Performance Prediction for MPI Parallel Jobs

Weizhe Zhang, Tianyu Han, Yuanjing Zhang  
School of Computer Science and Technology  
Harbin Institute of Technology  
Harbin, HL, China  
wzzhang@hit.edu.cn

Albert M.K. Cheng  
Department of Computer Science  
University of Houston  
Houston, TX, USA  
cheng@cs.uh.edu

Abstract—Performance prediction of run time in the cluster is the foundation of efficient resource management and task scheduling. Considering the defects and limitations of traditional methods based on the history and modeling analysis, this paper proposes a new approach based on the Performance Skeleton. Through the use of the MPI library’s PMPI interface, we can insert wrapper-functions to the source code, which can access all communication traces without changing the original program or affecting the operation of the original program. To merge these trace logs, we designed the trace log regularization and merging algorithm. For compressing circulatory traces, the most central and difficult problem, this paper converts it into a circular sub-string compression problem, and proposes an algorithm based on the suffix array. Its performance is better than the existing algorithms. To automatically reconstruct the Performance Skeleton, it solves the scalable problems of calculation and communication time. Experimental results show that these methods can accurately estimate the run time of computing jobs. The error is less than 3% for a homogeneous cluster.

Keywords—parallel job; performance prediction; performance skeleton; circular sub-string compressing

I. INTRODUCTION

The performance prediction of parallel job run times is the foundation of efficient resource management and task scheduling in the various high performance computing environments [1, 2]. Usually the performance prediction methods can be divided into two categories: predictions based on history and theory, and predictions based on modeling and analysis.

Predictions based on history are from the precedent running logs of the same application under a given number of resources, and the relatively fixed settings and allocated resources according to certain principles. It leads hard to meet the requirements of a complicated and dynamic network computing environment. Predictions based on modeling and analyses are on the basis of the existing performance model, application programming model, and computing environment performance model. Forecasting results rely on the understanding of the application and the accuracy of the performance model; predictions based on modeling and analysis are widely applied but the precision needs further improvement.

By considering the restriction of historical prediction and modeling analysis in the cluster, the idea of utilizing the Performance Skeleton [3, 4] has been raised in recent years, the core idea of which demonstrates as follows: the most approaching predicting method lies in the usage of a short program, which could characterize the computing features of the original program, followed by an actual run under the designated large-scale parallel computing environment. Then we scale the time of the measured Performance Skeleton by the same proportions to acquire the estimated run time of the original program for accuracy of prediction and low complexity. In view of the practical measured performance of an application limited in scale, this method embodies the merits of flexibility by contrast of historical prediction and the accuracy of theoretical prediction.

This paper is organized as follows. Firstly, we discuss the progresses of parallel job predictions based on Skeleton reconstruction and its algorithm limitations in Section II, and then come up with the technology of MPI parallel job traces collection and merging in Section III. In Section IV, we put forward the issues of compressing the circulatory traces, which is the core problem, and transform it into a circular sub-string compression problem solved by a new kind of algorithm based on compressing suffix arrays. Section V and VI present the method of automatically reconstructing the Performance Skeleton and the experiment results and performance analysis in a cluster. Finally, we conclude in Section VII.

II. RELATED WORKS

In 2004, Sodhi and Subhlok proposed a method of application performance prediction based on the case program [3, 4] named Performance Skeleton, which is perhaps derived from Algorithmic Skeletons. The word Performance shows that this “skeleton code” has no primitive program functions, and that it just has similar performance with the original program. The method for generating this Performance Skeleton is as follows: firstly run the original program with the specified resources and environment, and collect and merge the running trace logs of different processes. Then a Performance Skeleton reflecting the original running features is reconstructed according to the traces. Finally, we can predict the run time of the original program under different computing resources according to the run time of the Performance Skeleton with reduced data and proportionate loop numbers throughout. As Xu and others’ work mention this method [5-7], it has been identified that recognizing and compressing the circulatory traces is the core programming technique of automatic construction of the case program. Moreover, there is
something regarding the similarities between the study of parallel programs and automatic construction case programs, such as Lu and others have proposed a method of using curve fitting to compress parallel programs for significantly reduced program run time, while maintaining the original program features to a certain extent [8]. Sherwood and others studied a method of automatic analysis periodicity of parallel programs [9]. But the purpose of these studies is generally irrelevant with constructing a Performance Skeleton to predict the running time of the original program.

So far, researches on the performance of case prediction programs are scant in quantity, and the aforementioned methods contain numerous fallacies. The crux lies in the fact that previous research realized on the homogeneous cluster excludes the adaptation of heterogeneity. In addition, when combining one-to-one communication, the same record that could be merged generally has different source/dest parameters. Xu and others have put forward solutions to this problem by using communication mode database matches with the communication mode of application programs. However, the drawback of this method is costly and can only deal with the condition that one single computing program has only one single communication mode. And as the other two algorithms mentioned in document [5, 7] solve the problems of recognizing and compressing the circulation of the traces, one has poor run time performance that could not be put into use on the condition that the number of traces exceeds 10^5; the other is unsatisfied in its performance referring to the effect of length recognition and compression algorithm O(n^2). This paper will develop the case prediction technology and come up with a solution method to the extant problems.

III. TRACE ACQUISITION AND COMBINATION OF MPI PARALLEL JOBS

In the procedure of generating the Performance Skeleton, we first need to run the original program to record run time traces. Due to acquiring SPMD programs from multiple processes, many trace logs need to be merged into one. Recording the original program's computing features and regularization is the basis of Performance Skeleton generation.

A. Trace Acquisition

MPI tasks adopt standard MPI functions to complete IPC (Inter process Communication). After recording traces of MPI calls, we could construct a Performance Skeleton that fully reflects the application features. On the other hand, limited but standardized MPI libraries provide conditions for acquiring the traces of MPI function calls.

After analysis of implemented MPI libraries, all the functions preset standard analysis interface PMPI [10]. For all the MPI functions with a prefix of "MPI ", there exists an identical function in the parameter as well as the function, with a prefix of "PMPI ". For that matter, we insert wrapper functions into our own realized class library, and intercept the parameter and time when called, then invoke the function prefixed "PMPI ". in the original way to realize the original function of the user application.

The wrapper function of MPI_Init and MPI_Finalize record information of the beginning and the end of programs, and maintain a trace log for each process, and for each of the remaining functions which are called by applications; the wrapper functions keep records of the meaningful parameter values. (1) Functions with no need to record parameters: MPI_Init, MPI_Finalize, MPI_Wait, MPI_Barrier, etc. This type of function has no parameters, or the parameter value of the function is irrelevant with the trace log to be recorded when the Performance Skeleton is created. (2) One to one blocking send and receive functions: MPI_Send, MPI_Recv; One to one non-blocking send and receive functions: MPI_Isend, MPI_Irecv. The parameters to be recorded: count, datatype, dest/source. (3) Non-blocking communication functions for checking completion: MPI_Waitall, etc. The parameters to be recorded: count, datatype, op, root.

NPB parallel compiler options are realized by modifying the configuration file make.config in the directory config. Take an IS application as example, set Class as A, the number of process as 2, and compile the MPI dynamic library with MPI automatically generated wrapper functions, the run time trace log is shown as Fig 1.

![Figure 1. Fragment of running traces (IS, Class=A).](image)

B. Trace-merge Algorithm

For the purpose of generating the SPMD Performance Skeleton, traces from each process need to be unified for constructing the Performance Skeleton. And it must be noted that even though application programs based on MPI are the parallel application programs of SPMD, the behavior of each process is not exactly the same, and neither are the disparity of traces from different processes. To maximize the possibility that the Performance Skeleton can simulate the
computing characteristics of the original application, it is
necessary to recognize the running content of each running
process of the SPMD program.

The inconsistencies of the SPMD programs running
during the execution process include two situations: (1) The
difference of sequence: each process receives a message
from the previous one, followed by calculations, then sends a
message to the following process, and finally, the number n-
1 process sends this message to the original process, which is
the process numbered 0. The 0 process first calls the function
MPI_Send, and then executes the function MPI_Recv while
the other processes have been asked to perform MPI_Recv,
and then executes MPI_Send. More common types of
sequence differences are caused by non-blocking
communications. For instance, each process using the
blocking MPI_Send to send messages to communicate, use
non-blocking communications MPI_Irecv to receive
messages, and each non-blocking receive function has a
response corresponding to MPI_Wait. (2) Function calls are different:
one or some of the processes call a certain MPI function, but
other processes do not. Some of the processes only send the
action followed by no receive action, while the other
processes only have the receive action and no send action.
Along with increasingly complex communication patterns,
results in different situations of function calls become more
complex.

To avoid sequence differences in the pre-processing
stage, the order of the communication traces from all
processes, which perhaps appear to be sequence differences,
will be adjusted. The principle is that receive statements
should be placed before send statements. In other words, the
placement and usage of MPI_Send and MPI_Recv, which
exist in relationship of sending and receiving, MPI_Send
should be placed in front of MPI_Recv; for the usage of
MPI_Irecv, MPI_Send and the accompanying MPI_Wait
statement, which exist in relationship between sending and
receiving, MPI_Irecv statement should be in front, followed
with the MPI_Wait statement, and finally the MPI_Send
statement. As shown in Algorithm 1.

Algorithm 1. Sequence-adjusting Algorithm.
Input: trace sequence set T.
Output: trace sequence set T after adjusting sequence.
1. Initialize the trace sequence set T.
2. \textbf{for} (enumerate each process number i)
3. \{s=0; tmpT.clear();
4. \textbf{for} (enumerate each trace number j of T[i])
5. \{ if \ (T[i][j].func==MPI_Send or
T[i][j].func==MPI_Recv)
6. \{ k=j; recvList.clear(); sendList.clear();
7. \textbf{while} \ ((T[i][k].func==MPI_Send or
T[i][k].func==MPI_Recv) and Match(T[i][k], T[i][j]))
8. \{sendList.push_back(T[i][k]);
9. \} else \ {recvList.push_back(T[i][k]);
10. k++;
11. j=k-1; tmpT.push_back(sendList + recvList);\}
12. \}
13. \} else
14. \{tmpT.push_back(T[i][j]);

15. \}
16. T[i]=tmpT;
17. \}

The solution to function call inconsistencies needs to be
after the above-mentioned algorithm 1. All inconsistencies of
sequence have been adjusted. At this point, the traces in
chronological order, if they are in a position different from
the process cannot be merged, indicating that the calling
function must be different. The processing thought at this
point is to correctly identify the trace from which the current
process should remain in the relative front position of the
Performance Skeleton, so that it will be merged into the trace
log. Therefore, Algorithm 2 is designed with the maximal
tracing downward position parameter m, if a certain trace and
its current position in the other processes downwards next m
entries cannot be combined then that indicates it should be in
the relative front position in the Performance Skeleton.

Algorithm 2. Trace-Merge Algorithm.
Input: Trace sequence set T after adjusting sequence.
Output: Trace sequence C after merging.
1. Initialize trace sequence set T; Initialize pointer
arrays p=0;
2. \textbf{while} (p exists a pointer which is not the end of trace
sequence) do
3. \{ if (current pointer corresponding all entries can be
combined) \{C.push_back(combine(T[p]));Add(p);
4. \} else \{Max=0;
5. \textbf{for} (enumerate each process number i)
6. \{ calculating the distance d[i] of number i
process matching downward;
7. \} if (d[i]>Max) \{Max=d[i];j=i;\}
8. \}
9. \textbf{for} (enumerate each process number i)
10. \{q.clear();
11. \textbf{if} (Match(T[i][p[i]]), T[i][p[i]])
\{q.push_back(p[i]);\}
12. \}
13. \}
14. \textbf{for} (enumerate each process number i)
15. \{ if (process i is merged in this loop) p[i]++;
16. C.push_back(combine(T,q)); Add(q);
17. \}
18. \}

Considering the number of traces can be large (for the
NPB 3.3, the amount of maximum trace entries in the
application of LU can be up to 10^6, and the practical
application may be greater), the efficiency of the trace-merge
algorithm is worthy of concern. Set the number of processes
as n, and the maximum of each process traces number as L.
In the pre-processing, each trace need to traverse, and the
time complexity is O(\pi nL); During the worst situations, every
trace needs to trace downward in length to be m in all other
trace logs, therefore the time complexity is O(mnL); Consequently, the trace-merge algorithm in this paper is very efficient
IV. TRACE LOG RECOGNITION AND COMPRESSION OF MPI PARALLEL JOBS

In order to scale down the original program to a Performance Skeleton by a certain proportion under different conditions, we need to recognize the loops in the trace logs for compressing. The trace sequence, which communication traces are produced by the same statement of the original program, can be identified to recognize the original program's loop structure, and then recover the loop structure. Multiple traffic records generated by the same statement can be merged together.

Compressing circulatory traces, which are abstracted to finding the optimal string compression, is still difficult to solve. This paper breaks it down into two steps: (1) Find out all of the consecutive repeated substrings; (2) Select a solution. This paper breaks it down into two steps: (1) Find out all of the consecutive repeated substrings, and each substring will be found more than once. Taking $S=AAAAB$ as example, substring $AAA$ will be recognized twice. First time is when $i=1$, and second time is when $i=2$. Obviously, only the shortest of loop body has the practical value. For this reason, we can ignore the certain substring in the second iteration through setting a flag.

B. Compression Cycle Algorithm

Set $S=S'$, repeat the procedures of constructing suffix arrays and searching for the maximal consecutive repeated substrings and markings, until one cannot find any consecutive repeated substrings. The algorithm using pseudo-code is shown in Algorithm 3.

Algorithm 3. Compression cycle algorithm.
Input: trace sequence $C$ after merging.
Output: array $R$ recording the number of compression.

1. Transform the trace sequence $C$ to string $S'$
2. do
3. $R.clear(); S=S'$
4. call the functions of constructing suffix array and RMQ preprocessing;
5. call the functions of searching for consecutive repeated substring;
6. $S=Sort(R)$; //mark the consecutive repeated substring by sequence
7. $comList.clear();$
8. for (enumerate the number $i$ of $R$)
9. {if ($R[i]$ does not overlap with the marked compression) $comList.push_back(R[i])$;}
10. $S=Compress(comList);$  
11. } while (NotEmpty($R$));

For a string $S$, set its length $|S|=n$, in the case of using the DC3 algorithm, each time suffix array construction time complexity is $O(n)$. The time complexity of constructing array $height$ is $O(n)$. The time complexity of the RMQ problem into the longest common prefix pre-processing algorithm is $O(n)$. From 1 to $n/2$, to enumerate the loop length, for each length $i$, the adjacent characters need to calculate are $ni$, total amount is $P=n/1+n/2+n/3+...+n/(n/2)$.

So, the time complexity of this step is $O(n\log n)$. string $S$. All maximal repeated substrings are no more than $O(n\log n)$. In the marking steps, each substring which has been stored will be traversed once, so the time complexity of the marking step is $O(n\log n)$. In summary, the total time complexity of the implementation of a suffix array construction, searching for $l$ the maximal consecutive repeated substring and marking is $O(n) + O(n) + O(n\log n) + O(n\log n) = O(n\log n)$. In repeating the above steps, the attention is to the shortening length of the new string. With each compression, the compression substring has at least two loop bodies, so the length is at least reduced by half. Therefore, the overall time complexity of the algorithm is $O(n\log n) + O(n/2\log(n/2)) + O(n/4\log(n/4)) + ... = O(n\log n)$.

V. PERFORMANCE SKELETON AUTOMATIC CONSTRUCTION

Each MPI communication trace, by recording the entry time, the execution time and exit time, can simulate two types of time information: First, communication time, through the implementation of time simulation; Second, the computation time (the interval between two communications
of a parallel program needs attention to synchronization communication, such as MPI_Wait, MPI_Barrier both belonging to the area of MPI communication within the bound of recorded time information will not be treated as computing time. By the gap between the start and end of two adjacent communications' time, and the difference in simulation, we get an estimate of computing time. In the case of the automated construction procedure, computing time is simulated by inserting an idle loop in each communication statement. Computing and communication time scales down with the number of cycles in the main communication. But in some programs, the main computing time may not appear in the loops, on the count of procedures simulated by an idle loop. Therefore, the main computing time outside the main loop can be scaled with the number of idle loops to achieve scaling function.

In the compression cycle, due to multiple statements combining into one, the corresponding computing time and communication time must be dealt with. Parameter values associated with the communication time of communication statements basically depends on count parameters. Communication statements including count can be divided into two categories, one is that the parameter value of count is the same or within a small range, the other is based on a rule about count parameter value varying upon the loop variable. For both cases, the count of joining the cycle, the inserted locations computing time will appear to be changed.

As shown in Figure 2, after adding circulations, the additional loop's start and end statements bring about a new interval in the original sequence. In these four delay statements before and after the two beginning loop statements, the delay statement S1 and E3 are outside the loop, and statement S2 and E2 are within the loop. Their delay time is calculated according to different needs. For all the delay statements (including S2 and E2) within the loop, according to the log records compress the interval between each of two communication statements, and averaged as the corresponding delay time of the location after compression; for the new additional delay statements outside the loop (S1 and E3), after S2 and E2 calculated, compensate for corresponding delay and the deviation of the log.

In compressing the point and point communication function log, the proposed algorithm preserves the information from each process in the process of sending and destination. Reconstruction of the case program needs to arrange the corresponding function parameters of each process according to this information. In order to have unified case proceedings, first obtain the value of the current process program immediately after executing MPI_Init parameters, and save in the global variable id_proc. In front of performing each point to point communication function, set a matrix to send Client / destination class parameter values according to the merging information recorded, and then call the communications function. The parameters in the corresponding position, according to the value of id_proc assignment matrix can be obtained. Another big problem of point to point communication function is avoiding deadlock. If using blocking sends and receives functions on both ends of the type of communication, you need to arrange the order of sending and receiving; if one side is using non-blocking functions, and the other side is using blocking functions, you need to make reasonable arrangements for sending and receiving functions and the MPI_Wait function order.

Record the necessary parameters only based on the type of MPI communication function when recording the running traces, for undocumented parameters completion requires accordance to the specific situation.

![Figure 2. Adding delay statements after cycle-compression](image)

VI. EXPERIMENTAL RESULTS AND PERFORMANCE ANALYSIS

This experiment was divided into two sections. Part 1 is correctness and time efficiency test for the running traces generation and section merged. Part 2 is the overall test based on the time predicted by the Performance Skeleton, and recording the running time of the original program.

A homogeneous cluster includes 16 homogeneous hosts, with the hardware configuration as follows: dual quad-core Intel Xeon (TM) 2.40GHz CPU, 2GB memory, using star network connectivity, bandwidth 1000Mbps. The MPI parallel library uses the MPICH 1.2.7 and MPICH2.NPB 3.3. The C compiler used is GCC 3.4.3, while the Fortran compiler selects F77. The random number generator uses randdp. Using a standard test set of 8 parallel NPB Application, and all of the NPB applications select C class size compiled.

A. Running Trace Merging and Performance Skeleton Generation

(1) The purpose of wrapper program construction testing is to check whether all the function calls can be recorded; the inserted wrapper functions are able to correctly compile the original program without affecting operating results; necessary record parameters are correctly recorded.

After 8 NPB applications experiments, by comparing MPI functions existing in the application source code and recorded in the log, we confirm that all wrapper functions are properly record. In the calculating size of 16 parallel processes, the amount of communication function calls is recorded in Table 1. Since some applications exist such as
unstructured point to point communication log inconsistencies, function call numbers of 16 processes is not the same. After finding the specified circumstances of different sizes, the numbers of traces of 8 NPB applications produce little difference (part of the applications log does not alter the calculation of the number of scale changes, such as IS and so on, another part of the applications log grew with the increased size of calculation, but the oscillation is not too great, such as LU), and the running time grows with the increased of the calculation growth size.

(2) The purpose of regularization and merging testing is to check the log after regularization and merging maintaining the original trace log appearance. The proposed algorithm is whether can be completed in a relatively short period of time to regularize and merge. The experimental results are shown in Table 2. This method can successfully eight regularize and merge application log, and the traces numbers are in line with the maximum process of function calls in Table 1. The longest combined time LU was 375.84 seconds, compared to the running time of LU itself is far less.

| TABLE I. THE NUMBER OF COMMUNICATIONS FUNCTION CALLS |
|----------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| Application    | BT            | CG            | EP            | FT            | IS            | LU            | MG            | SP            |
| Amount(maximum)| 17111         | 41954         | 5             | 47            | 38            | 324355        | 10043         | 26891         |
| Amount(minimum)| 17111         | 41954         | 5             | 47            | 36            | 162189        | 9329          | 26891         |

| TABLE II. RESULTS OF REGULARIZATION AND MERGING EXPERIMENT |
|----------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
| Application    | BT            | CG            | EP            | FT            | IS            | LU            | MG            | SP            |
| traces Amount after merged | 17111         | 41954         | 5             | 47            | 38            | 324355        | 10043         | 26891         |
| Merging time(s) | 20.22         | 49.68         | 0.02          | 0.12          | 0.08          | 375.84        | 12.57         | 35.47         |

B. Running Trace Merging and Performance Skeleton Generation

In this section, we re-run the Performance Skeleton scaling according with the different proportions under the original environment to get the prediction time compared with the actual time. Because of too little communication of EP, FT, IS applications, the experiment is not meaningful. The following experiments were on the other five C-class computing applications scale testing. In the homogeneous structure, generating the Performance Skeleton thorough running the original program, then reduce the Performance Skeleton 10 times, executing again in the first cluster set and finally getting the estimated time and the actual running time of the original program compared in Table 3.

As shown in Table 3, for the homogeneous cluster, time prediction error is less than 3% for 5 NPB applications.

<p>| TABLE III. RESULTS OF TIME PREDICTION IN HOMOGENEOUS CLUSTERS |
|----------------|---------------|---------------|---------------|---------------|</p>
<table>
<thead>
<tr>
<th>Application</th>
<th>Initial program running time(s)</th>
<th>Performance Skeleton running time(s)</th>
<th>Prediction time(s)</th>
<th>Deviation estimation (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT</td>
<td>1 094.65</td>
<td>113.04</td>
<td>1 130.4</td>
<td>1.04</td>
</tr>
<tr>
<td>CG</td>
<td>384.81</td>
<td>37.93</td>
<td>379.3</td>
<td>−1.43</td>
</tr>
<tr>
<td>LU</td>
<td>877.27</td>
<td>85.16</td>
<td>851.6</td>
<td>−2.92</td>
</tr>
<tr>
<td>MG</td>
<td>786.34</td>
<td>77.57</td>
<td>775.7</td>
<td>−1.35</td>
</tr>
<tr>
<td>SP</td>
<td>1 219.16</td>
<td>118.56</td>
<td>1 185.6</td>
<td>−2.76</td>
</tr>
</tbody>
</table>

VII. CONCLUSIONS

This article delves into the run time prediction of parallel programs based on a skeleton program in all aspects, and designs an effective method for estimating time under different computing resources, to avoid the defects and limitations of traditional methods based on history and modeling analysis providing the basis for the task-level scheduling of distributed computing. (1) By studying the features of set communication and one to one communication, we design the trace log regularization and merging algorithm for parallel programs. (2) For compressing the circulatory traces, which is the most central and difficult, this paper converts it into a circular sub-string compressing problem, and proposes an algorithm based on the suffix array. Its algorithms time complexity is $O(n \log n)$ better than the existing $O(n^3)$ algorithm. (3) To automatically reconstruct the Performance Skeleton, it solves the scalable problem of calculation and communication time.

Experimental results show that these methods can accurately estimate the running time of computing jobs. The error is less than 3% for homogeneous clusters.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work is...
supported by National Natural Science Foundation of China (NSFC) under grant No. 61173145, the National Grand Basic Research Program (973 Program) of China under grant No. 2011CB302605, the National High Technology Research and Development Program of China under Grant No. 2010AA012504 and 2011AA010705, Albert Cheng is supported in part by the United National Science Foundation under Award No. 0720856.

REFERENCES