f where f is the sequential portion of the overall execution time. The asymptotic behavior of overall execution time predicted by *Amdahl's law* suggests that massively parallel processing may not gain high speedup. The simple speedup formulation by *Amdahl* gave good insight into the limitations on parallel systems, and could explain the observed speedup behavior of many parallel applications.

In practice, a system with more processors is not only used to reduce the time for a given problem size, but to solve a larger problem in a given time. So, *Amdahl's* simplified model may not give a fair model for speedup. Hence, *Gustafson*[9,10] approached the speedup formulation from a slightly different point of view. Instead of fixing the problem size and then studying the improvement in overall execution time as a function of the number of processors, he fixed the response time and then studied speedup by determining how large a problem could be solved in this time by a system of given size. The speedup is then computed as a ratio of the time taken to run the scaled problem on a single processor system to the time to run the problem on a parallel system with P processors. Instead of fixing the response time as the scalability constraint, *Ni* and *Sun*[3] have considered the finite memory size on individual nodes of a parallel system as the scalability constraint. They scaled the problem size to fill the entire memory of the system for a given system size. The speedup is then computed by finding the computation times for the scaled problem size on a single processor and on the parallel system.

The speedup formulations described above were simple, and could explain the observed speedup values for various parallel applications. They also helped in understanding the effect of various architectural and algorithmic characteristics on speedup when the number of processors is increased. The models still lack in accuracy and have several limitations, which have prompted other researchers to improve these models. *Scherson* and *Corbett*[11] accounted for communication overheads by adding a fraction of parallel time to the sequential time. In their recent studies, *Driscoll* and *Daasch*[2] further observe that the sequential portion of the algorithm is not a constant but is also a function of the number of processors. In their speedup formulation, they retain the separation of execution time into parallel and sequential time as proposed by others, but assume the sequential and parallel times to be functions of problem and system size. They also show how to include the impact of large number of processors on the time required for initialization, synchronization, communication, and input/output. Experiments were done using simple computational models to support the new formulation.

The accuracy of the speedup models can be further improved by analyzing the critical segments of a parallel application (which account for most of the computation times) to determine how they vary with the system and problem size. The functional forms for typical code segments that contribute to the execution time of a realistic parallel program can be a complex function of system size and problem size, which may not be captured by the simple models for speedup proposed by others. These functional forms can, for example, take linear/logarithmic dependencies on the machine size and problem size. The analysis of these functional forms can result in more accurate predictions of speedup of a parallel application for a given system and problem size. More accurate prediction will help in determining the right problem size to achieve optimal speedup for a given system size. It may also be possible to predict the execution time of parallel applications on multiple platforms based on characterizations of the algorithmic components of a parallel application and the characteristics of the parallel architecture of the platform under consideration.

In this MS thesis, we further explore the problem of accurate speedup prediction for real applications. We develop a methodology to predict the overall execution time as a function of the system and problem size by combining simple analysis with a few experimental results. We show that runtimes and speedup can be predicted more accurately by analyzing the functional forms of the sequential and parallel times of code segments. The functional forms of critical code segments of a parallel application can be analyzed and expressed as functions of the problem size N and the system size P. We then combine the functional forms to model the runtime of a parallel application on a specific system as a linear function of the terms in N and P with unknown constant coefficients. A small set of experiments are sufficient to get a good estimate of the coefficients, and thus model the overall runtime as a function of N and P. Speedup can then be derived for any case of the parallel application from the runtime model. Experiments are also used to measure the execution time for several common contributors to sequential time such as barriers, spinlocks, and process creation. The results from the experiments can then be used to model how the sequential time scales with P on a given system. Our methodology is also used to analyze how speedup is affected by communication and I/O in memory bounded parallel systems[3].

Throughout the thesis we use the bitonic merge sort as a typical realistic parallel application to illustrate our methodology. Several variations of the sorting algorithm (such as problem size greater than or equal to the system size, unlimited or limited buffer size) suitable for a wide range of problem sizes are implemented in two parallel environments and the speedups for them are measured and compared with different speedup predictions. We have conducted numerous experiments using Multi[1] and PVM[4] to empirically study speedup for the different realistic implementations of the bitonic merge sort.

The results show how well the various models predicted speedup, and that our methodology can predict speedup reasonably accurately for a given parallel application. The predictions also compare well with the speedup estimates provided by the tool, Multi.

This thesis is organized as follows. In Chapter 2 we will introduce some preliminary knowledge and terminologies. The bitonic merge sort algorithm and the parallel implementations are discussed in Chapter 3. The methodology to predict the speedup will be discussed in Chapter 4. The experiments and results obtained under Multi environment are presented in Chapter 5. Chapter 6 describes the experiments and results under the PVM environment. Conclusions and comments are given in Chapter 7.

CHAPTER 2

Background

2.1 Parallel Computer Systems

The basic concept behind the parallel computer is to simply have more than one processor connected by a network to operate concurrently. The key feature that makes it a "parallel" computer is that all the processors are capable of operating at the same time. There are currently three major class of parallel computers: Shared memory, also called *multiprocessors*, message passing, also called *multicomputers,* and shared distributed memory.

Shared memory multiprocessors constitute an important class of parallel processing systems. The architecture of machines representing this class is characterized by a number of processing elements connected to a comparable number of memory elements through an interconnection network. The interconnection medium selected governs the communication bandwidth of the system and hence determines the maximum speed at which processes running on different processors can interact[12]. Some of the well known multiprocessor systems are Convex, BBN's Butterfly, Cray Y-MP and Sequent's Symmetry.

Multicomputers are distributed-memory multiprocessors. They are organized as an ensemble of individual programmable computers, called nodes, and communicate through an interprocessor communication network. The memory is distributed and a portion is associated with each node. When the number of processors increases, the memory capacity also increases[3]. Some of the most powerful and large scale parallel computing systems such as NCUBE, Intel's Paragon and iPSC series, Cray T30 and CM-5 of Thinking Machines belong to the class of multicomputers.

The current trend in parallel architectures seem to be towards distributed shared memory systems [14][15]. These systems have physically distributed memory as in multicomputers. A user process running on a single node can access memory on any other node by utilizing the high speed communication network hardware. When there is a cache miss and the processor requests the required cache block from memory, the network interface component (NI) will respond if the requested address is mapped to a remote node, instead of the memory controller. The NI will then automatically send a request to the remote node on the high-speed interconnection network for the required memory block. The NI on the remote node will receive the request, access the memory and send back the requested memory block. Upon receiving it, the NI will return the data to the processor. The entire process is handled automatically by the hardware. The only difference noticed by the processor when the address it is trying to access (on a cache miss) maps to a remote node's primary memory instead of the local node's primary memory, is increased delay. (Latency to access memory on a remote node on current distributed shared memory systems tends to be an order of magnitude slower than latency to access local memory). Various protocols and architectures have been proposed to maintain the coherency of shared memory. Thus, the OS can support a shared memory model for individual user processes even though the memory is physically distributed. Distributed shared memory systems are gaining popularity as they can scale to hundreds of nodes, but still provide a programming model that is simpler to understand. Currently available commercial systems are KSR-1, NEC SX/3, with several academic architectures such as Stanford DASH system [14] and MIT's Alewife[15]. Scalable Coherent Interconnect (SCI) standard is another IEEE standard proposed for implementing distributed shared memory systems.

2.2 Sorting Algorithms

Sorting is one of the most common activities in parallel algorithms. There are several parallel sorting algorithms, namely insertion sort, quick sort, rank sort, merge sort, bucket sort, bitonic merge sort, etc.[7]. Simple sequential algorithms require $O(N^2)$ time for the worst case, and parallel versions of them still take O(N) time with *N* processors. The best* sequential algorithms take O(N log N) time**. However, their parallel versions may not do well as they are limited by the sequential parts. For example, quick sort, which is based on a divide and conquer strategy, becomes more parallel as the problem gets divided into smaller problems. However, the first step in which a single sorting problem is divided into two problems is hard to parallelize. On the other hand, if we consider merge sort, we find that it can be parallelized more easily in the beginning when we merge many small lists. However, the last step, in which two *N*/2 sorted lists are to be merged into a single sorted list of length *N* is difficult to parallelize. The parallel rank sort takes O(N) time to find the position of each element in the sorted list. Since finding the position can be done in parallel, the total execution time of rank sort will be O(N). The performance of the bucket sort is dependent on the evenness of the distribution phase, where different portions of the array of elements are assigned to different processors. The sorting of the individual buckets is carried out in parallel.

The bitonic merge sort, described in detail in Chapter 3, takes $O(N \log^2 N)$ time on 1 processor, and hence is not the best sequential algorithm. However, it is more easily parallelizable. A parallel version of bitonic merge sort takes only $O(\log^2 N)$ time to sort *N* elements using *N* processors, with a shared memory model or ignoring the communication delays. It is one of the best parallel algorithms for sorting, and is used as an example of a real application throughout the reminder of this thesis.

* By *best* we mean minimum execution time.

** All logarithms in this thesis are to the base 2.

CHAPTER 3

Bitonic Merge Sort

The bitonic merge sort[1] is a compare-exchange type of sort, very similar to a merge sort. It is more suitable for parallelization, as it has $O(\log^2 N)$ basic steps, called binary split, where N is the total number of elements in the list, and each binary split operation involves $N/2$ independent compare and exchange operations. Sorting is accomplished by repeatedly merging small, partially sorted sublists, called bitonic lists, in pairs to form bitonic lists of twice the length. A bitonic list has at most one local maximum and/or one local minimum, where a local maximum (minimum) is an element that has a value higher (lower) than that of either of its neighbors. A list of two elements is a bitonic list, and so any list that needs to be sorted can be thought of as consisting of $N/2$ bitonic lists of length 2. Two bitonic lists of length K can be combined into a single bitonic list of length $2K$ by sorting the first bitonic list in ascending order and the second in descending order. The two bitonic lists can be sorted independently, and sorting of each bitonic list of length K involves $\log K$ basic steps, and each of these basic steps involve multiple binary split operations done in parallel on various parts of the list (described in detail later). Thus, the binary split operation on a bitonic list forms the heart of the bitonic merge sort algorithm.

In a binary split operation on a list of N elements, each element in the first half of the list is compared with the element at the same relative position in the second half. If the element in the first half has greater value than the corresponding element in the second half, then the two elements are swapped. It can be shown that when we apply the binary split operation to a bitonic list, the resulting list will have the following properties [1].

```
(* Sort a bitonic list of length N in ascending or descending order,
    by repeatedly applying (log N steps) binary split operation *)
Procedure bitonicsort (blist, N, order)
begin
    forall k:= 0 to N-1 do
    begin
        for i: = log N-1 down to 0 do
        begin
            (* determine the index of the partner element k1, that needs to be
               compared and swapped with element k *)
            if odd(k div $2^i$) then k1:= k - $2^i$ else k1:= k + $2^i$ ;
            (*ASC=Ascending, DESC=Descending*)
            (* compare elements at k and k1*)
            send(k1, blist[k]) ;  (* send copy of my element to my partner *)
            my_part_val := recv(k1) ; (* receive my partner's value = blist[k1] *)
            if ((order = ASC and k < k1) or (order = DESC and k > k1)) then
                (* replace k th element by the minimum of the two *)
                if (my_part_val < blist[k]) then blist[k] := my_part_val ;
            else
                (* replace k th element by the maximum of the two *)
                if (my_part_val > blist[k]) then blist[k]:= my_part_val ;
            barrier ; (* wait till all (N) tasks reach this statement *)
        end ;(* for *)
    end ;(* forall *)
end.(* procedure *)
```

FIGURE 1. Parallel algorithm for sorting a bitonic list

- each element in the first half of the list is less than every element in the second half
- the first half and the second half of the list are each a bitonic list of length *N*/2.

Hence, by recursively applying the binary split to the smaller bitonic lists, we will eventually sort the original bitonic list. The binary split operation can be done in parallel in O(1) time, if we have one processor for each element. A bitonic list can be sorted in parallel in O(log N) time as it involves log *N* steps of binary split operations. The algorithm is shown in Figure 1. The example given in Figure 2 illustrates clearly how the algorithm works and shows the steps involved in sorting a bitonic list of length 8 in ascending order where each step involves one or more binary split operations done in parallel.

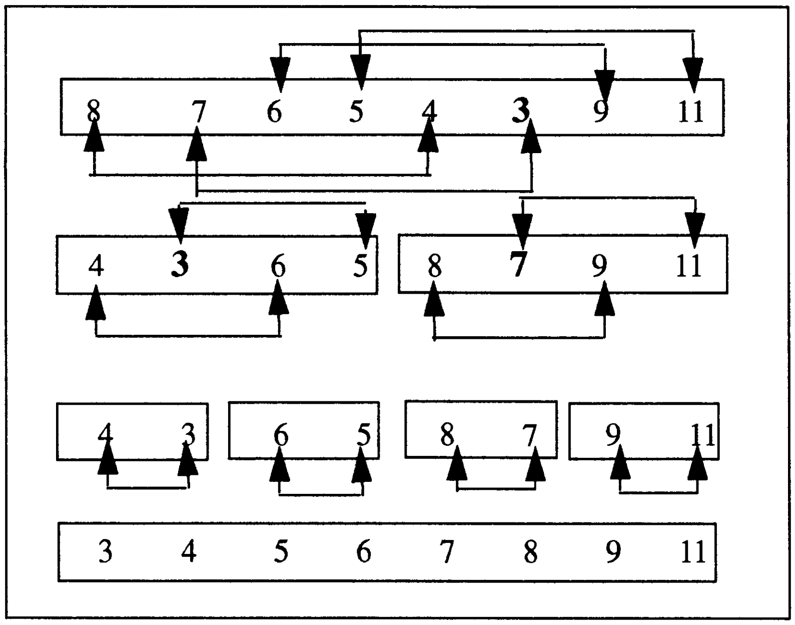


FIGURE 2. An example to illustrate the steps in sorting a bitonic list

The given list is a bitonic list with 3 as the local minima. The first step is a single binary split operation done on the entire list. The value of each element in the first half of the list is compared with the value of the corresponding element in the second half of the list. If the element in the first half is larger than the corresponding element in the second half, then the two elements are exchanged. For example, the value of the first element, 8 is compared with the value of the fifth element, 4. Since 8 > 4, we exchange the two elements. In the first step, the first and second element (values 8 and 7) were exchanged with the fifth and sixth elements (values 4 and 3) respectively, but the third and the fourth elements (values 6 and 5) were not exchanged with the seventh and eighth elements (values 9 and 11). At the completion of the first step we have two bitonic lists of length 4, where 3 and 7 are the local minima, with all the elements in the first bitonic list (first half of the list) less than the elements in the second bitonic list (second half of the list).

In the second step, the binary split operation is applied to each of the two bitonic lists of length 4 in parallel. After doing the required exchange operations, we end up with four bitonic lists of length 2 with every element in each of the bitonic lists less than any of the elements in the bitonic lists to its right. In the third and the final step, we apply binary split operations to the four bitonic lists in parallel to get the entire list sorted in ascending order.

An unsorted list can be thought of as made up of bitonic lists of length two. As explained earlier, the algorithm in Figure 1 can be used to repeatedly merge adjacent sublists until the whole list is sorted. This idea leads us to the bitonic merge sort. An

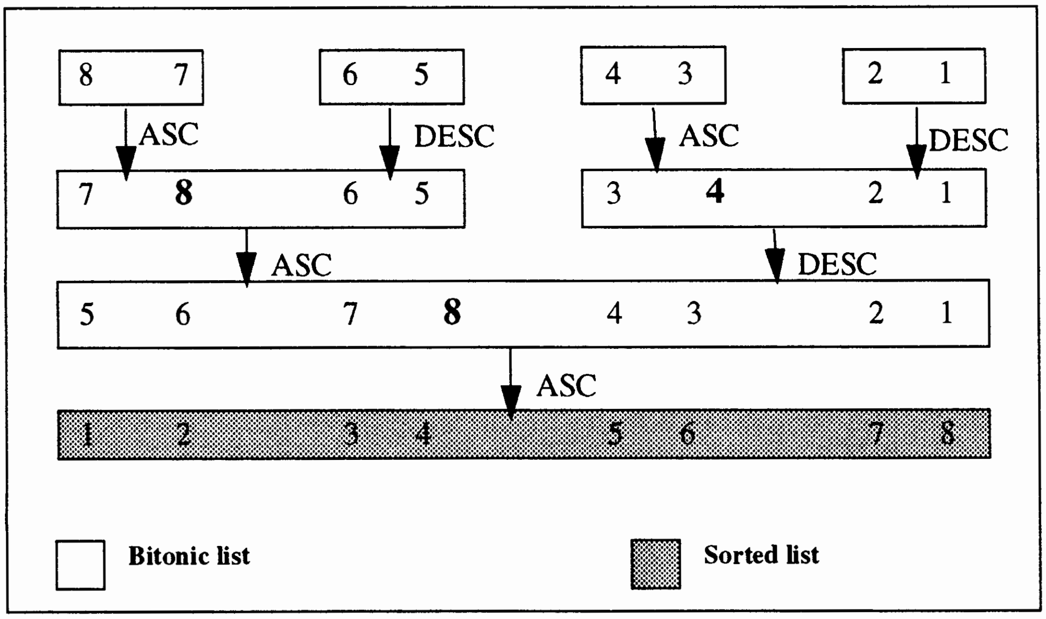


FIGURE 3. Steps involved in merging bitonic lists to sort a list of 8 elements where ASC is Ascending order and DESC is descending order

example is shown in Figure 3 to illustrate the steps involved in sorting a list of 8 elements in ascending order. In this example, we have 4 bitonic lists at the start. In the first step we sort the odd numbered bitonic lists in ascending order and the even numbered bitonic lists in descending order in parallel using the algorithm in Figure 1. At the completion of the first step we will have 2 bitonic lists of length 4, where 8 and 4 are the local maxima. We repeat step 1 for each of the bitonic lists and end up with a bitonic list of length 8. We will then sort the single bitonic list with 8 as the local maxima, in required order using the steps in Figure 2. The algorithm is formally described in Figure 4. We now describe the various implementations of this algorithm in detail.

```
(* Sort a list of length N in ascending or descending order by repeatedly merging adjacent bitonic lists to form longer bitonic lists.
   Bitonic lists are merged by sorting the first list in ascending order and the second list in descending order. *)
Procedure sort(list, N, order)
    begin
    suborder: integer ;
        for i: = 1 to log N do
            begin
             (* Merge adjacent bitonic lists of length $2^i$ to form a bitonic
                 list of length $2^{i+1}$*)
             for j:= 0 to N-$2^i$ step $2^i$ do (* sort this list starting at index j *)
                begin
                 (* determine the order for the sort *)
                 (* ASC =Ascending, DESC=Descending *)
                 if ($2^i$ = N) then suborder:= order
                 else if (odd(j div $2^i$)) then suborder:= DESC
                 else suborder:= ASC ;
                 bitonicsort(blist[j:j+ $2^i$-1],$2^i$ ,order) ;
                end;(* for j *)
           end;(* for i *)
    end. (* procedure *)
```

FIGURE 4. Algorithm to sort any list by using the algorithm to sort bitonic lists

3.1 Implementation when $N = P$

We implemented the algorithm in Figure 1 and Figure 4 for a general parallel architecture (shared or distributed memory) by the data-parallel function blsort and bmergesort given in Figure 5 and Figure 6 respectively. We made the following modifications to the algorithms given earlier to obtain a flexible and efficient implementation.

- The creation of the parallel tasks (denoted by the FORALL construct in Figure 1) is moved to the top level before the main loop in bmergesort function, so that this costly operation is done less often (in this case only once).
- Each process or task handles the element at the same position (= start + my_id). The value of the arguments computed in bmergesort and passed to blsort each time determines the bitonic list to which the element belongs for a given iteration. Even though the same code of blsort is executed by each process, the arguments determine the sequence of compare and exchange operations done by any process. The arguments also make blsort general and flexible data parallel code.
- The communication of the elements inside the for loop of the blsort function is used to exchange elements, and also to synchronize the individual processes implicitly. No barrier operations are done inside the blsort function. Processes are only synchronized at the end of the top level *for* loop in bmergesort function.
- We assume that N and P will be powers of 2 to keep the functions simple.

The functions were successfully implemented both in Multi-Pascal and PVM-C.

3.2 Implementation when $N > P$

We further generalized the algorithms for the case when the number of processes is less than or equal to the number of elements in the list, i.e., $N > P$, by doing the following modifications to the algorithms given in Figure 1 and Figure 4.

```
(* Sort a bitonic list in a given order, by repeatedly participating in binary split
 operations)
Procedure blsort(my_id,start,len,order)
   (* my_id : is the process identification number *)
   (* start   : is the index of the first element of the bitonic list *)
   (* len     : is the length of the bitonic list to be sorted *)
   (* order : is the order for sorting : ascending or descending *)
begin
 my_val := data[start+my_id] ; (* assuming data to be sorted is available in
   shared memory, otherwise it has to be received via a message recv call *)
 for pd := len/2 to 1 step pd/2 do (* pd is the distance to my partner *)
 begin
    (* Determine my_partner_pos - to left or right of my position ? *)
    my_partition_no :=(my_id -(my_id mod pd) ) div pd ) ;
    my_partition_dir := my_part_no mod 2;
     (* Determine partner process id *)
   if my_partition_dir=0 then
       my_partner_pos:=my_id+pd
   else
       my_partner_pos:=my_id-pd;
   (* Send copy of my_element's value to my partner *)
   send(my_partner_pos,my_val,1) ;
   (* Receive copy of my partner's element's value *)
   receive(my_partner_pos, my_part_val,1) ;
   if dir = my_partition_dir then begin
       (* Retain the smaller of the two items *)
       if (my_val > my_part_val) then my_val := my_part_val ;
       (* Retain the larger of the two items *)
       if (my_val < my_part_val) then my_val := my_part_val ;
   end;(* if *)
  end (* for loop *)
  data[start+my_id] := my_val ;
end. (* end procedure *)
```

FIGURE 5. BLSORT - Implementation to sort bitonic lists when *N* = *P*

```
(* Sort a list of length N in ascending or descending order by repeatedly merging
adjacent bitonic lists*)
Procedure bmergesort(list, N,sortorder)
(* sortorder: order in which the bitonic list should be sorted *)
  forall k := 0 to N-1 do
  (* Each process my_id runs this code*)
    for bl:=2 to N/2 step bl*2 do
      (* bl is the length of the Bitonic list*)
      (* Determine the bitonic list to which the element k belongs in this
         iteration, and the relative position of element k in the list,
         and call blsort to sort the bitonic list in required order *)
      start:=k-k mod bl;  len := bl ; (* start and length of the bitonic list *)
      my_id:=k mod bl;  (* relative position of element k in the bitonic list *)
      order :=(start div bl) mod 2; (* order in which bitonic list is to be sorted *)
      blsort(start,my_id,len,order) ;
      barrier ;
      (* wait till all processes finish sorting bitonic lists for this iteration *)
    end ; (* for loop *)
   blsort(0, k, N, sortorder)
 end ; (* forall loop *)
end. (* end procedure *)
```

FIGURE 6. BMERGESORT - Implementation to sort any list when $N = P$

1. Process i ($0 \le i < P$) is assigned a sublist of (N/P) elements (instead of 1) starting from element i * (N/P). This will ensure that each process will do an equal amount of work in each iteration.

2. Before starting the bitonic merge sort, all processes sort the sublists assigned to them using the sequential version of the functions blsort and bmergesort. The sequential version is obtained by changing the *forall* to *for*, and by sequentially executing the inner loops for all possible my_id values (= N/P). All processes run in parallel on their partitions, so this step of the bitonic merge sort takes O((N/

```
(* Sort a bitonic list in a given order, by repeatedly participating in binary split
operations)
Procedure blsort_ngtp (start, group, my_start, len, order) begin
   (* group : number of elements handled by each process *)
   (* my_start : position of the first element of the group assigned to me*)
   (* all other arguments are same as for blsort *)
for i := 0 to group-1 do my_val[i] := data[start+my_start+i] ;
for pd := len/2 to group step pd/2 do (* pd is the distance to my partner *)
begin
    (* Determine my partner direction & position *)
    my_partition_no :=(my_start -(my_start mod pd) ) div pd ) ;
    my_partition_dir := my_part_no mod 2;
     (* Determine partner process id *)
   if my_partition_dir=0 then my_partner_pos:=my_start+pd
   else my_partner_pos := my_start-pd;
   my_part_process_no: = (start + my_partner_pos) div group ;
   (* Send copy of my_element's values *)
   send(my_part_process_no, my_val, group) ;
   (* Receive copy of my partner's element's value *)
   receive(my_part_process_no, my_part_val,group) ;
   (* Compare and exchange value if necessary *)
   if dir = my_partner_dir then
        for i: = 0 to group do   (* Retain smaller of the two values *)
            if my_val[i] > my_part_val[i] then my_val[i]: = my_partval[i] ;
   else
        for i: = 0 to group do (* Retain the larger of the two items *)
            if my_val[i] < my_part_val[i] then my_val[i]: = my_partval[i] ;
 end; (* for pd loop *)
 for i := 0 to group-1 do data[start+my_start+i] := my_val[i] ;
 blsort_seq(start + my_start,group,order) ;
 barrier ;
end.(* procedure *)
```

FIGURE 7. BLSORT_NGTP - Implementation to sort bitonic lists when $N > P$

P)log^2(N/P)) time.

3. During each compare and exchange step in the innermost loop of blsort, a process compares and swaps (*N/P*) elements instead of just one element. To keep the algorithms simple, it was designed to run only when N and P are powers of 2. The loop is stopped when *pd* becomes less than or equal to *N/P*, and the remaining steps (*pd* = *N/P*, *N/2P*, ... ,1) are executed sequentially by the individual processes as all the elements will be available locally. Here, blsort_seq is the sequential version of the algorithm blsort. The algorithms with the above modifications for $N > P$ are given in Figure 7 and Figure 8.

(Comment : Multi supports a feature called group that allows multiple Multi processes to be run on the same processor. We tried to use the group feature of Multi to run *N/P* processes on each processor to extend the functions in the previous section (that requires $N = P$) to the general case ($N > P$). However, the scheduling policy used in Multi seems to be non pre-emptive, and hence the system will deadlock on the first innermost communication operation).

3.3 Implementation when $N \geq P$ and limited buffer size

Primary memory available in each node of a parallel architecture is limited. This also affects the performance of a parallel application, and needs to be considered in predicting speedup. The limited memory will require access to disks. To consider the effect of limited memory on speedup, we modified our algorithms further to use a limited buffer size.

The communication time for disk I/O can be quite significant, and I/O has to be handled properly to achieve reasonable performance and speedup as we scale the system size. One effective technique is to overlap I/O communication time with computation by pre-fetching the required data blocks, and post-dumping the results. The technique is illustrated in Figure 9. In this example, a large number of blocks need to be processed by a single process. The technique shows that in any iteration in which block i is being processed, block i+1 will be pre-fetching the required data, and block i-1 (that

```
(* Sort a list of length N in ascending or descending order by repeatedly merging
adjacent bitonic lists, by using P processors where N > P *)
Procedure bmergesort_ngtp (P, N, order)
 begin
 forall k := 0 to P-1 do
   begin
   group = N div P ; groupstart = k * group ;
   (* sort the group of elements assigned to me in the right order *)
   bmergesort_seq(groupstart,group,(groupstart div group) mod 2) ;
   barrier ; (* wait till all processes finish sorting their groups *)
   for bl:=2*group to N/2 step bl*2 do
     begin
       (* Determine the bitonic list the to which the element k belongs in this
          iteration, and call blsort to sort the bitonic list in required order *)
       start := groupstart - groupstart mod bl;   len := bl ;
          (* start/length of the bitonic list *)
      my_id := groupstart mod bl;
      (* relative position of element k in the bitonic list *)
      order :=(start div bl) mod 2; (* order in which bitonic list is to be sorted *)
      blsort(start,group,my_id,len,order) ;
      barrier ; (* wait till all processes finish sorting bitonic lists for this iteration *)
     end (* for loop *)
 end (* forall loop *)
end.(* procedure *)
```

FIGURE 8. BMERGESORT_NGTP - Implementation to sort any list when *N* > *P*

has been already processed) will be dumped back to the disk*. If communication of two blocks can be completed in less than the time taken to process a block, then I/O

* If communication overheads (delay to initiate a send or receive) is very high for the system under consideration, then this approach may not be the best. In such a case, we may want to use just one request to fill the entire buffer, process it and send it back. Even though communication and computation are now serialized, there will be fewer send/receive calls and less overhead.

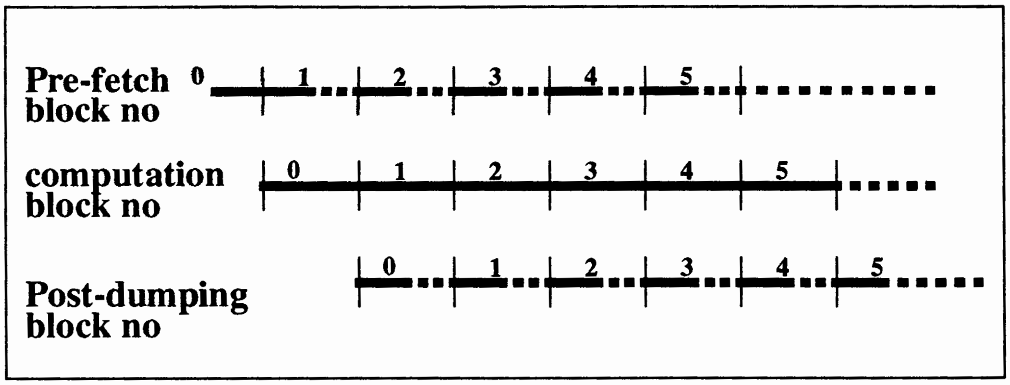


FIGURE 9. Overlapping I/O communication with computation

communication time can be hidden behind computation. The only overhead will be the time to initiate read and write of blocks to disk. With communication times no longer critical, it may be possible to achieve good speedups even with smaller data granularity. The total buffer size required is three times the block size or block size will be 1/3 of the buffer size available.

We used the above technique to extend the blsort_ngtp and bmergesort_ngtp to work with limited size buffers. We divided the available memory into six buffers of equal size, say *B* elements. The *N* elements to be sorted are divided into $L = N/B$ blocks, so each block of elements can be held in a single buffer. The following modifications were done to handle the case when $N >> B$.

1. In the first phase (pre-processing phase), each process is assigned *L/P* contiguous blocks. Each process will then sort the blocks assigned to it using bmergesort_seq (sequential version of bitonic merge sort to sort any list) by using the technique described above to overlap the I/O communication time and the computation time. In this phase only three of the six buffers are used. Function bmergesort_ext_pre in Figure 10 gives the details for this step.
2. The parallel merge phase is more interesting. In each binary split iteration, we have *L/2* pairs of blocks that need to be compared and exchanged (if necessary). Each process needed to process $(L/2)/P$ pairs, which was done by applying the technique

```
Procedure bmergesort_ext_pre (B, groupstart, group, order)
(* Do an external, sorting of bitonic list starting at groupstart and of length group
in the specified order, using a limited internal buffer of size of (maxbuf * B)
  where maxbuf ≥ 3 . I/O server is expected to acknowledge when a buffer
  is sent which can be received by recv_ack function. Pre-fetch can be initiated
  by sending a recv request by calling send_rreq function *)
begin
  npart:=group div B; (* npart is the number of partitions for IO access *)
    for I:= -1 to npart do (* sort each partition *)
    begin
      if I > 0 then (* flush previous computation *)
        send_buf(buffer[(I-1) mod maxbuf], groupstart + (I-1) * B);
      if I < (npart-1) then (* prefetch next partition *)
        send_rreq(groupstart + (I+1) * B);
      if (I >=0) and (I < npart) then (* call sequential sort *)
        bmergesort_seq(buffer[I mod maxbuf], (order + I) mod 2);
      if I > 0 then (* receive ack for the previous flush *)
        recv_ack(groupstart + (I-1) * B);
      if I < npart-1 then (* receive response for data request*)
        recv_buf(buffer[(I+1) mod maxbuf], groupstart + (I+1) * B);
    end;(* for *)
end.(* procedure *)
```

FIGURE 10. BMERGESORT_EXT _PRE: Preprocessing for external sorting of any list

shown in Figure 9, which is efficient as $N >> B$. Since pair of blocks were prefetched or post-dumped, we needed six buffers instead of th three as shown in Figure 9.

3. The I/O communication was simulated in Multi by having an I/O process for each regular process. The send/receive requests for I/O from a regular process was sent to its I/O server. In PVM we did it slightly differently by having only one I/O node serving all compute nodes to study the impact of the I/O bottlenecks.

```
Function bufno (i) : integer ;
  (* Find the buffer number that should be assigned to the first block of the next
  block-pair to be processed in the i th iteration *)
begin
  bufno := (2 * I) mod NBUF ; (*NBUF is the total number of buffers used, 6 here
  *) (* buffers are used in round-robin fashion and each iteration requires 2 buffers
  *)
end.
Function bufposn (blockpair_no, pd) : integer ;
  (* Determine the position of the first element of the first block of the given block
  pair, if the block pairs are at a distance of pd *)
begin
  bufposn := (((blockpair_no div pd) * 2 * pd + (blockpair_no mod pd)) * BUF-
```

FIGURE 11. Functions that manage the buffers

The algorithms with the above modifications for the limited buffer are shown in Figure 11 Figure 12 and Figure 13.The *bufposn* function in Figure 11 assumes that the *L* block-pairs are numbered sequentially.

```
Procedure blsort_ext (B,my_id,ln,order)
(* Sort Bitonic lists of length N using only six buffers of size B *)
begin
 totblocks := N / B ; blockpairs_per_proc := (totblocks / P)/2 ; oln := ln ; ln = ln/2 ;
   while( ln  ≥ 1) (* Do binary split operation with peer distance = ln *) begin
      for i := -1 to blockpairs_per_proc do begin
          blockpair := my_id + i * P
          if i > 0 then begin (* dump the output of the previous iteration results *)
             send_buf(buffer[bufno(i-1)],B,bufposn(blockpair - P,ln)) ;
             send_buf(buffer[bufno(i-1)+1],B,bufposn(blockpair-P,ln)+ln*B)) ;
         end ;
         if i < blockpairs_per_proc then begin (* Prefetch the blockpair to be
             computed in next iteration *) send_rreq(bufposn(blockpair+P,ln),B) ;
             send_rreq(bufposn(blockpair+P)+ ln*B,ln) ;
         end ;
         if    i ≥ 0   and i < blockpairs_per_proc then begin (* Do binary splits *)
             if (odd (blockpair div oln) then curorder = (NOT order) ;
                         else curorder := order ; (* Determine the order for current pair *)
             bmerge(buffer[bufno(i)], buffer[bufno(i)+1],B,curorder) ;
             if ln = 1 then begin (* continue binary split inside the block *)
                 blsort_seq(buffer[bufno(i)],0,B,curorder) ;
                 blsort_seq(buffer[buffno(i)+1],0,B,curorder) ;
             end ;
         end ;
         if (i > 0) then begin (* receive acknowledgments for results sent at the
                 beginning of the loop *) recv_ack(bufposn(blockpair - P,ln)) ;
                 recv_ack(bufposn(blockpair - P,ln) + ln* B) ;
         end;
         if (i < blockpairs_per_proc) then begin(* receive data for next iteration *)
             recv_buf(buffer[bufno(i+1)],B) ; recv_buf(buffer[bufno(i+1)+1],B) ;
         end ;
     end ;
     barrier ; (* Wait until all processes complete current step *)
     ln := ln/2 ; (* One Binary split operation completed, so length of bitonic lists
             will be half its value earlier *)
end ;(* while loop *)
end.(* procedure *)
```

FIGURE 12. BLSORT_EXT - External sorting of bitonic lists with buffer of 6*B

```
Procedure bmergesort_ext (N,P,B,order)
(* Sort data[N-1:0] array in required order using P processes and
        a limited Buffer of size 6B *)
begin
    totblocks := N / B ; blocks_per_proc := totblocks / P ;
    forall p := 0 To P-1 do begin (* Create P processes to sort the data concur-
rently *)
        (* Preprocess the blocks : sort them in individually in alternating order to
create
            bitonic lists of length 2*B *)
        bmergesort_ext_pre(B,p*blocks_per_proc, blocks_per_proc,order) ;
        barrier ;
        (* Repeatedly merge bitonic lists by sorting then in appropriate order until
            the complete data list is sorted in required order *)
        for bln := 2*B To N step bln*2 do begin
            blsort_ext(B,p,bln,order) ;
            barrier ;
        end (* for bln loop *)
```

FIGURE 13. BMERGESORT_EXT - External sorting to sort any list using six buffers of limited size

CHAPTER 4

Predicting Speedup

In this chapter, we describe in more detail our methodology to accurately predict the speedup of an application by analyzing the algorithm.

We retain the model proposed by *Amdahl* [8], and later extended by *Driscoll and Daasch* [2] for the execution time of a parallel application. In their model, the execution time is separated into two components, sequential and parallel time. The sequential time is the portion of the execution time that is constant or increases with the number of processors and the parallel time is that portion of the execution time that decreases with the number of processors[2] for a given problem size. Both sequential and parallel times are assumed to be functions of the number of processors and problem size.

Thus the execution time for a parallel application in general can be given by the expression:

$$T(P,N) = Seq(P,N) + \frac{Par(P,N)}{P} \tag{1}$$

where P is the number of processors and N is the problem size. For a given problem size the expression for the execution time becomes:

$$T(P) = Seq(P) + \frac{Par(P)}{P} \tag{2}$$

The speedup in execution times can be defined as:

$$Speedup = \frac{T(1)}{T(P)} \tag{3}$$

where $T(1)$ is the execution time on a single processor, and is given by

$$T(1) = Seq(1) + Par(1) \tag{4}$$

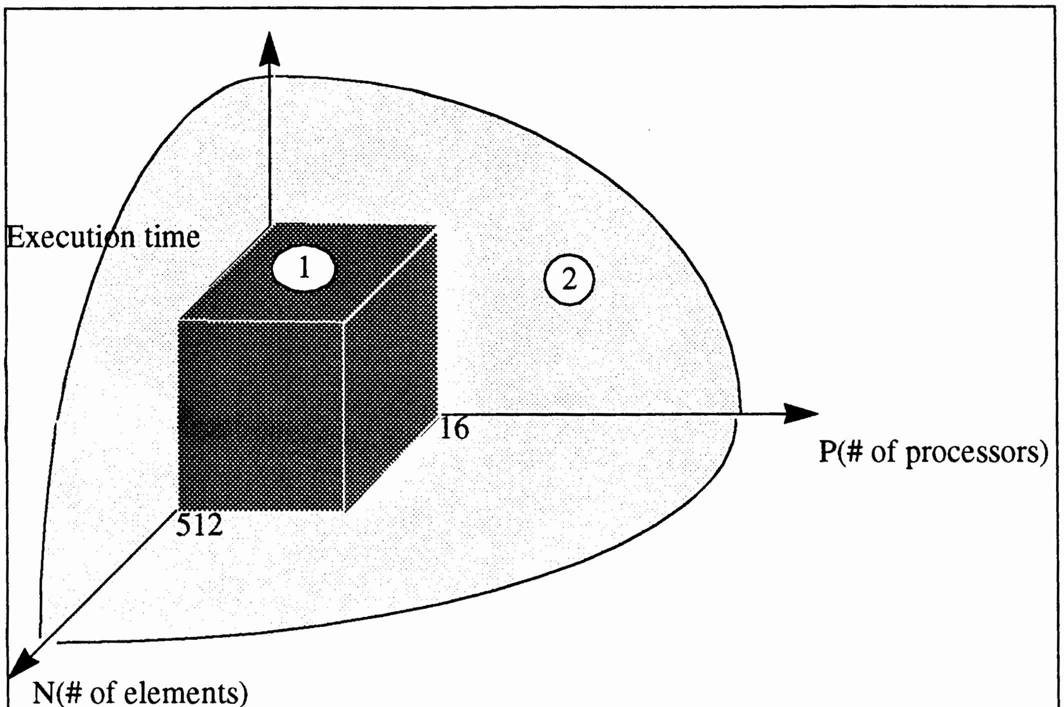
The sequential and parallel times for a parallel application are complex functions of P and N in general, and it is very difficult to find the exact functional forms by analyzing the application. It is possible to analyze sequential and parallel times to determine asymptotic bounds on runtimes. But such analysis cannot predict speedup with reasonable accuracy. On the other hand, one can do extensive experiments to find out runtimes for various problem sizes and system sizes to predict execution time. Here, we take a middle approach based on the following observations

- 90-10 rule : In typical parallel applications, 10% of the code contributes 90% of the runtime. So, runtimes for the 10% critical code can predict the overall runtime with good accuracy [13].
- Most critical code sections have algorithmic constructs that have well understood dependency on P and N for a specific parallel architecture.

We can thus quickly analyze the critical code segments to determine the terms that contribute to the sequential and parallel time. The overall sequential and parallel times are then expressed as a linear function of the various terms identified. A few experiments will be sufficient to give a good estimate of the coefficients for these terms. These ideas lead to our methodology for predicting speedup which is shown in Figure 14. Experiments are performed in the characterization region to find the coefficients in the expression for execution time. We then predict the execution times for the prediction region and validate the predicted values by a set of experiments in that region.

The steps involved in our methodology are:

1. Identify the critical code segments in a parallel application.



① Characterization Region : Range of experiments conducted to find the coefficients in the expression for execution time.

② Prediction Region : Range of experiments used to validate the execution times predicted by our methodology

FIGURE 14. Methodology for predicting speedup of a parallel application

The performance analysis tools such as prof, gprof available on Unix systems, profiler on DOS/Windows or similar tools on parallel computers can be used to analyze the time taken by various code segments of the parallel application and to determine those segments that contribute for most of the runtime. It may also be possible to easily identify those critical segments in case of applications that have well understood algorithms.

2. For each critical code segment, determine the sequential and parallel time contributors.

   As defined earlier parallel times are those components of the runtime that decrease as the number of processors is increased. Runtimes for code segments that are executed in parallel by different processors contribute to the parallel time. Examples are code segments that are executed by all the processors to operate on different data in data-parallel applications. Parallel *for* loops where different iterations of the *for* loop are independently executed by different processors are used to achieve parallel execution. The form factors for parallel runtimes can be determined by examining the total computational cost of the parallel code and dividing it by *P* - number of processors executing the parallel code.

Sequential times are contributed by those code segments that cannot be executed in parallel by all the processors. We refer to such code segments as sequential code segments. Examples are code segments that create new processes, code segments in critical sections, operations on lock or semaphore variables etc. Sequential runtimes can be determined by finding how may processors can execute the code in parallel. For example if only one processor can execute the code segment at any instant, then sequential time will be *P* times the time for one processor to execute the sequential code where *P* is the total number of processors executing the code segment. If the number of processors that can execute the sequential code segment follow the sequence - *P/2*, *P/4*, *P/8*, ... , *1* (or the reversed sequence), then the sequential time will be logP times the time taken by a single processor to execute the sequential code segment.

Analysis of bitonic merge sorting applications later in this section, will further illustrate the typical parallel and sequential code segments that can be found in an application.

3. For the parallel architecture under evaluation, find how the sequential and parallel parts vary with the problem size *N* and number of processors *P*.

   This step is essentially a refinement of the parallel and sequential terms determined in step 2 for the specific parallel system or architecture under consideration. The refinements can be done based on the data supplied or available for the specific parallel system. For example, the time for a barrier operation may have been already characterized by the parallel system vendor, or other users of the parallel system. Another option to apply such system-dependent refinements to the sequential and parallel terms is to conduct a limited set of experiments to study and characterize common parallel and sequential code segments. We actually used this option to characterize how barriers and process-creation times vary under the two parallel programming environments used in our studies - Multi and PVM.

4. Express the estimates for overall sequential time and parallel time as a linear function of the terms determined in step 3 above.

   Since the various sequential and parallel runtimes determined in steps 2 and 3, are independent of one another, we can express the overall runtime as a linear function of the different components. The resultant expression will express runtime as a function of the problem size *N* and the system size *P* with unknown constant coefficients for each term.

5. Conduct a limited set of experiments to get a good estimate of the coefficients for various terms that contribute to the overall runtime.

   We determine the coefficients by the numerical methods such as least square regression fit which will minimize the sum of the squares of error values, i.e., the difference between the value predicted by the runtime expression with the selected values for the coefficients and the actual values. We need to run at least as many experiments as the number of unknown coefficients using different values of *N* and *P*, and use the runtimes from those experiments to estimate the coefficients.

6. Use the expression for overall runtime to predict the speedup for problem sizes in prrediction region.

   Since, the expression obtained in step 5 models the overall runtime for any problem and system size, we can use it to find the ratio of appropriate runtimes to predict speedup expected for different system sizes.

The analysis of the bitonic merge sort implementations and experiments further illustrate how the above steps of our methodology can be applied to predict the speedup of parallel applications.

As mentioned earlier, the methodology is based on some assumptions about the characteristics of the parallel application and parallel architectures. In the following paragraphs we further elaborate on characteristics that are desirable and that will limit our methodology.

- data-parallel vs. task-parallel applications : Data parallel applications that use a predefined (defined at compile time) number of processes to execute the critical code segments are suited to our methodology as it is easier to analyze the sequential and parallel times for them. Task parallel applications where different tasks are loosely synchronized with each other and do different amount of computations are more difficult to analyze using our methodology.
- computation intensive vs. I/O intensive : Computationally intensive applications are better suited as the times for critical code segments can be analyzed and characterized. Overall runtime for I/O intensive applications may very much depend on the I/O device characteristics, network performance and other factors that make identification and characterization of critical segments difficult.
- shared vs. distributed memory : Architectures that support shared memory are desirable if contention for shared memory locations is negligible. In this case, we do not have to worry about the communication network characteristics or time to access non-local memory locations which may depend on many different features of the parallel architecture.
- overlapped vs. non-overlapped communication : Applications that are not sensitive to communication performance and that overlap all communication with computation are better suited to our methodology. The reason is that communication time will then be the overhead to initiate and complete communication which are typically fixed or may be proportional to message length for long messages. If communication time is not completely overlapped, than network characteristics and traffic patterns may determine the idle time spent by a processor waiting for communication to complete, and make analysis difficult. This observation is especially true for message-passing, distributed memory architectures.

- well defined critical code segments : It is also desirable that the number of critical code segments is small and fixed, and is not a function of the data input.

To summarize, data parallel applications that are computation intensive and have a few predefined critical code segments that contribute to most of the overall runtime are the best candidates for our methodology.

To evaluate our methodology for speedup prediction, we selected a data-parallel application, bitonic merge sort, and actually implemented several variations on two different parallel environments/systems. The results from the actual runs were used to compare predicted speedup with actual speedup. The details are given in later sections.

In the following sections, we describe in detail how our methodology was applied to the case of bitonic merge sorting. In section 4.1, we discuss several typical contributors to the sequential time which controls the speedup achievable for large number of processors. In Section 4.2, Section 4.3 and Section 4.4 we analyze the different versions of bitonic merge sorting to determine the critical code segments and the terms that contribute to the sequential and parallel times (steps 1-4 of our methodology).

4.1 Sequential time

A few of the complex tasks that contribute to the sequential time and hence limit the speedup achievable by a parallel application are the process creation overhead, process synchronizations and memory contention.

The most important building block of parallel applications is the process. Computational activity takes place when a process is created and is assigned to a processor in the underlying parallel computer to initiate parallel activity in the application. The creating process is called the parent process and the created processes are called child processes. There are at least two mechanisms that can be used for process creation. In linear process creation a single parent process has to create the child processes sequentially and the time for creation will be O(P). Tree process creation is used when the number of processes created is very large. In the case of tree process creation, there are parent processes at each level of the tree creating the child processes and the creation takes O(logP) time. We actually measured the time for process creation in both Multi and PVM environments. The measured times are given in Table 1. We observe that in both the environments process creation time is proportional to the number of processes created. We had some practical problems in creating more than

No of processes	Multi (in simulation time units)	PVM (in ms)
1	15	50
2	28	50
4	42	100
8	78	200
16	150	500
32	298	NA
64	582	NA
128	1158	NA
256	2310	NA
512	4618	NA

TABLE 1. Time for process creation in Multi and PVM

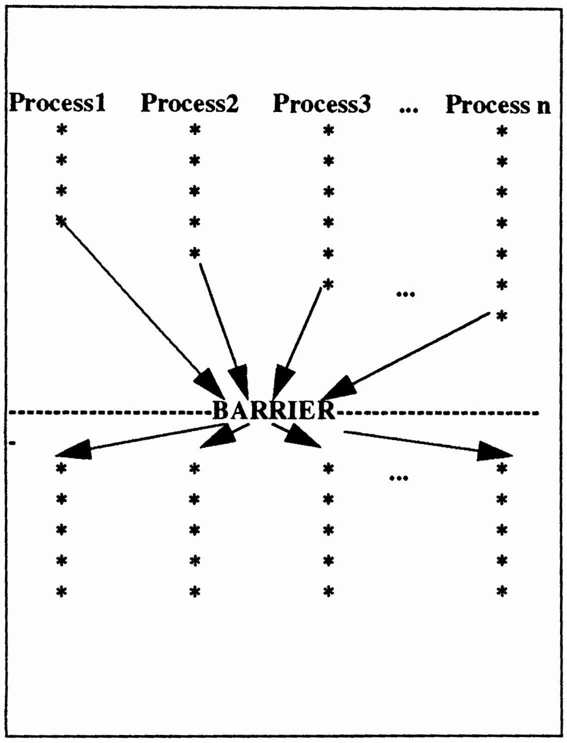
32 processes in PVM*. So the values for these cases are not available.

Since process creations are more expensive than process synchronizations, the usual tactic is to create the desired number of processes when the application begins execution and to synchronize them when necessary. Many processes can be created to work on different parts of the application called iterations. However, the processes must be synchronized in order for the results of an iteration to be used by the other processes for the next iterations.

The *Barrier* operation in the parallel program is one of the well known synchronization techniques. A *Barrier* is a point in the application where parallel processes wait

* Under PVM environment, we could not run 32 or more tasks due to the limits on number of files that can be opened by the PVM daemon. The limits could not be avoided even when we use multiple workstations to run all the tasks of parallel application.

for each other[1]. The first process to execute the *Barrier* statement will simply wait until all the other processes have arrived. After all the processes have reached the *Barrier* statement, they are all released to continue parallel execution. The barrier implementation can be linear, where a single entity is used to coordinate the synchronization. In tree synchronization, all the processes are brought to the root of the tree [1] and the process takes O(logP) time. Figure 15 shows the *Barrier* sychronization in detail.



```
Procedure INIT_BARRIER;
BEGIN
  bcount := 0 ; bn := proc
  unlock(barrival) ;
  lock(bdeparture) ;
END;

Procedure BARRIER ;
BEGIN
 lock(barrival) ;
 bcount := bcount+1 ;
 if(bcount < bn) then unlock(barrival)
 else unlock(bdeparture) ;
  lock(bdeparture) ;
  bcount := bcount-1 ;
   if bcount > 0 then
     unlock(bdeparture)
   else unlock(barrival) ;
 END;
```

FIGURE 15. Barrier synchronization of processes (* indicates process execution), and its implementation in Pascal

The procedure INIT_BARRIER is called only once before calling the barrier. Each time a barrier is required, each of the processes calls the procedure BARRIER. As shown in Figure 15, the body of the BARRIER has two critical regions that control access to the shared variable *bcount.* The lock variables, barrival and bdeparture, ensure that only one function can be in these critical regions accessing and modifying the variable *bcount.* This serialization effect implies O(P) time for a BARRIER operation to complete.

The time for barrier operation was measured both in Multi and in PVM on a Unix system, and is given in Table 2. In case of Multi the time was measured by setting a breakpoint before and after barrier calls that are executed by a unique processor. Measuring the time for barrier operation on PVM was not that easy. We measured the total time for all barriers by measuring the difference in runtimes with and without barriers. We then divided the total time for all barriers by the total number of barriers executed by the application. The results show that the time for barrier operation scale linearly with the number of processors in both Multi and PVM as expected. On Multi, the time

No of processors	Multi (in simulation time units)	PVM (in ms)
	Unix	Unix
1	30	50
2	40	100
4	70	250
8	130	500
16	300	1100
32	660	NA
64	1310	NA
128	2590	NA
256	5150	NA
512	10270	NA

TABLE 2. Time for barrier function for a single process in Multi and PVM

is much less because, in a shared memory model, the barrier will involve atomic accesses only to shared memory locations. The time is in milliseconds on PVM, as the barrier operation involves communication over TCP/IP sockets between the tasks of a parallel application.

The programming constructs explained so far that contribute to the sequential time are only a small part of the overall code of a typical parallel application and are referred to as the 'overhead'. The bulk of a parallel application code is usually executed by different processors concurrently, and therefore contributes to the parallel time. It is also considered as the useful work of a parallel application. In the data parallel programming model the code that contributes to the parallel time is executed by all the processors concurrently, but the processors will be executing the same code or operation on different portions of the data set.

We now consider the bitonic merge sort application in more detail to identify the sequential and parallel parts and how they scale with the problem size and the system size.

4.2 Analysis of Bitonic Merge Sort for $N = P$

We first analyze the simplest algorithms, bmergesort and blsort, given in Chapter 3 for the case $N = P$ (Figure 1 and Figure 4). The functional forms for the major blocks of code in that algorithm are:

- Initial process creation : There are P processes created in the beginning and assuming constant time to create a process, we find that this step will take O(P) time.
- Parallel merge core : The core is the innermost *for* loop in the blsort function which merges adjacent bitonic lists by sorting individual bitonic lists in appropriate order. Each loop iteration is a binary split which takes O(1) time as P processes execute them in parallel. Since there are log l binary splits to sort a bitonic list of length l, and the length of bitonic lists merged are *2,4,8...N*, the total time for this step is

$$\begin{aligned} \begin{bmatrix} \text{parallel time} \\ \text{for bmerge sort} \end{bmatrix} &= \sum_{(bl=2)}^{\log N} \sum_{(l=1)}^{(bl-1)} \begin{bmatrix} \text{time for binary split with} \\ \text{partners at distance } 2^{l} \end{bmatrix} \\ &= \sum_{(bl=2)}^{(\log N)} \sum_{(l=1)}^{(bl-1)} O(1) = O[\log^2 N] \end{aligned} \tag{5}$$

Here, we assumed that the time for one iteration of the parallel merge core takes a constant time. This is true as the parallel merge core requires a fixed amount of computation time and the communication time to exchange a single element is a constant with negligible contention.

- Barriers : Barriers are executed at the end of each merge step that increases the length of the sorted lists (and also the bitonic lists) by a factor 2. Since there are log P such merge steps, and each barrier takes a time proportional to the number of processors, we find that barriers contribute O(P log P) time. Since, $N = P$ for this algorithm, we can also say that there are logN barriers, and they contribute O(NlogN) time.

The total runtime can be expressed as a linear function of the three functional forms mentioned above, and hence can be modeled as:

$$T_{N=P}(N) = A \cdot \log^2 N + B \cdot N \cdot \log N + C \cdot N + D \tag{6}$$

The values of the three constants A, B and C can be calculated from a limited set of experiments and the curve fitting algorithms used to obtain the best values for the three constants. The runtime for sequential version will be

$$T_{P=1}(N) = E \cdot N \log^2 N + F \tag{7}$$

We get the speedup for $N = P$ case by the ratio *T(N,P=1)/T(N,P=N)*. From equations (6) and (7) we get

$$\text{speedup} = \frac{T_{N=P}(N)}{T_{P=1}(N)} = \frac{A \cdot \log^2 N + B \cdot N \cdot (\log N - 1) + (C \cdot N) + D}{E \cdot N \cdot \log^2 N + F} \tag{8}$$

In the next chapter we give the empirical values for constants, and show how this speedup model compared with the other models.

4.3 Analysis of Bitonic Merge Sort for $N > P$

In the case of $N > P$, a few more code constructs are added to the kernel loops as shown in Figure 5 and Figure 6. The effect is to add a few more functional forms to the expressions for runtime as described below.

- Preprocessing : Each processor initially sorts the group of elements assigned to it in the appropriate order by using the sequential version of bitonic merge sort. Since, each processor is assigned *N/P* elements, this step will take $O(N/P \log^2 N/P)$ time.
- Parallel merge core : This core which merges the initially sorted lists is executed $\log^2 P$ times as explained in the previous section, and in each step it has to compare and set *N/P* pairs of elements. So, this step will contribute $O(N/P \log^2 P)$ time.
- Sequential merge : At the end of each merge step in blsort_ngtp function, each processor does the final steps of a bitonic merge to sort the block of elements assigned to it (= *N/P*). This processing time is $O(N/P \log^2 N/P)$ and is executed logP times. So, total contribution of this step to runtime will be $O(\log P * N/P \log^2 N/P)$.

The above observations can be used to modify equation (6) to obtain the model for the parallel runtime when $N > P$ which is given below

$$\begin{aligned} T_{N>P}(N, P) &= A \cdot \frac{N}{P}\log^2 P + B \cdot P\log P + C \cdot P \\ &\quad + D \cdot \frac{N}{P}\log^2\frac{N}{P} + E \cdot \log P \cdot \frac{N}{P}\log^2\frac{N}{P} \end{aligned} \tag{9}$$

Speedup is given by the ratio: $(T_{N>P}(N, P)) / (T_{N>P}(N, 1))$

The equation below shows the expression for Speedup:

$$Speedup = \frac{T_{N>P}(N,P)}{T_{N>P}(N,1)}$$

$$= \frac{A \cdot \frac{N}{P}\log^2 P + B \cdot P\log P + C \cdot P + D \cdot \frac{N}{P}\log^2\frac{N}{P} + E \cdot \log P \cdot \frac{N}{P}\log^2\frac{N}{P}}{A \cdot N\log^2 N} \tag{10}$$

4.4 Analysis of Bitonic Merge Sort for $N \geq P$ and limited buffer size

The I/O communications are overlapped with computation, and we assume that I/O communication for two blocks can complete in less time than it takes to do computation in the innermost loop. So, the only effect of I/O communication is the overheads caused by the send and receive calls. The preprocessing and parallel merge core are different from those for the previous two cases, and hence their functional forms also differ as described below.

- Preprocessing : Total number of blocks sorted is *N/B* in this step, and each process handles *1/P(N/B)* blocks. Sorting each block takes time proportional to (B $\log^2$ B). So, overall parallel runtime for this step is of the form (N/(PB)) *(B$\log^2$ B + F) =(N/P($\log^2$B+F/B)) where F is a constant equal to the communication overheads per iteration in the function blsort_seq_ext. Hence, for a given *B*, the time for this step is proportional to N/P.
- Parallel merge core : The parallel merge core, which is the innermost *for loop* of the procedure blsort_ext, is executed about ($\log^2$ N/B) times as in the algorithms before, and in each for loop every processor processes N/2PB block pairs in pipeline. Processing of each block pair involves eight communication calls and compare and exchange operations on *B* pairs of elements (to do the binary split). Also in the last iteration of the core, *B* elements are sorted in 2 B $\log^2$ B time. So, overall runtime for this step is of the form ($\log^2$N/B) * N/2PB * B + log N/B * N/2PB

* 2 B log ^{2}B. Since, B is constant , the time for this step is of the form N/2P * log ^{2} N/B + N/P log N/B. We can ignore the log term compared to log^2 term*. So, we find that this step contributes a term N/P log^{2}N/B to the overall runtime.

- Barriers : Barriers are executed at the end of the parallel merge core, and are executed about (log ^{2} N/B) times instead of (log N/B) times as for the previous two cases. So barriers contribute a term Plog^{2}N/B.

The overall runtime for this case is due to the three factors described above and the creation of *P* processes, and therefore is of the form

$$T_{buf}(N,P) \;=\; A\cdot\frac{N}{P}+F\cdot\frac{N}{P}\cdot\log^2\frac{N}{B}+D\cdot P\cdot\log^2\frac{N}{B}+E \qquad (11)$$

Speedup is obtained as in the previous cases by the ratio T_{buf}(N,P)/T_{buf}(N,P=1) which is given by the equation below.

$$T_{buf}(N,P) \;=\; \frac{A\cdot\frac{N}{P}+F\cdot\frac{N}{P}\cdot\log^2 N+D\cdot P\cdot\log^2 N-D'\cdot P+E}{A\cdot N+F\cdot N\cdot\log^2 N+D\cdot\log N+E'} \qquad (12)$$

In the analysis for parallel merge core above, we assumed that computation time dominates communication time. If that is not true, then communication time dominates the computation time. The communication involves 8 messages and the communication time will be a linear function of the buffer size, and is of the form (a + bB). Since *B* (the buffer size) is a constant for an implementation, the communication time for the innermost loop of the parallel merge core is a constant. If the communication time dominates the overall loop time, the overall loop time will also be a constant. Hence, we find that the term contributed by the parallel merge core will have the same form irrespective of whether the communication time dominates the computation time or not.

* If N/B > 5 then log(N/B) will be less than 20% of log^2(N/B).

CHAPTER 5

Multi : Experiments and Results

We used the parallel simulation environment, Multi to implement the bitonic merge sort application and to verify that our methodology can predict speedup accurately. In this chapter, we give the implementation details and results. The speedup results obtained from our methodology are compared with the speedups estimated by other available techniques. The results show that our methodology results in better speedup prediction than simple theoretic models, and comparable or better than the prediction supported by the tool.

5.1 Environment

The Multi-Pascal interpreter[1] and debugging tool runs on a variety of small and large computers, including IBM PC-compatibles. The interpreter allows multi-pascal programs to be checked for syntax errors and executed to see if they produce the desired results. The debugger allows the programmer to work interactively, by allowing the programmer to set the break points and keep track of the status and location of parallel processes. It helps the programmer to monitor the performance of the program as a whole. The Multi-Pascal simulation system is designed for use with a maximum of 512 processors(256 in the MS-DOS version).

The Multi-Pascal system is able to simulate the performance of a parallel program on a real multiprocessor or multicomputers. At the end of program execution, the system will display the total program execution time called the "Parallel Execution Time", the estimated value of the actual running time of the program on the target multiprocessor, "Sequential Execution Time", the estimate of the execution time on a uniprocessor computer, the total number of processors used during program execution, and "Speedup", the sequential time divided by the parallel time.

5.2 Speedup prediction models

The following are the different speedup prediction mechanisms that were used in the experiments to predict speedup, and their accuracies are compared in the next section.

5.2.1 Speedup prediction by simple theoretic models

The simple theoretic model for speedup as given by *Amdahl's law* and extended later by others expresses speedup as given by the equations (1)-(4). We compute the sequential time by running the parallel version of the program for several values of *P*, with the problem size set to '0'. To compute the parallel time, we run the sequential version of the program for various problem sizes with *P* set to '1'.

The experiments to compute the sequential time were performed for *P* varying from 2^0 to 2^9. Table 3 gives the Seq(P) computed for various number of processors with the problem size set to 0.

Number of Procs	**Seq(P)**
1	247
2	307
4	437
8	727
16	1287
32	2387
64	4587
128	9007
256	17847
512	35507

TABLE 3. Seq(P) for various number of processors with problem size set to '0'

The experiments to compute the parallel time were performed for N varying from 2^3 up to 2^{13}. Table 4 gives the parallel time Par(N) computed for various problem sizes with number of processors equal to '1'.

Problem Size	Par(N)
8	4161
16	10865
32	27183
64	66807
128	163377
256	395097
512	943303
1024	2186846
2048	5110577
4096	11815946
8192	27061469

TABLE 4. Par(N) for various problem sizes with a single processor.

5.2.2 Speedup prediction by Multi tool

The Multi simulation tool has a built in utility that estimates the sequential time of an application on P processors from a parallel simulation. The sequential time is used to estimate the speedup achieved and is reported at the end of an experiment. We used this feature of Multi as one of the ways to find speedup. As we see later, estimates by Multi were better than simple theoretic models, but still not accurate for fine grain parallel applications.

The Pascal program for bitonic merge sort was executed for the problem sizes (N) from 2^3 to 2^{13} and for the number of processors (P) from 2^0 to 2^9. Each processor handled N/P elements of the list. The problem size was limited to 2^{13} as Multi failed to run for larger problem sizes. Numbers are also not shown for the cases when $P = 512$ and $N \geq 1024$, as Multi failed to run for these cases. Table 5 gives the speedup numbers as reported by Multi for various problem sizes and number of processors. The cases when $P > N$ are not valid as our algorithm was designed only for $N \geq P$, and are shown by the shaded cells in Table 5.

Prob-lem size = *N*	**Number of processors = *P***									
	1	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
8	1.0	1.69	2.69	3.88	▨	▨	▨	▨	▨	▨
16	1.0	1.77	2.92	4.49	5.74	▨	▨	▨	▨	▨
32	1.0	1.78	3.06	4.93	7.04	7.81	▨	▨	▨	▨
64	1.0	1.82	3.18	5.31	7.86	9.68	9.77	▨	▨	▨
128	1.0	1.87	3.32	5.61	8.78	11.75	12.33	11.47	▨	▨
256	1.0	1.90	3.46	5.94	9.49	13.50	15.76	14.64	12.94	▨
512	1.0	1.92	3.55	6.24	10.17	15.05	19.37	19.40	16.68	14.29
1024	1.0	1.97	3.82	7.22	13.09	22.01	32.51	37.22	29.41	NA
2048	1.0	1.97	3.85	7.34	13.51	23.33	36.38	47.80	44.78	NA
4096	1.0	1.98	3.87	7.44	13.83	24.36	39.38	55.41	63.73	NA
8192	1.0	1.98	3.89	7.51	14.09	25.18	41.62	61.43	77.83	NA

TABLE 5. Speedup for bitonic sort algorithm as estimated by Multi

5.2.3 Speedup estimated by simulated runtimes

Parallel applications can be run on Multi to simulate its run on a specific type of a parallel system. Multi uses a single processor to simulate all the processors of a specified parallel architecture, and reports the overall time. We can then divide the runtime obtained for a single processor ($P = 1$) by the runtime for P processors to compute the speedup.

Multi gives the parallel runtime in addition to speedup. The parallel runtimes obtained from our experimental runs for various N's and P's are shown in Table 6. We can compute the speedup using its definition by dividing the runtimes for P processors by the time for the same problem size when $P=1$.

Prob-lem size = *N*	**Number of processors = *P***									
	1	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
8	4161	2201	2141	2491	▨	▨	▨	▨	▨	▨
16	10865	5418	4038	3898	4998	▨	▨	▨	▨	▨
32	27183	12922	8672	6752	6942	10102	▨	▨	▨	▨
64	66807	30947	19257	13357	11387	13427	21237	▨	▨	▨
128	163377	74117	44147	28577	21097	19827	27097	46057	▨	▨
256	395097	177760	101900	63030	43180	34910	37430	57210	101100	▨
512	943303	423636	236566	140826	91936	67926	60486	75286	122886	222116
1024	2186846	962669	510649	279869	161779	102379	75689	75729	119119	NA
2048	5110577	2262849	1188519	641479	361779	219449	149529	124029	151289	NA
4096	11815946	5262359	2743669	1464269	811479	478239	310279	233409	220699	NA
8192	27061469	12119279	6288269	3324109	1817819	1049129	658229	466199	389829	NA

TABLE 6. Parallel runtimes for bitonic sort algorithm

5.2.4 Speedup prediction by our methodology

The speedup in our methodology is modeled by the equations in Chapter 4. We only run a small set of experiments with a small problem size N and number of processors P varying from 2^3 to 2^9. We then use a curve fitting algorithm, namely the least square, to determine the best values for the constants in those equations.

For $N = P$ case, we performed the characterization experiments with N and P varying from 2^0 to 2^5. The following equation gives the best estimates for the coefficients and the overall expression for the execution time for the simplest version ($N = P$) of bitonic merge sorting algorithm (bmerge_sort).

$$T_{N=P}(N) = 341 + 31.23 \cdot N \cdot \log N - 210.81 \cdot N + 403.2 \cdot \log^2 N \qquad (13)$$

The coefficients look reasonable except for N which has a higher negative value than expected. In fact, as we will shortly notice that the execution runtimes for other cases also turned out to have higher negative coefficients for the term N or P than expected.

For $N > P$ case, we performed the characterization experiments with N varying from 2^3 to 2^9 and P varying from 2^0 to 2^4. The following equation gives the best estimates for the coefficients and the overall expression for the execution time for the version ($N > P$) of bitonic merge sorting algorithm (bmergesort_ngtp).

$$\begin{aligned} T_{N>P} &= 14773 + 146 \cdot \frac{N}{P} \cdot \log^2 P + 899 \cdot P \log P - 4486 \cdot P \\ &+ 22.6 \cdot \frac{N}{P} \cdot \log^2 \frac{N}{P} + 0.811 \cdot \log P \cdot \frac{N}{P} \cdot \log^2 \frac{N}{P} \end{aligned} \tag{14}$$

For $N > P$ with limited buffer size, we performed the characterization experiments for a buffer size B of 4 and 16 elements, with N varying from 2^3 to 2^9 and P varying from 2^0 to 2^4. The following equations give the best estimates for the coefficients and the overall expression for the execution time for the version ($N \geq P$ and limited buffer) of bitonic merge sorting algorithm (bmergesort_ext). Equation (15) refers to the execution time for a limited buffer of size 4 elements and equation (16) refers to that for a buffer of size 16 elements.

$$T_{buf}(N, P) = 23041 - 2748 \cdot \frac{N}{P} - 2470 \cdot P + 285 \cdot \frac{N}{P} \cdot \log^2 N + 45 \cdot P \log^2 N \tag{15}$$

$$T_{buf}(N, P) = 13573 - 1952 \cdot \frac{N}{P} - 4126 \cdot P + 87.34 \cdot \frac{N}{P} \cdot \log^2 N + 80.26 \cdot P \log^2 N \tag{16}$$

The graphs in Figure 16 thru Figure 19 show the runtimes that were used in curve fitting to determine the coefficients for our runtime models. Figure 16 shows the curve fitting for $N = P$ case. The curve fitting for the $N > P$ case is shown in Figure 17. Figure 18 and Figure 19 shows the curve fitting for the limited buffer algorithm when buffer size is 4 and 16 elements respectively.

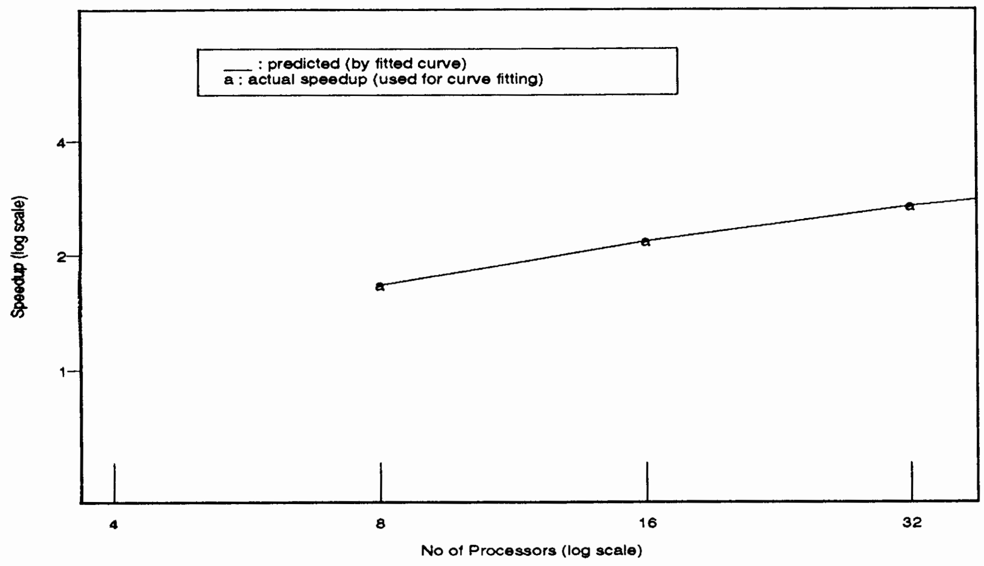


FIGURE 16. Curve fitting runtimes for bmergesort ($N = P$)

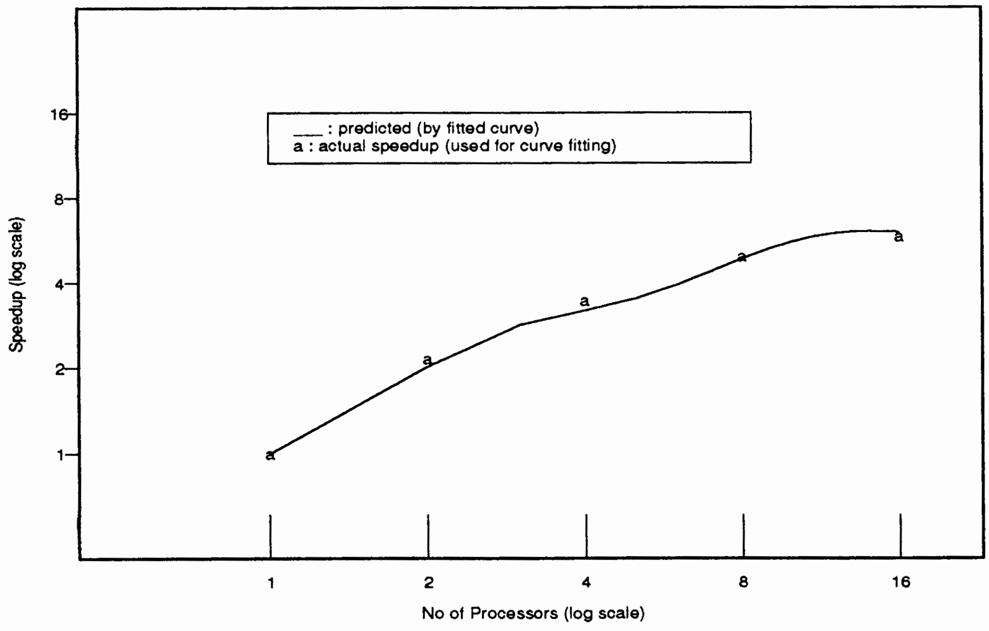


FIGURE 17. Curve fitting runtimes for bmergesort_ngtp ($N > P$) for $N = 2^6$

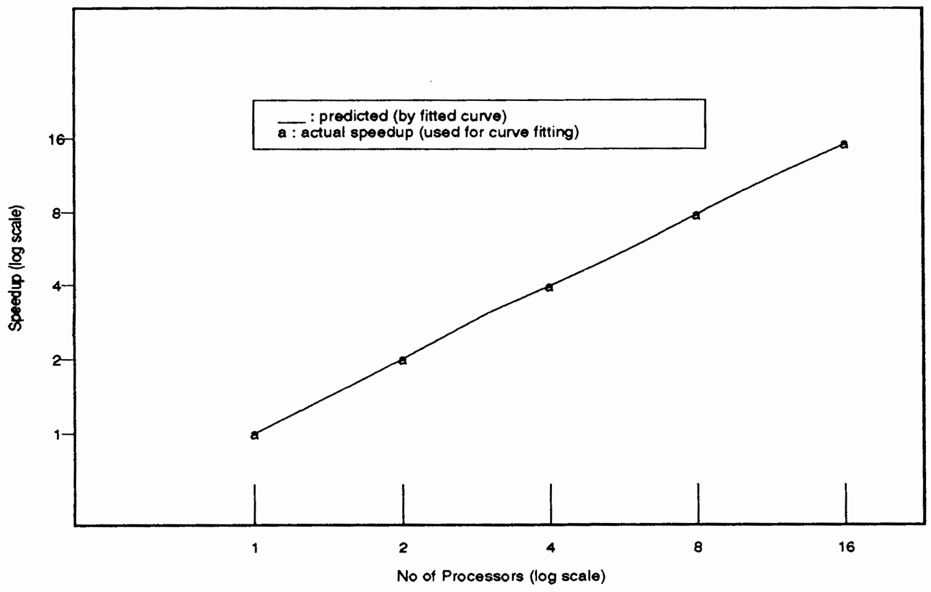


FIGURE 18. Curve fitting runtimes for bmergesort_ext for $B = 4$ and $N = 2^{10}$

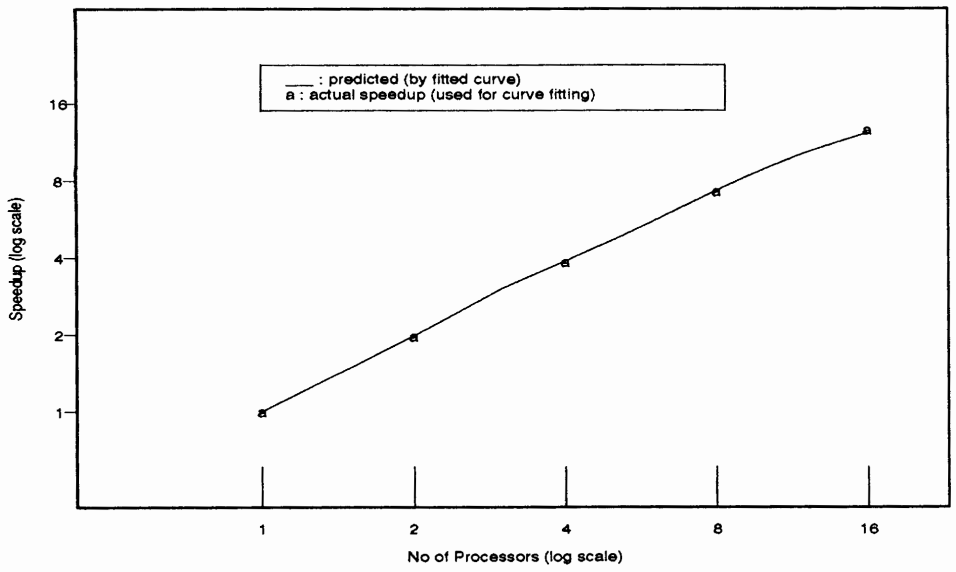
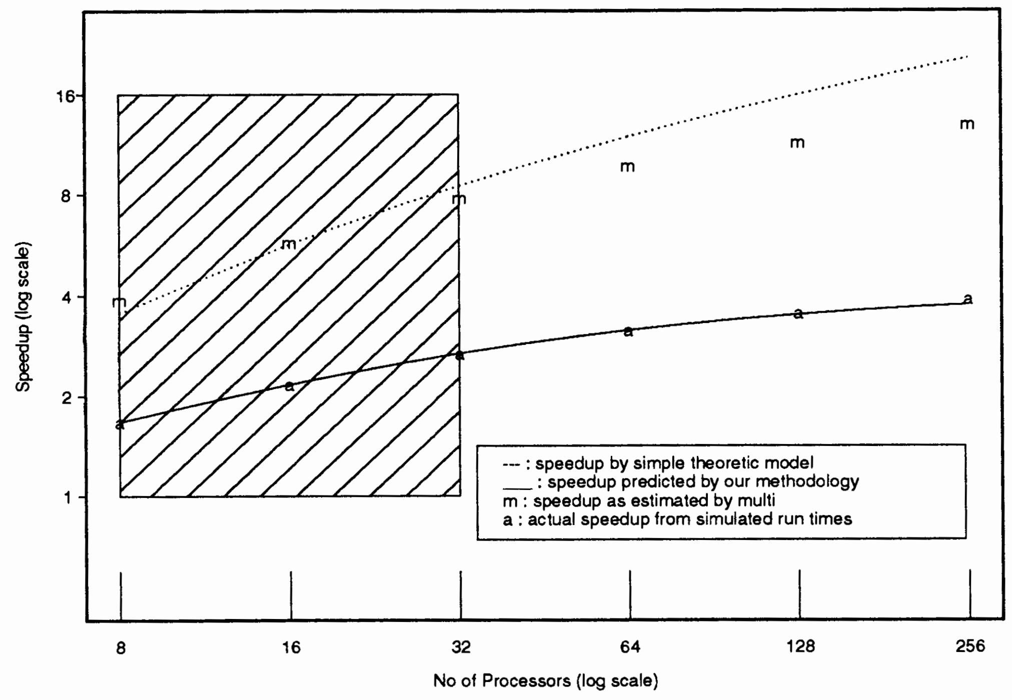


FIGURE 19. Curve fitting runtimes for bmergesort_ext for $B = 16$ and $N=2^{10}$

5.3 Results

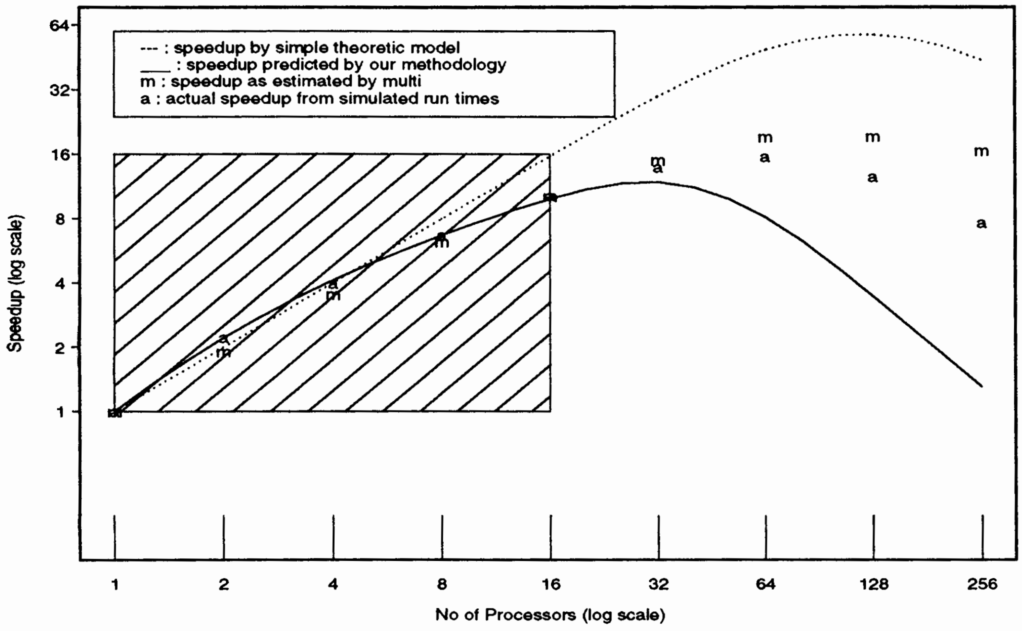
The speedup values computed by different methods are shown in Figure 20, Figure 21 and Figure 22. Figure 20 gives the speedup values for the simpler version of the algo-



▨ Characterization region

FIGURE 20. Speedup for linear problem size when $N = P$

rithm that worked only for $N = P$. Hence, the size of the problem is scaled with the number of processors in the system so as to keep the number of data elements handled by a processor same (1 in this case). The curves show that speedup from simple theoretic models are not accurate. The estimates by Multi are better than the theoretic models. However, our methodology that depends on the functional forms for the critical code segments, accurately predicts the speedup and the results match very well with the actual values.



▨ Characterization region

FIGURE 21. Speedup for a constant problem size for $N = 2^9$ elements

Figure 21 gives the speedup results for a constant problem size of 512 elements. Again, we see that the prediction accuracy was same as for the linear problem size case for different methods. Our model even predicted that speedup will improve only for P up to about 32 processors, and would drop if we use more number of processors, where as the simple theoretic models predicted speedup will improve until 128 processors. Actual speedup numbers show that our methodology predicted the region of linear performance very well.

Figure 22 gives the results for the best problem size for a given number of processors. All the results shown in the graph are in the prediction region. Our experiments showed that the best problem size happens to be the largest problem size that we could run under Multi which is 8192 elements. The results again show that the simple theoretic models give a crude approximation, where as our methodology can predict the speedup reasonably accurately, and gives more conservative estimates by effectively taking into account the sequential components of runtime.

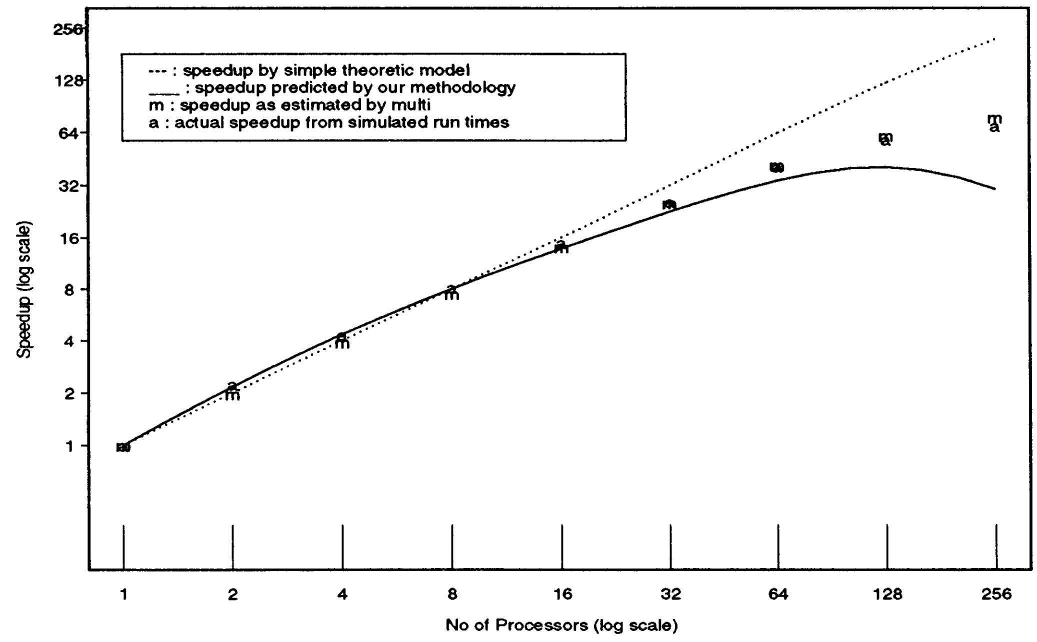


FIGURE 22. Speedup for the best (largest) problem size for $N = 2^{13}$

The next two graphs, Figure 23 and Figure 24. , show the results when memory is limited, and effects of I/O on runtimes are considered. The speedup results shown in these graphs are in the prediction region. In this case, we could get higher speedups as communication was overlapped with computation. We have not shown the curves for the simple theoretic model as its accuracy was poor for the previous cases.

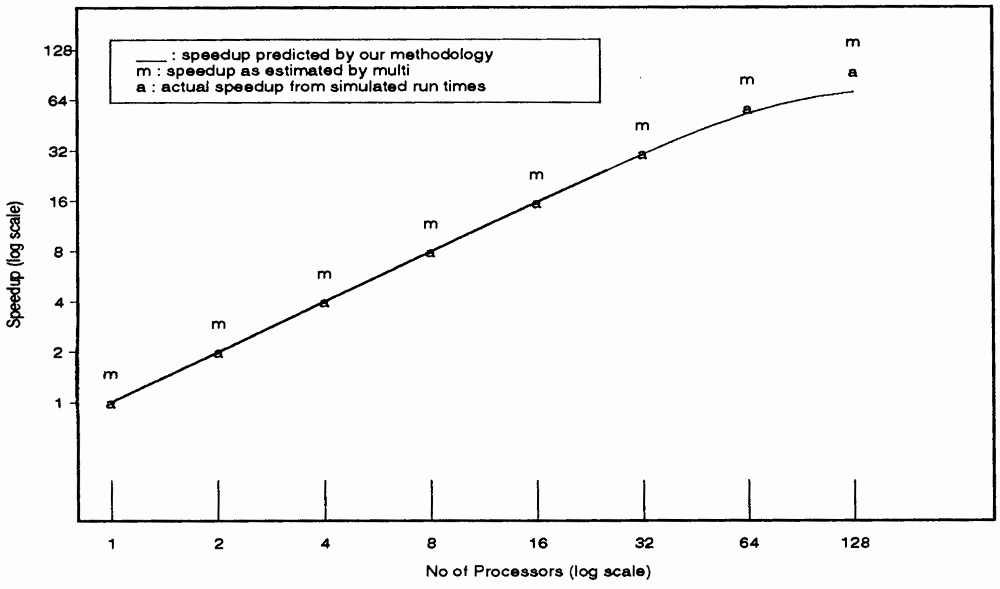


FIGURE 23. Speedup for a limited buffer of size 4 elements for $N = 2^{11}$

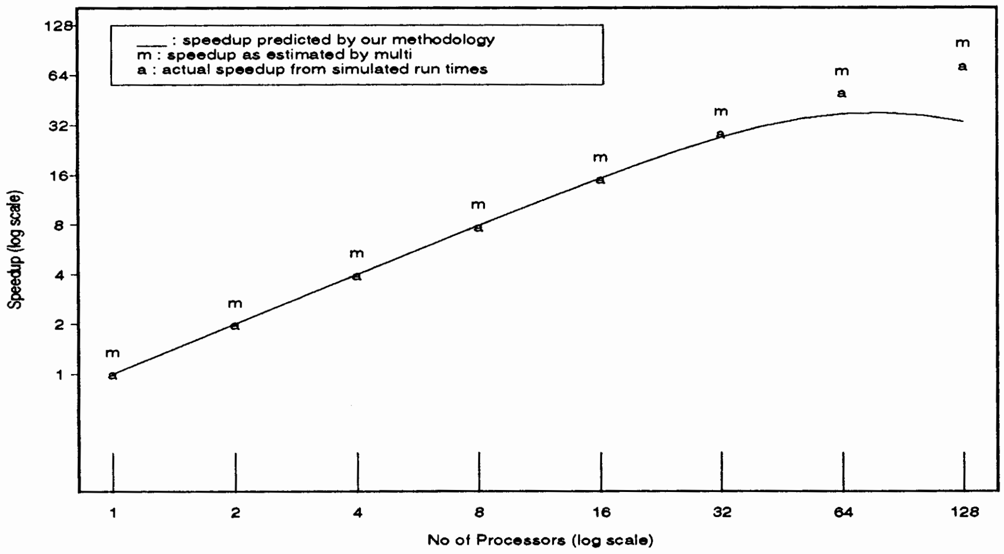


FIGURE 24. Speedup for a limited buffer of size 16 elements for $N = 2^{12}$

5.4 Summary

In this chapter we described the application of our methodology to different implementations of bitonic merge sorting application in Multi environment, and presented the results. The results showed that our methodology could predict speedup more accurately than simple theoretic models, and were better or comparable to the speedups estimated by the built-in feature of the Multi tool. We found that prediction will be more accurate if we extend the characterization region along the system size dimension i.e. increase the range of system size used for estimating the coefficients. The speedup was predicted well for all the three different implementations. Our methodology could predict the system sizes at which the speedup will peak, and beyond which we will get diminishing returns i.e. speedups decrease with increase in system size.

CHAPTER 6

PVM : Experiments and Results

We also evaluated our methodology by implementing the bitonic sort applications in C for the PVM environment. PVM is very different from Multi and is a very popular parallel programming environment. The results show that our methodology still gives good speedup estimates, thus demonstrating its applicability to different parallel architectures and programming environments.

6.1 Environment

PVM(Parallel Virtual Machine)[4] is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational resource. The individual computers may be shared or local memory multiprocessors that may be interconnected by a network. PVM presents a unified, general, and powerful computational environment for concurrent applications.User programs written in C or Fortran are provided access to PVM through the use of calls to PVM library routines for functions such as process initiation, message transmission and reception, and synchronization via barriers. Users may optionally control the execution location of specific application components.

6.2 Speedup Prediction by our Methodology

Unlike Multi, PVM does not support any built-in utility to estimate the speedup. Hence, for PVM we applied our methodology to predict the speedup and compared it with the actual speedup obtained from the runtimes. We also limited the experiments to two implementations of bitonic merge sort. One implementation was for the general case when $N \geq P$ assuming unlimited memory, and the other was the limited memory implementation for large problems. The runtimes were determined by dividing the total runtime for N iterations by N as the time for single iteration was too small to measure.

A small set of experiments were run in PVM, and our methodology was applied to determine the coefficients. The runtime results for N up to 2^{13} and P up to 2^2 were used to derive the coefficients. The actual experiments were run for N up to 2^{17} (128K) and P up to 2^4. The graphs in Figure 25 and Figure 26 show how the curves predicted by the model, fit the actual experimental values used to estimate the coefficients. The model for runtime with estimated coefficients are shown in equations (17) and (18) where all the coefficient values are expressed in milliseconds(ms).

$$\begin{aligned} T_{N>P} &= 161.9 + 0.23 \cdot \frac{N}{P} \cdot \log^2 P + 25.36 \cdot P \log P + 167.5 \cdot P \\ &\quad + 0.0025 \cdot \frac{N}{P} \cdot \log^2 \frac{N}{P} - 0.00047 \cdot \log P \cdot \frac{N}{P} \cdot \log^2 \frac{N}{P} \end{aligned} \tag{17}$$

$$T_{buf}(N, P) = 2921 - 41.72 \cdot \frac{N}{P} - 2424.6 \cdot P + 0.63 \cdot \frac{N}{P} \cdot \log^2 N + 46.03 P \log^2 N \tag{18}$$

We observe that the coefficients for the overhead terms (P and P log P) are much larger than the coefficients for the computation terms. As we see in the next section, the high overheads prevented speedups of more than 4 as predicted by the following equation. The graphs in the next section show how well these equations could predict the speedup.

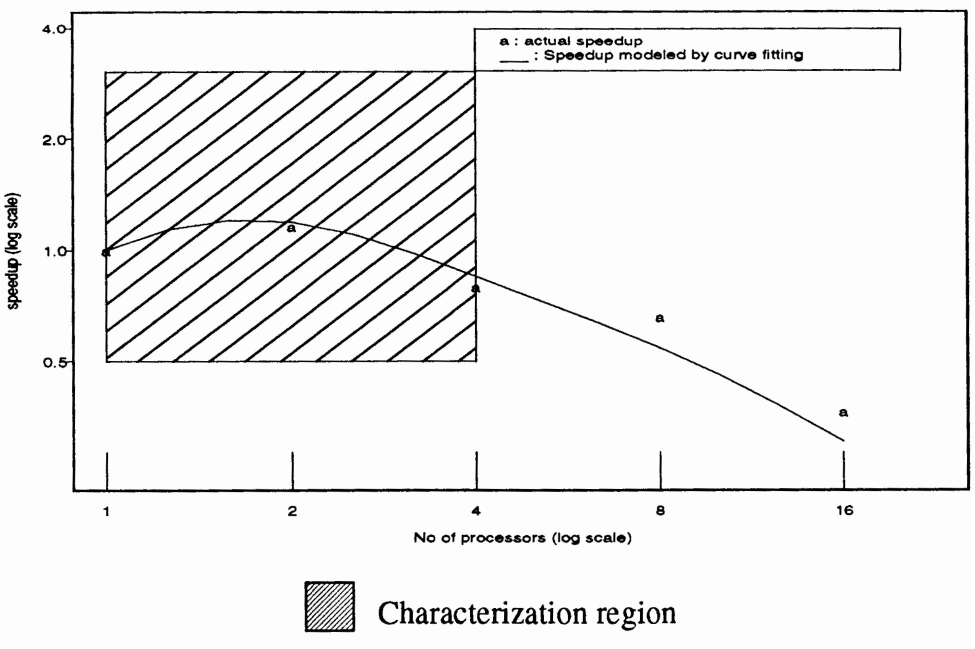


FIGURE 25. Curve fitting runtimes for unlimited memory implementation of bitonic merge sort for $N = 2^{12}$

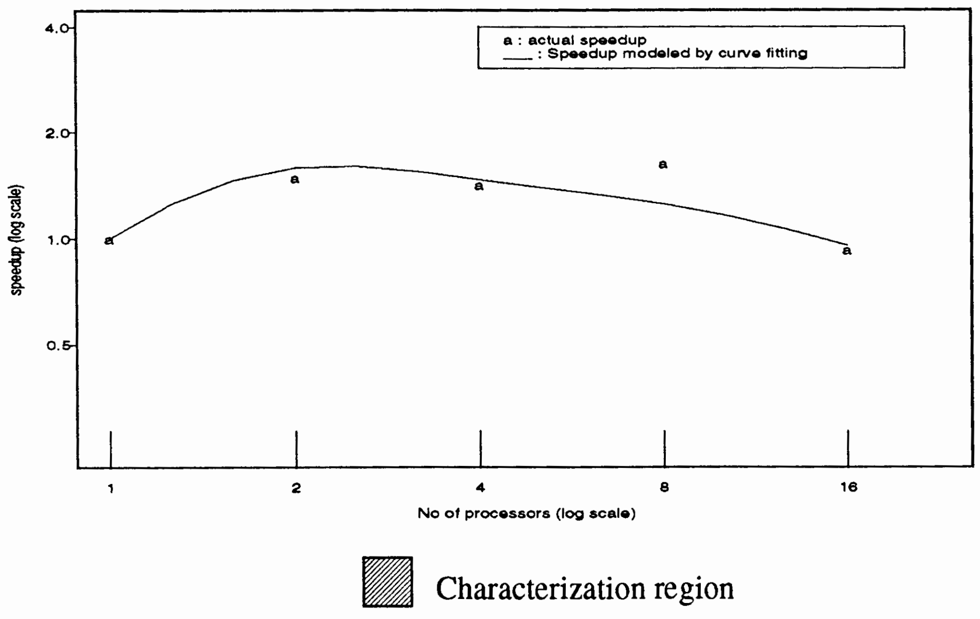


FIGURE 26. Curve fitting runtimes for limited-memory implementation of the bitonic merge sort algorithm for $N = 2^{10}$ and $B = 16$

6.3 Results

The speedup results are shown in Figure 27 and Figure 28. Figure 27 gives the results for the unlimited memory implementation, and Figure 28 gives the results for the fixed buffer size (the actual buffer size in the experiment was 16 elements) implementation. All the results given in Figure 27 and Figure 28 are in the prediction region. The speedup prediction by our methodology indicated that the PVM implementations will give very poor speedup numbers because of the high overheads for process creation, barriers and communication. The experimental results confirm that predictions were reasonably accurate. In fact until about $N = 8192$, the best speedup was achieved for $P = 1$ for the first implementation which means that we actually loose performance by parallelizing.

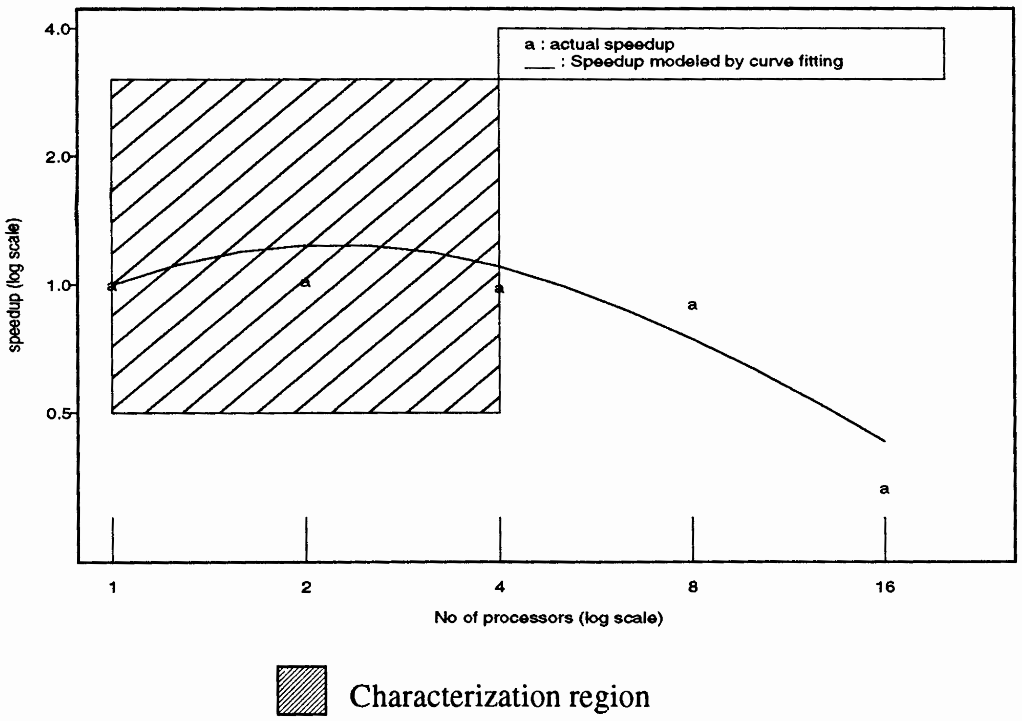


FIGURE 27. Speedup estimates for bitonic merge sort ($N \geq P$)for $N = 2^{14}$

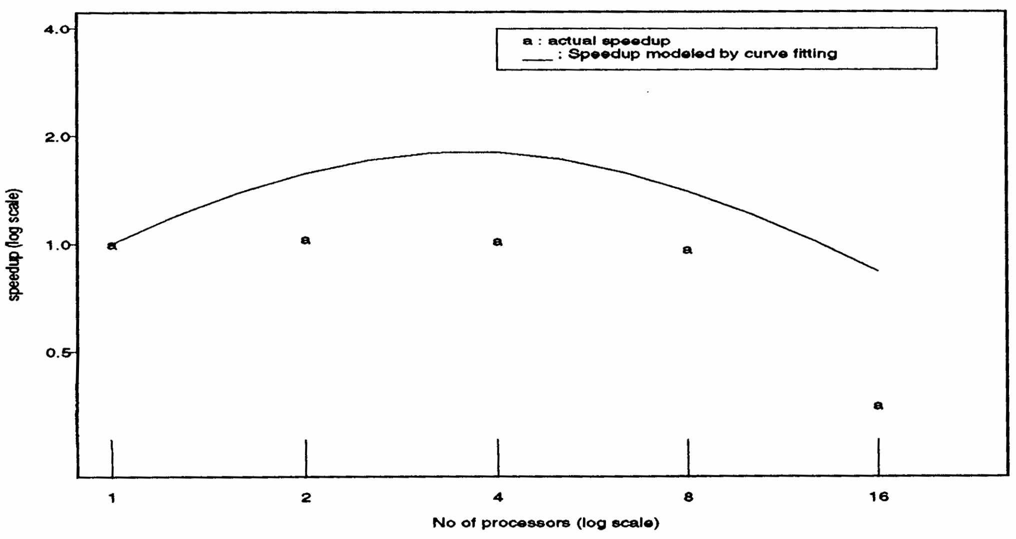


FIGURE 28. Speedup for $N \geq P$ and limited buffer size for $B = 16, N = 2^{11}$

6.4 Summary

In this chapter, we described the results of the application of our methodology to predict the speedup of bitonic merge sort application in PVM environment. Unlike Multi, PVM is a real parallel processing environment that is available on network of workstations and massively parallel processors. Our implementation used PVM on a network of workstations. PVM also differed from Multi in the underlying architecture of the parallel system. Multi implementations assumed shared memory systems, where as PVM used message passing (using TCP/IP sockets) on distributed memory systems. As the results showed, PVM has high overheads to access memory of remote nodes.

Our methodology predicted speedup well even for PVM implementation, which had different underlying parallel architecture and programming model. The terms in the model for total runtime remained the same as in multi, but the results from experiments in the characterization region resulted in significantly different values for the coefficients of the terms. The predicted speedup matched well with the actual values computed by taking the ratio of runtimes as determined from the experiments. The predicted speedup values clearly showed that speedup will actually start decreasing with increasing number of processors for even a small number.

CHAPTER 7

Conclusions and Future Research

In this thesis, we developed a methodology to show the effect of the number of processors on the performance of a parallel application, measured by the speedup achieved for a given number of processors. Our methodology is based on a good combination of analysis and empirical studies to accurately predict the runtime of a parallel application. The method involves the analysis of the critical code segments of an application to determine how the parallel and sequential parts of the code vary with the number of processors and the problem size. A limited set of experiments can then be used to predict the runtime of a parallel application for any number of processors and problem size.

We successfully applied the methodology to various bitonic-list based sorting programs by implementing them in Multi and PVM and showing that our methodology results in better speedup estimates than current theoretic models. The methodology was applied to get the results for smaller number of processors and problem size, and then to predict how the speedup will vary for larger numbers of processors and system size. The actual speedups as measured from the experiments were compared with the predicted values to show that the estimates were reasonably accurate.The methodology was able to predict the speedups well even in case of PVM which has high communication and other sequential overheads.

Our methodology was able to predict speedup better than simple theoretic models as we use a limited set of experiments to evaluate machine/architecture dependent parameters. The speedup curves are sensitive to how the parallel and sequential components of a parallel application vary with problem size and more importantly the system size. Actual experiments will help in accurate estimation of the magnitude of various sequential and parallel components of the overall runtime. In our studies, we found that the prediction accuracy was sensitive to the processor range of the characterization region. Prediction accuracy improved when we increased the range of processor sizes for the characterization region. More experiments with larger number of processors help in estimating the coefficients of various terms better, and thus predict the speedup more accurately. However, additional experiments will take more time. In fact this illustrates an important feature of our methodology - it is possible to get more accurate speedup prediction by expanding the characterization region and doing more experiments.

Our methodology is very attractive for those applications where the runtimes increase quickly with problems size, for example as square or cube of the problem size. In such a case, by restricting the characterization region to small problem sizes, time taken for the experiments in the characterization region will be negligible compared to the time taken to run experiments for the entire problem sizes of interest. For example, consider an application whose runtime is proportional to the square of the problem size. To determine the speedup over the range *N* to *10N*, one can do ten experiments at problem sizes *N, 2N, ... , 10N* at a total runtime cost of $\sum_{i=1}^{10} i^2$. If we use our methodology and restrict the characterization region to only 3N and can do only three experiments at *N, 2N* and *3N*, then the cost of the experiments in our methodology will be only $\sum_{i=1}^{3} i^2$ which is only about 14/385 or 4% of the total cost of the experiments.

The methodology also helped in understanding the shape of the speedup curves, as they show how the magnitude of various terms (that correspond to various segments of the code) change when the problem size or number of processor is increased. The coefficient values depend on the parallel architecture whereas the actual terms depend on the algorithm. The terms can only be altered by changing the algorithms used by the application, whereas the coefficients of the various terms can be altered by using an appropriate parallel machine and/or compiler/runtime environment. On PVM the coefficients for various terms clearly show that sequential components and overheads dominate the overall runtime, and hence will result in poor speedups that were observed.

7.1 Limitations

As explained earlier in the thesis, our methodology is not applicable for every case and is suitable only if they have the characteristics mentioned in Chapter 4. However, many data parallel applications in scientific applications do satisfy the requirements and hence our methodology is applicable to a wide number of actual applications. The following are some of the limitations of our methodology which may prevent its usefulness and applicability for some applications.

- runtimes that are non-linear functions of the problem and system size, and may have different functional forms in different ranges of problem or system sizes. One example is when there is overlap of communication and computation, and computation time dominates communication in one range, and vice-versa in the other. If the form factors for the computation time and communication time are different, then the overall runtime will have different form factors in different system and problem size ranges.
- Parallel machine or architecture under consideration have characteristics that result in non-linear runtimes for sequential components such as barriers. For example, barrier operation may take O(P) time for small number of processors, but may take O(log P) time for large number of processors.

- Applications are communication intensive and will depend on the performance of the communication network, and are sensitive to the delays due to the contention in the network or to the access of shared variables. Such delays due to resource conflicts are difficult to characterize without understanding its details and considerable analysis effort.

We feel that the limitations sometimes can be overcome by handling the analysis at the right granularity. The limitations may also restrict the applicability of our methodology for a certain applications over a limited range of system and problem sizes.

7.2 Future Research.

The methodology we developed was successfully applied to variations of bitonic list based on sorting programs for real machines - a cluster of workstations. For lack of time, we could not verify our methodology for MPP systems such as Paragon, Cray T3D etc. It would be interesting to apply our methodology for real applications running on these machines which have a different architecture and a high performance communication network.

It would be interesting to apply our methodology to existing real applications for current supercomputers that have some or all the desirable characteristics we mentioned in Chapter 3, and to evaluate the accuracy of speedup predictions for those applications.

The next step would be to automate some of the steps in our methodology. Profilers and performance analysis tools can be used to identify and select critical segments of a parallel application. Various functions provided by the message passing library for distributed memory machines, or the synchronizations primitives for shared memory machines can be characterized to show how they scale with the problem size and the number of processors. Such information can be used to assist or automate the effort to analyze the critical code segments.

References

[1] B. P. Lester, "The Art of Parallel Programming", Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] M. A. Driscoll and W. R. Daasch, "Accurate Predictions of Parallel Program Execution Time", to appear in *Journal of Parallel and Distributed Computing*, Feb 1995.

[3] X. Sun and L. M. Ni, "Scalable Problems and Memory Bounded Speedup", *Journal of Parallel and Distributed Computing*, 1993.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, "PVM 3 User's Guide and Reference Manual", 1994.

[5] A. Osterhaug, "A Guide to Parallel Programming on Sequent Computer Systems", Prentice Hall, Englewood Cliffs, NJ, 1989.

[6] M. J. Quinn, "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill Books, 1988.

[7] E. Horowitz and S. Sahni, "Fundamentals of Data Structures", Galgotia Booksource, 1983.

[8] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS Conference Proceedings*, pp 483-485, 1967.

[9] J. L. Gustafson, "Reevaluating Amdahl's law", *Communications of ACM*, pp 532-533, 1988.

[10] J. L. Gustafson, "Response to once again Amdahl's law", *Communications of ACM*, pp 263-264, 1989.

[11] I. D. Scherson and P. F. Corbett, "Communications Overhead and the Expected Parallel Speedup of Multidimensional Mesh-connected Parallel Processors", *Journal of Parallel and Distributed Computing*, pp 86-96, 1991.

[12] A. K. Nanda, H. Shing, T. H. Tzen, and L. M. Ni, "A Replicated Workload Framework to study Performance Degradation in Shared Memory Multiprocessors", in *Proc. of the 1990 International Conference on Parallel Processing*, pp I-161 - I-168, 1990.

[13] J. Hennessy and D. A. Patterson, "Computer Architecture : A Quantitative Approach", Morgan Kaufmann Publishers, 1990.

[14] D. Lenoski et. al., "The Stanford Dash Multiprocessor", *Computer*, March 1992, pp 63-79.

[15] A. Agarwal et. al., "Limitless Directories : A Scalable Cache Coherence Scheme.", *Proc. Fourth Int'l Conf. Architecture Support for Programming Languages and Operating Systems, ACM*, pp 224-234, 1991.