ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, Supplement, December 2010, pp.238−250
© by Institute of Software, the Chinese Academy of Sciences. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于判例构造的并行作业性能预测*

张伟哲+, 张宏莉, 张元竞

(哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

Parallel Job Performance Prediction Based on the Case Reconstruction

ZHANG Wei-Zhe+, ZHANG Hong-Li, ZHANG Yuan-Jing

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

+ Corresponding author: E-mail: wzzhang@hit.edu.cn, http://pact518.hit.edu.cn

Zhang WZ, Zhang HL, Zhang YJ. Parallel job performance prediction based on the case reconstruction. *Journal of Software*, 2010,21(Suppl.):238−250. http://www.jos.org.cn/1000-9825/10025.htm

Abstract: Accurate prediction of the running time of parallel jobs under different computing resources is the foundation of many job scheduling approaches. A job performance prediction method based on the Performance Skeleton is proposed to avoid the inaccuracy of historical and modeling analysis prediction methods in heterogeneous clusters. To record the running trace, a method is designed to access all communication traces during the runtime. To merge these traces, this paper designs a trace-merge algorithm to structure the communication traces. To compress 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 performance is theoretically and practically better than the existing algorithms. 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, and 10% for heterogeneous clusters.
Key words: network computing; parallel job; performance prediction; case program; circular sub-string compressing

摘 要: 针对基于MPI的并行作业性能预测问题,鉴于历史预测与建模分析方法在异构网络计算环境中性能预测的局限,提出了基于判例构造的并行作业性能预测方法.在MPI库PMPI接口中插入封套函数,获取通信日志,并设计了日志规整和合并算法.将最核心的日志循环收缩问题,转化为字符串循环子串收缩问题,提出了一种基于后缀数组算法,在理论和实际的性能方面均优于已有算法;判例程序自动构建阶段,解决了计算时间与通信时间等比例缩放问题,设计了自动构建可执行判例程序的方法.同构与异构机群环境实验结果表明,判例预测方法能够比较准确地预估计算作业的运行时间,对于同构机群误差不超过 3%,异构机群误差不超过 10%,与同类算法相比,具有较好的综合性能.
关键词: 网络计算;并行作业;性能预测;判例程序;循环子串收缩

* Supported by the National Natural Science Foundation of China under Grant No.60703014 (国家自然科学基金); the National Basic Research Program of China under Grant No.G2011CB302605 (国家重点基础研究发展计划(973))

Received 2010-06-15; Accepted 2010-12-10

© 中国科学院软件研究所 http://www.jos.org.cn

随着近年来相关硬件和软件技术的发展,大规模并行计算已经成为国内外具有重要意义的研究课题.封闭于单一组织的超级计算机中执行并行作业的数目与规模不断增加,跨越多个组织与管理域的网格计算、多机群共享与协同工作也逐渐成为大规模并行计算的主流[1,2].因此,无论是单一组织的超级计算机的内部并行作业调度,还是同构乃至异构的多机群共享与协同工作的任务调度,都具有十分重要的研究意义.

并行作业在不同计算资源之下的运行时间的预估是高效资源管理与任务调度方法的基础.性能(运行时间等)预测通过预估任务在不同资源条件下的开销为任务调度服务.对于高效的资源调度来说,无论是提高系统的整体吞吐率,还是按优先级使用户程序达到最优调度,准确的运行时间预估都是必不可少的.通常的性能预测方法可以分为两类:即基于历史的预测和理论预测(基于建模分析的预测).

基于历史的预测是根据同一应用程序在某些计算资源条件下运行过的结果,按照一定原则进行资源分配;基于建模分析的预测是根据已有的机群性能模型和应用的编程模型,对不同资源下程序运行的时间进行预估,据此进行资源分配.上述两种方法的资源分配的主要原则都是减少等待时间和避免饥饿.

基于历史的预测基于先前的运行结果,面向给定数目的资源和相对固定的应用;难以满足网络计算环境的要求.理论预测基于应用的编程模型和性能模型,以及计算环境的性能模型,预测结果依赖于对应用程序的理解和性能模型的准确程度,适用面广泛但网络环境下精度需进一步提高.

基于建模分析的预测,在同构的计算环境下相对简单,主要是由于计算资源的模型固定,并且在分配资源时,也容易根据模型估计程序的运行时间.相比之下,异构的、网络共享式的计算环境将使问题变得复杂很多,一些已有的研究在使用建模分析的同时,还使用网络测量等技术对异构的计算环境中的一些性能参数进行理论预测[3,4].

鉴于历史预测与建模分析方法在异构的网络计算环境中的局限性,近年来人们提出了一种利用判例程序(performance skeleton,也被译作"模版程序"或"程序骨骼")的思路[5].其核心思想是:最接近准确的预测方法莫过于使用一个能够表征原始程序的计算特征的短小程序,在指定的大规模并行计算环境之下实测一次.将实测判例程序的时间同比例缩放,获得原始程序的预估时间,达到预测的准确性和时间复杂度的降低.由于这种方法选取代表资源和有限规模的应用实测性能,因此相比基于历史的预测具有灵活性,相比理论预测又具有准确性.

本文第 1 节分析基于判例构造的并行作业预测研究现状及其算法存在的局限性.第 2 节提出 MPI 并行作业日志收集与合并技术.第 3 节提出日志中循环与收缩这一核心问题,并将其转化为字符串的循环子串收缩问题,提出一种基于后缀数组的收缩算法.第 4 节提出并行作业判例程序自动创建技术.第 5 节分别在同构与异构平台上给出实验结果及性能分析.第 6 节对本文进行总结.

1 相关工作

1989 年,Cole 提出了"算法模板(algorithmic skeletons)"的概念,其最初的目的是为一些常用的并行算法设计固定的程序模板[6],是与判例程序相关的最早的学术研究.另一方面,在研究分布式计算任务运行时间预估的领域,学术界提出了在模拟原始作业程序性能特点的前提下,缩短运行时间的预估方法.1993 年,Dikaiakos 等人使用被称为 FAS(functional algorithm simulation)的方法扩展了对并行程序模拟的范围,在大多数计算不被实际运行的情况下进行模拟.该方法针对一些并行科学计算算法的计算特征进行生成和收集,并得到保留这些计算特征的轻量级预测程序[7].Dikaiakos 等人还建立了原型系统 FAST,采用收集外部信息的方法,而不必关心映射策略和硬件的选择,对大规模并行计算程序性能进行了预测[7].这种方法可以看作是判例技术的前身.

1999 年,Dinda 等指出,应用程序的运行时间与负载是紧密相关的[8].因此需要关心的核心问题是负载是否是稳定可预测的,文献[8]同时还证明了负载是稳定可预测的,为基于判例的预测技术奠定了基础.

2004 年,Sodhi 等人提出了一种基于判例程序进行程序性能预估的思路[5],其名称 Performance Skeletons 或许即来源于 Algorithmic Skeletons,Performance 一词表明这种"骨骼"代码已经不具有原始程序的功能了,只是在"性能"方面的表现与原始程序相似.而获取这种判例程序的方法是:先将原始程序在指定资源环境下运行一次,收集它的每个进程的运行日志,再将来源于不同进程的日志设法合并,再由日志反推出程序,并识别出程序的主

© 中国科学院软件研究所 http://www.jos.org.cn

干循环,通过按比例缩小循环次数,减小数据传输量,减小计算时间等方法最终获得判例程序.随后 Xu 等人按照这一思路做了许多工作[9−11],也明确了识别与收缩日志循环是自动构建判例程序技术中的核心问题.此外,有一些关于并行程序的研究与自动构建判例程序有相似之处,如 Lu 等人提出了一种使用曲线拟合的方法压缩并行程序的代码,使程序的运行时间能够大幅缩短,而又在一定程度上保持原有程序的计算特征[12].Sherwood 等人研究了一种自动分析并行程序周期性的方法[13].但这些研究的目的一般都与构造判例程序预测原始程序的运行时间无关.

目前为止对基于判例预测程序性能的研究较少,且上述方法有很多不完善之处.最为重要的是前述的研究工作都是基于同构机群来实现的,没有考虑异构的适应性.此外,合并一对一通信时,能够合并的同一条记录通常具有不同的 source/dest 类参数,Xu 等人提出的解决方法是使用通信模式数据库匹配应用程序的通信模式,这种方法的代价较大,而且只能处理一个计算程序中只使用一种通信模式的情况.又如,文献[9,11]中的解决识别和收缩日志中循环的两种算法,一种在时间性能上较差,当日志数量达到 10^5 以上时几乎无法使用;另一种在识别和收缩长度的效果方面表现不佳.本文将对判例预测技术进一步深入研究,对其存在问题提出新的高效解决方法.

2 MPI 并行作业运行日志的获取与合并

在生成判例程序的步骤中,首先需要将原始程序运行一次,获取运行时的日志信息.由于获取的日志来自 SPMD 程序的多个进程,而后需要将多份日志合并成一份.目的是获取原始程序的计算特征和并对其规格化处理工作,是判例程序创建基础.

2.1 运行日志的获取

MPI 作业使用标准 MPI 函数进行进程间通信工作,收集 MPI 函数调用日志后,即可得到程序关于通信的全部信息,建立能够充分反映程序特征的判例程序.另一方面,有限而又标准化的 MPI 函数库为获取 MPI 函数的调用日志提供了条件.

分析 MPI 函数库实现发现,MPI 函数都预设标准分析接口 PMPI[14],对于所有的以“MPI_”为前缀的 MPI 函数,存在一个以“PMPI_”为前缀、参数以及功能完全相同的函数.可以设计封套函数(wrapper function),将用户程序中对“MPI_”前缀类型的函数调用引入自己实现的函数库中,截获调用时的参数以及时间信息,然后再原样调用一次“PMPI_”前缀类型的函数,以实现用户程序原有功能.

通过 MPI_Init 和 MPI_Finalize 函数的函数封套,获取程序开始和结束的信息,为每个进程维护一个记录日志的文件.对其余每个被应用程序调用的函数,在封套函数中记录它们每次调用时对于构造判例函数有意义的参数取值.(1) 不需要记录参数的函数:MPI_Init,MPI_Finalize,MPI_Wait,MPI_Barrier 等.此类函数没有任何参数,或者参数值在生成判例程序时与运行时的日志无关,不需要收集日志时输出参数.(2) 一对一阻塞发送和接收函数:MPI_Send,MPI_Recv;以及一对一非阻塞发送和接收函数,即 MPI_Isend 和 MPI_Irecv.需要记录:count,datatype 和 dest/source.(3) 非阻塞通信的完成检测函数:MPI_Waitall 等.需要记录 count.(4) 广播类通信函数:MPI_Bcast,MPI_Alltoall,MPI_Alltoallv,MPI_Reduce,MPI_Allreduce 等.这些函数是一对多或者多对多.MPI_Bcast 是最基本的一对多广播,需要记录的:count,datatype,root.MPI_Alltoall 以及 MPI_Alltoallv 类似于给每个进程都做一次 MPI_Bcast 的广播操作,没有 root 参数但是需要多记一个接收方向的 count.MPI_Reduce 以及 MPI_Allreduce 则是在 MPI_Bcast 的基础上再作一次数学计算,因此要多记一个运算的类型,需要记录 count, datatype, op, root.

NPB 并行的编译选项需要通过修改 config 目录下的配置文件 make.def 实现.以 IS 应用为例,取 *Class*=*A*,进程数为 2,将自动生成的带有 MPI 封套的 MPI 动态库加入编译后,得到的运行时日志如图 1 所示.

© 中国科学院软件研究所 http://www.jos.org.cn

```
Trace: Rank=0 Function: MPI_Allreduce
Trace: Parameters: = ( void sendbuf, void recvbuf, int* count, MPI_Datatype* datatype,
Trace: Paravalues: = ( void sendbuf= 185894752, void recvbuf= 185886464, int count= 102
Trace: Starttime = [ 1273479811221585 ]
Trace: Endtime = [ 1273479811222023 ]
Trace: Durtime = [ 438 ]
Trace: Rank=0 Function: MPI_Alltoall
Trace: Parameters: = ( void sendbuf, int* sendcount, MPI_Datatype* sendtype, void recvb
Trace: Paravalues: = ( void sendbuf= 185898880, int sendcount= 1, MPI_Datatype sendtype
Trace: Starttime = [ 1273479811222058 ]
Trace: Endtime = [ 1273479811222118 ]
Trace: Durtime = [ 60 ]
Trace: Rank=0 Function: MPI_Alltoallv
Trace: Parameters: = ( void sendbuf, int sendcnts, int sdispls, MPI_Datatype* sendtype,
Trace: Paravalues: = ( void sendbuf= 185919392, int sendcnts= 185898880, int sdispls= 1
Trace: Starttime = [ 1273479811222137 ]
Trace: Endtime = [ 1273479811376025 ]
Trace: Durtime = [ 153888 ]
Trace: Rank=0 Function: MPI_Allreduce
Trace: Parameters: = ( void sendbuf, void recvbuf, int* count, MPI_Datatype* datatype,
Trace: Paravalues: = ( void sendbuf= 185894752, void recvbuf= 185886464, int count= 102
Trace: Starttime = [ 1273479811513798 ]
Trace: Endtime = [ 1273479811514089 ]
Trace: Durtime = [ 291 ]
...
```

(a) *ProcRank*=0

```
Trace: Rank=1 Function: MPI_Allreduce
Trace: Parameters: = ( void sendbuf, void recvbuf, int* count, MPI_Datatype* datatype,
Trace: Paravalues: = ( void sendbuf= 185894752, void recvbuf= 185886464, int count= 10
Trace: Starttime = [ 1273479811210098 ]
Trace: Endtime = [ 1273479811222022 ]
Trace: Durtime = [ 11924 ]
Trace: Rank=1 Function: MPI_Alltoall
Trace: Parameters: = ( void sendbuf, int* sendcount, MPI_Datatype* sendtype, void recv
Trace: Paravalues: = ( void sendbuf= 185898880, int sendcount= 1, MPI_Datatype sendtyp
Trace: Starttime = [ 1273479811222059 ]
Trace: Endtime = [ 1273479811222129 ]
Trace: Durtime = [ 70 ]
Trace: Rank=1 Function: MPI_Alltoallv
Trace: Parameters: = ( void sendbuf, int sendcnts, int sdispls, MPI_Datatype* sendtype
Trace: Paravalues: = ( void sendbuf= 185919392, int sendcnts= 185898880, int sdispls=
Trace: Starttime = [ 1273479811222149 ]
Trace: Endtime = [ 1273479811374890 ]
Trace: Durtime = [ 152741 ]
Trace: Rank=1 Function: MPI_Allreduce
Trace: Parameters: = ( void sendbuf, void recvbuf, int* count, MPI_Datatype* datatype,
Trace: Paravalues: = ( void sendbuf= 185894752, void recvbuf= 185886464, int count= 10
Trace: Starttime = [ 1273479811511235 ]
Trace: Endtime = [ 1273479811514092 ]
Trace: Durtime = [ 2857 ]
...
```

(b) *ProcRank*=1

Fig.1 Fragment of running traces (*IS*, *Class*=*A*)

图 1 运行日志(*IS*,*Class*=*A*)的部分片段

2.2 运行日志的合并

为了生成 SPMD 的判例程序,需要将来自每个进程的日志合并成一份,再根据该日志反推出判例程序.必须指出的是,基于 MPI 的应用程序虽然是 SPMD 的并行应用程序,但每个进程的行为并不完全相同,来自不同进程的日志也有所差别.为了尽可能地使判例程序能够模拟原始应用程序的计算特征,必须识别出 SPMD 程序在运行过程中各进程执行不一致的内容.

SPMD 程序在运行过程中各进程执行的不一致包含两种情况:1) 顺序差别:每个进程都在收到前一个进程发出的信息后,经过一些计算,再向后一个进程发送信息,最后,编号为 $n-1$ 的进程再将信息发给本次通信的起点——编号为 0 的进程.0 号进程是先执行函数 MPI_Send,再执行函数 MPI_Recv,而其他进程却都是先执行 MPI_Recv,再执行 MPI_Send.更常见的一类顺序差别是由非阻塞通信造成的.例如,每个进程都使用阻塞通信的 MPI_Send 来发送信息,使用非阻塞通信的 MPI_Irecv 来接收消息,并且每个非阻塞接收操作都有一个对应的 MPI_Wait 函数.2) 调用的函数不同:某个或者某些进程调用了某个 MPI 函数,而其他进程没有调用.某些进程只有发送动作而没有接收动作,而其他进程只有接收动作而没有发送动作.且随着通信模式的愈加复杂,造成调用函数不同的情形也变得更加复杂.

解决顺序差别问题需要首先对来自所有进程的日志可能出现顺序差别的通信段落进行预处理,将原有的日志调整顺序.其原则是:接收语句放在发送语句之前.也就是说对于存在相互收发关系的 MPI_Recv 和

© 中国科学院软件研究所 http://www.jos.org.cn

MPI_Send 语句,一律将 MPI_Send 放在前面;对于存在相互收发关系的 MPI_Irecv,MPI_Send 以及附带的 MPI_Wait 语句,一律将 MPI_Irecv 放在前面,随后跟着 MPI_Wait 语句,最后是 MPI_Send 语句.如算法 1 所示

算法 1. 处理顺序差别的算法.

输入:日志序列集合 ***T***.

输出:经过调整顺序后的日志序列集合 ***T***.

```
1. 初始化日志序列集合 T;
2.  for (枚举每个进程编号 i)
3.   {s=0; tmpT.clear();
4.     for (枚举 T[i]的每条日志编号 j)
5.     { if (T[i][j].func==MPI_Send or T[i][j].func==MPI_Recv)
6.         { k=j; recvList.clear(); sendList.clear();
7.            while ((T[i][k].func==MPI_Send or T[i][k].func==MPI_Recv) and Match(T[i][k], T[i][j]))
8.              {if (T[i][k].func==MPI_Send) {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.  }
```

解决调用函数不一致问题需经过前文描述的预处理算法 1,对所有的顺序不一致的情况进行了调整.此时,在按照时间顺序合并日志时,如果遇到在某个位置,来源于不同进程的日志不能合并,说明一定遇到了调用函数不同的情况.此时处理思想是:正确识别出来自哪个进程的当前一条日志应当处于判例程序的相对靠前位置,将能够与它合并的日志合并.为此,算法 2 设计一个最大向下追溯位置参数 m,如果某条日志在其他进程的当前位置乃至向下 m 条都不能合并,就说明它应当处于判例程序的相对靠前位置.

算法 2. 处理函数调用差别的算法.

输入:调整顺序后的日志序列集合 ***T***.

输出:合并后的日志序列 ***C***.

```
1. 初始化日志序列集合 T; 初始化指针数组 p=0;
2.  while (p 中有指针尚未到达对应日志序列末尾) do
3.   {if (当前指针对应的所有条目可以合并) {C.push_back(combine(T,p));Add(p);}
4.    else {Max=0;
5.          for (i 枚举每个进程编号)
6.           {计算第 i 个进程向下匹配的距离 d[i];
7.              if (d[i]>Max) {Max=d[i];j=i;}
8.           }
9.          for (i 枚举每个进程编号)
10.          {q.clear();
11.             if (Match(T[i][p[i]]),T[j][p[j]]) {q.push_back(p[i]);}
12.          }
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
13.         }
14.     for (i 枚举每个进程编号)
15.       {if (进程 i 在本次循环被合并) p[i]++;}
16.         C.push_back(combine(T,q)); Add(q);
17.       }
18.   }
```

由于日志的数量可能很大(对于 NPB 3.3 来说,日志数量最大的应用 LU 的条目数为 10^6 的数量级,实际应用可能更大),日志合并算法的时间效率是值得关注的问题.设进程数为 n,来自每个进程的日志数量中,最大值为 L.预处理时需要对每条日志遍历一次,时间复杂度为 $O(nL)$;合并时最坏情况下对每个条目需要在其他所有日志中向下追溯长度为 m 的距离,因此时间复杂度为 $O(mnL)$.可以看到本文的日志合并算法的效率很高.

3 MPI 并行作业日志循环的识别与收缩

为了能够使判例程序达到与原始程序相比"按比例缩小"的效果,并且能够在不同环境下的性能测试中按一定比例"缩放",需要识别出日志中的循环部分并加以收缩.通过日志序列,判断出哪些通信日志是由原始程序的同一条语句产生的,据此识别出原始程序的循环结构,并将循环结构复原,将同一条语句产生的多条通信记录合并到一起.

识别和收缩日志中的循环可以说是自动构建判例程序中最核心的问题,也是最难解决的一个问题.本节首先将这一问题抽象化:将已经合并后的日志中的每一条对应到一个字符,则日志对应一个字符串;由此,日志中的循环就是字符串中连续出现的重复子串;由于循环可以嵌套,因此这种重复子串也是可以嵌套的;问题就抽象为在字符串中寻找和收缩连续重复子串的问题.例如$(AB)^2$表示字符串 $ABAB$ 收缩后的结果.对于同一个字符串可能存在多种不同的收缩方案.例如字符串 $ABCABCABCA$ 就至少存在$(ABC)^3A$ 和 $A(BCA)^3$、$(ABC)^2ABCA$ 等几种收缩方案.为了设计一种自动识别和收缩循环的算法,需要先确立一个对各种收缩方案评价标准.

本文定义最优收缩标准如下:(1) 一个最优的收缩方案,必须是不能再收缩的;(2) 以收缩方案中字符的总个数作为该收缩方案的长度(不包括表示循环的括号和表示循环次数的数字),长度最短的方案为最优.根据以上两点,并列最优的情况是可能出现的,在上面的例子中,$(ABC)^3A$ 和 $A(BCA)^3$ 两个方案的长度均为 4,都是最优方案.

收缩日志中循环的问题抽象为寻找字符串最优收缩方案的问题后求解仍然比较困难,本文再将它分解成两个步骤:(1) 寻找出字符串所有的连续循环子串,又进一步可分为构造后缀数组和最长公共前缀的连续重复字串识别;(2) 从已经找到的连续循环集合中,选择一部分进行收缩,以达到整体上收缩长度最短的效果.

3.1 构造后缀数组

后缀数组是寻找出字符串所有的连续循环子串中的基础数据结构.最基本的后缀数组包括后缀数组 SA 和名次数组 $Rank$ 两个部分.对于字符串 S,本文定义 $Suffix(i)$表示字符串 S 的从位置 i 开始的"后缀",即 S 的从第 i 个字符到末尾的子串.对于不同的两个后缀 $Suffix(i)$和 $Suffix(j)$,可以以字典序来比较它们的大小.后缀数组 SA 是将 S 的 n 个后缀排序后,按照从小到大的顺序将它们的序号排列的一个数组.而名次数组 $Rank$ 则是记录每个后缀在 SA 中所排的位置.可以看到,$Rank$ 和 SA 互为逆运算,即 $SA[Rank[x]]=x$.

构造后缀数组后,本文定义 $height$ 数组和 h 数组如下 $height[i]$为 $suffix(SA[i])$与 $suffix(SA[i-1])$的最长公共前缀.$h[i]=height[Rank[i]]$.基于 $height$ 数组,就可以高效地解决"后缀的最长公共前缀问题"了.该问题转化为"RMQ 问题(range minimum query,多次查询区间内最小元素问题)"之后,存在 $O(n)-O(1)$时间复杂度的算法,即以 $O(n)$的时间预处理之后,对于每次查询,可以在 $O(1)$的时间计算出结果[15−17].是本文后续算法的基础.使用伪码表示的构造后缀数组以及 RMQ 预处理算法 3 所示.

算法 3. 构造后缀数组以及 RMQ 预处理算法.

输入:字符串 S.

输出:完成预处理的 *SA*,*Rank*,*height*,*m*,*best* 数组.

```
1.    DC3(S,SA,Rank);    //对字符串 S,使用 DC3 算法构造 SA 和 Rank 数组
//以下构造 height 数组
2.  k=0;
3.  for (i=0; i<n; i++)
4.      { height[Rank[i]]=k; i++;
5.          if (k>0) k−−;
6.          for (j=SA[Rank[i]−1];r[i+k]==r[j+k];k++)}
//以下预处理 RMQ 的 m 和 best 数组
7.  for (i=1;i<=n;i++) {RMQ[i]=height[i];}
8.  m[0]=−1;
9.  for (i=1;i<=n;i++)
10.     if (i&(i−1))==0) {m[i]=m[i−1]+1;}
11.       else {m[i]=m[i−1];}
12.   for (i=1;i<=n;i++) {best[0][i]=i;}
13.   for (i=1;i<=m[n];i++)
14.     { for (j=1;j<=n+1−Sqr(2,i); j++)
15.         { a=best[i−1][j]; b=best[i−1][j+Sqr(2,i)];
16.             if (RMQ[a]<RMQ[b]) {best[i][j]=a;}
17.               else {best[i][j]=b;}
18.           }
19.     }
```

3.2 最长公共前缀问题连续重复子串查询

设完整字符串为 S,其长度$|S|=n$,对于每个连续重复子串来说,其长度必定不超过 $n/2$.从 1 到 $n/2$ 枚举这个长度 i,然后求所有循环节的长度为 i 的连续重复子串.在 i 的枚举完成后,即可求出所有的极大连续重复子串,以及每个子串的循环节.

设完整字符串为 S,其长度$|S|=n$,对于每个连续重复子串来说,其长度必定不超过 $n/2$.从 1 到 $n/2$ 枚举这个长度 i,然后求所有循环节的长度为 i 的连续重复子串.不难想象,每个循环节的长度为 i 的连续重复子串必然跨越 $S[0],S[i],S[i\times 2],S[i\times 3]$…中的某两个连续字符,而且这两个字符必定分属于相邻的两个循环节中.因此,在存在连续重复子串的情况下,这两个字符必然是相同的,这个条件是必要非充分的.所以对于每个 i,只需要检查所有的 $S[0],S[i],S[i\times 2],S[i\times 3]$…字符,仅当发现有连续两个相同字符时,才有可能存在一个跨越过这两个字符的一个循环节长度为 i 的极大连续重复子串.当发现对于某个位置 j,有 $S[j]=S[j+i]$时,查询向前和向后各自能匹配多远.

在算法 3 的基础上,可以经过 $O(n)$的预处理,本文算法可以每次以 $O(1)$的时间代价查询,得到向后匹配的最大长度.向前匹配可以用将 S 反转的方法做.这样就找出了所有极大连续重复子串,并标记出了循环节.向前匹配可以用同样的方法做,即在预处理时,将 S 反转,也建立相应的后缀数组和 *height* 数组,这样就每次可以在 $O(1)$的时间内求出向前匹配的最大长度了.这样,在 i 的枚举完成后,即可求出所有的极大连续重复子串,以及每个子串的循环节.

需要注意的是,某些连续重复子串会被多次发现,例如在 S=*AAAAB* 时,子串 *AAAA* 会被发现两次,第 1 次是在 i=1 的时候,第 2 次是在 i=2 的时候.显然,只有循环节最短的一次是具有实际价值的,因此可以通过设标志位的方法,在某个子串第 2 次被发现时直接将其忽略.

© 中国科学院软件研究所 http://www.jos.org.cn

3.3 收缩循环算法

在使用上节的算法获得字符串 *S* 的所有极大连续重复子串之后,按照由长到短的顺序在整体字符串中标记可收缩的连续重复子串,在长度相同时,优先标记靠左边的子串.对于与已标记的子串相重叠的连续重复子串则跳过.随后,根据记录下的对应的循环节,收缩所有被标记的子串,得到新串 *S*′.

令 *S*=*S*′,重复上述的构造后缀数组、寻找极大连续重复子串和标记的步骤,直到找不到任何连续重复子串为止.用伪码描述的算法 4 如下所示:

算法 4. 收缩循环算法.

输入:合并后的日志序列 ***C***.

输出:记录收缩情况的数组 ***R***.

```
1. 将日志序列转为字符串 S′
2. do{
3.      R.clear(); S=S′;
4.      调用构造后缀数组以及 RMQ 预处理算法;
5.      调用寻找连续重复子串的算法
6.      Sort(R);   //按照由长到短的顺序标记可收缩的连续重复子串
7.      comList.clear();
8.      for(枚举 R 的编号 i)
9.          {if (R[i]未与已标记收缩重叠) comList.push_back(R[i]);}
10.     S′=Compress(comList);
11.   } while (NotEmpty(R));
```

对于字符串 *S*,设其长度$|S|=n$,在使用 DC3 算法的情况下,每次构造后缀数组的时间复杂度为 $O(n)$.构造 *height* 数组的时间复杂度为 $O(n)$.转化为 RMQ 问题的最长公共前缀算法的预处理的时间复杂度为 $O(n)$.由 1 至 $n/2$ 枚举循环节长度时,对于每个长度 *i*,需要计算的相邻字符有 n/i 个,总共有 $P=n/1+n/2+n/3+...+n/(n/2)$个,因此这一步的时间复杂度为 $O(n\log n)$.字符串 *S* 的所有极大重复子串的数量不超过 $O(n\log n)$,在标记步骤,每个已被存储的子串会被遍历一次,因此标记的步骤的时间复杂度为 $O(n\log n)$.

综上,执行一次构造后缀数组、寻找极大连续重复子串和标记操作的总的时间复杂度为 $O(n)+O(n)+O(n\log n)+O(n\log n)=O(n\log n)$.在重复上述步骤时,注意新字符串的长度在缩短.由于每次收缩时,被收缩的子串至少具有 2 个循环节,因此其长度至少缩短一半.因此,整体上算法的时间复杂度为 $O(n\log n)+O(n/2\log(n/2))+O(n/4\log(n/4))+\ldots=O(n\log n)$.

4 MPI 并行作业判例程序自动创建

通过记录每条 MPI 通信语句的进入时间、执行时间和退出时间,可以模拟两类时间信息:一是通信时间,通过执行时间来模拟;二是计算时间(指并行程序在进行两次通信之间的间隔时间,在这段时间通常并行程序在进行计算,注意通信中的各种同步,如 MPI_Wait,MPI_Barrier 都属于 MPI 通信的范畴,已经属于被封套程序记录时间信息的范围,不会被当作计算时间),通过相邻两次通信的结束时间与开始时间的差来模拟.获得计算时间的估计值后,在自动构建判例程序时,由程序在每两条通信语句之间插入空循环模拟计算时间.

计算时间和通信时间都是可以随着主要通信语句所在循环的循环次数进行缩放的.但是在某些程序中,可能出现主要的计算时间并不处在循环中的情况,由于对于计算时间,在生成的判例程序中通过空循环进行模拟,因此,在主要循环之外的计算时间是可以随着这些空循环的循环次数进行缩放的.综上,就实现了判例程序按比例缩放的功能.

在收缩循环时,由于多条语句被合并成一条,因此对相应的“计算时间”和“通信时间”都必须作出处理.参数取值与通信时间相关的通信语句,其通信时间基本都取决于 *count* 类参数.这些带有 *count* 类参数的能够合并的

通信语句又可以分为两类,一类是 *count* 类参数值相同或者在某一个很小范围内变化的,另一类是根据某项规则,*count* 类参数值随着循环变量按照规则变化的.对于这两种情况,在加入循环的情况下,都会出现"计算时间"的插入位置被改变的情况.

如图 2 所示,在增加循环后,引入的循环开始语句和循环结束语句在原有的通信语句序列中增加了新的间隔空位.在这两条循环起始语句的前后 4 个延时语句之中,延时语句 *S*1 和延时语句 *E*3 是在循环之外的,而延时语句 *S*2 和延时语句 *E*2 是在循环之内的.它们的延时时间需要按照不同的方式计算.对于循环之内的所有延时语句(包括 *S*2 和 *E*2),按照日志记录中收缩前每两条通信语句之间的间隔时间,取平均值作为对应的收缩后位置的延时时间;对于新增加的循环外的延时语句(即 *S*1 和 *E*3),在 *S*2 和 *E*2 被计算出来以后,按照相应位置的延时与日志中的偏差进行补偿.

在合并点对点类通信函数日志时,本文算法保留了来自每个进程的日志中的发送端和目的端进程的信息,在重建判例程序时需要根据这些信息安排每个进程的对应的函数参数.为了使生成的判例程序具有统一形式,首先在判例程序中执行 *MPI_Init* 参数之后立刻取得当前进程的值,保存在一个全局变量 *id_proc* 中.在每个点对点通信函数执行前,安排一个发送端/目的端类参数的矩阵,按照合并时记录的信息分别赋值,随后在调用通信函数时,在相应的参数位置上,根据 *id_proc* 的值获取矩阵中的赋值即可.点对点类通信函数的另一大问题是避免死锁.如果收发两端使用的都是阻塞类通信函数,需要合理安排收发的顺序;如果一端使用的是非阻塞类函数,另一端使用的是阻塞类函数,则需要合理安排收发函数以及 MPI_Wait 函数的顺序.在判例程序初步生成之后再扫描一遍程序,对上述情况进行检查,作出相应调整即可.

在记录运行日志时只根据 MPI 通信函数的类别记录了生成判例程序时必要的参数,对于未记录的参数,需要依据具体情况进行补全.

Fig.2 Adding delay statements after repeat-compression

图 2 收缩循环后增加延时语句示意图

5 实验结果及性能分析

本节实验分为两部分.第 1 部分是对于运行日志生成与合并部分的正确性和时间效率的单元测试.第 2 部分是基于判例的时间预测整体测试,记录原始程序的运行时间,并按照本文的方法生成判例程序,使用判例程序按照不同的缩放比例在原始环境下重新运行,用得到的"预估时间"与实际时间进行对比,用两者之间的偏差来评价本文方法.

实验环境分为同构与异构机群.同构机群中包括 16 台同构主机,硬件配置均为双路四核 Intel Xeon(TM) 2.40GHz CPU,2GB 内存,使用星型网络连接,带宽为 1 000Mbps.异构机群由 16 台异构的主机构成,其中 4 台主机为双核 Intel Pentium III (Coppermine) 866MHz CPU,另外 12 台主机为双核 Intel Pentium III (Coppermine) 1GHz CPU,16 台主机均为 2GB 内存.16 台机器分布在 4 个子网中,每个子网内部为星型网络连接,带宽为 1 000Mbps,4 个子网之间为星型网络连接,带宽为 100Mbps.

© 中国科学院软件研究所 http://www.jos.org.cn

MPI 并行库使用了 MPICH 1.2.7 以及 MPICH2.NPB 的版本为 3.3.C 编译器选择 GCC 3.4.3,Fortran 编译器选择 F77.随机数发生器使用 randdp.使用标准测试集 NPB 中的 8 个并行应用程序,所有的 NPB 应用程序选择 C 类规模进行编译.

5.1 运行日志合并与判例生成实验

(1) 封套程序构造测试目的是检查是否所有的函数调用都能够被记录;插入的封套函数是否能够使原有的程序正确编译,且不影响运行结果;需要记录的参数是否都被正确记录.

经过对 NPB 的 8 个应用程序的实验,通过扫描应用程序源代码中出现的 MPI 函数与被记录的日志中 MPI 函数的对比,确认所有的函数封套都能够正确记录.在计算规模为 *C*,并行进程数为 16 的情况下,各程序被记录的通信函数调用次数见表 1,由于某些应用中的非结构化点对点通信等进程间日志不一致情况,16 个进程的函数调用次数不相同.经过实验发现,在指定不同规模的情况下,NPB 的 8 个应用程序产生的日志条数差别不大(一部分应用的日志条数是不随计算规模变化的,例如 IS 等,另一部分应用的日志条数随计算规模而增长,但变化也不算很大,如 LU),而运行时间则随计算规模的增大而增长.

(2) 规格化与合并测试的目的是检查经过规格化与合并之后的日志是否能够保持日志原貌,本文算法能否在较短时间内完成规格化与合并操作.实验结果见表 2,本文方法能够成功地规格化与合并 8 个应用的日志,合并后日志条数都与表 1 中调用函数次数最多的进程相一致.合并时间中最长的 LU 为 375.84 秒,相比 LU 本身的运行时间要小得多.

Table 1 The counting number of called communications function

表1 通信函数调用次数

应用	BT	CG	EP	FT	IS	LU	MG	SP
次数(最多)	17 111	41 954	5	47	38	324 355	10 043	26 891
次数(最少)	17 111	41 954	5	47	36	162 189	9 329	26 891

Table 2 Results of regularization and merging experiment

表 2 规格化与合并实验结果

应用	BT	CG	EP	FT	IS	LU	MG	SP
合并后日志条数	17 111	41 954	5	47	38	324 355	10 043	26 891
合并时间/秒	20.22	49.68	0.02	0.12	0.08	375.84	12.57	35.47

(3) 日志收缩模块测试目的是检查收缩循环后的日志是否在逻辑上与原始日志一致,以及收缩后对计算时间的估算是否与展开后相符.时间效率测试主要检查本文算法能否在较短时间完成日志收缩,并可与文献[9]中的两种算法进行对比.NPB 程序属于科学计算类应用,每个程序基本上都由比较完整的循环结构组成,且覆盖范围最大的一个循环(包括嵌套)往往能够覆盖通信记录的绝大部分条目,称这个循环为主循环结构.合并后的日志的收缩长度以及经过实验得到的每个应用的主循环结构的长度和循环次数见表 3,其中 EP,FT,IS 这 3 个进程的通信记录过少,结果省略.对于主循环结构,由于将全部循环展开后形式过于复杂,这里只列出最外层循环的长度和循环次数.

Table 3 Results of Compression experiment

表 3 各应用的收缩情况

应用	收缩后长度	主循环结构	原始长度	压缩比(%)
BT	44	(15)×200	17 111	99.743
CG	26	(12)×75	41 954	99.938
LU	38	(7)×249	324 355	99.998
MG	302	(134)×20	10 043	96.993
SP	43	(15)×400	26 891	98.840

文献[9]中的两种算法为"自顶向下"算法和"自底向上"算法,根据本文第 3.3 节的理论分析,在收缩长度方面,本文算法与自顶向下算法相当,而比自底向上算法优秀.在时间性能方面,本文算法的时间复杂度为

© 中国科学院软件研究所 http://www.jos.org.cn

$O(n\log n)$,而这两种算法均为 $O(n^2)$.因此,本文算法在理论上的收缩长度和时间性能方面都比这两种算法优秀.在实际运行中,比自顶向下算法在时间性能上优秀,比自底向上算法在收缩长度上优秀,在时间性能上接近.表 4 和表 5 为实验的对比结果.

Table 4 Contraction length comparison of these three algorithms

表 4 3 种算法的收缩长度对比

应用	日志长度	本文算法	自顶向下算法	自底向上算法
BT	17 111	44	44	44
CG	41 954	26	26	26
LU	324 355	38	–	47
MG	10 043	302	302	302
SP	26 891	43	43	43

Table 5 Run time comparison of these three algorithms

表 5 3 种算法的运行时间对比

应用	日志长度	本文算法/秒	自顶向下算法/秒	自底向上算法/秒
BT	17 111	4.05	276.46	6.96
CG	41 954	6.81	1 498.27	7.59
LU	324 355	26.96	–	41.87
MG	10 043	2.76	104.84	9.05
SP	26 891	5.61	649.12	10.21

可见,在收缩长度方面,在对 LU 的日志收缩循环时,本文算法优于自底向上算法,对于其他应用程序的日志,3 种算法表现相同;在运行长度方面本文算法比自顶向下算法优秀得多,与自底向上算法相比也略有优势.表中没有自顶向下算法对于LU日志的结果,这是由于该算法在该项测试中消耗时间太长,远超过 10^5 秒.由于自顶向下算法时间复杂度是 $O(n^2)$,随着日志长度的增长时间消耗将快速增加,显然是难以使用在真实的时间预测之中的.综上,本文算法优于这两种已知算法.

5.2 基于判例的时间预估实验

在本节实验中,使用判例程序按照不同的缩放比例在原始环境下重新运行,用得到的"预估时间"与实际时间进行对比.由于 EP,FT,IS 等 3 个应用程序的通信过少,实验意义不大,因此以下实验均对另外 5 个应用程序的 C 类计算规模进行测试.在同构机群上运行原始程序获得判例程序,随后将判例程序缩小 10 倍,重新在第一组机群上运行,得到的预估时间与原始程序的实际运行时间对比见表 6.

Table 6 Results of time prediction in homogeneous clusters

表 6 同构环境下时间预估结果

应用	原始程序运行时间(s)	判例程序运行时间(s)	预估时间(s)	估计偏差(%)
BT	1 094.65	113.04	1 130.4	1.04
CG	384.81	37.93	379.3	−1.43
LU	877.27	85.16	851.6	−2.92
MG	786.34	77.57	775.7	−1.35
SP	1 219.16	118.56	1 185.6	−2.76

将判例程序分发到异构机群上运行,在得到基于判例程序的预测时间后,再将原始程序分发到第 2 组机群上运行,实际运行时间与预测时间的对比结果见表 7.

Table 7 Results of time prediction in heterogeneous clusters

表 7 异构环境下时间预估结果

应用	原始程序运行时间(s)	判例程序运行时间(s)	预估时间(s)	估计偏差(%)
BT	3 558.72	337.65	3 376.5	−5.12
CG	1 042.09	101.72	1 017.2	−2.39
LU	3 191.50	297.54	2 975.4	−6.77
MG	2 040.17	183.94	1 839.4	−9.84
SP	4 295.63	401.81	4 018.1	−6.46

© 中国科学院软件研究所 http://www.jos.org.cn

由表 6 和表 7 可以看出,对于同构机群,针对 NPB 的 5 个应用程序的时间预测误差不超过 3%;对于异构机群,误差不超过 10%.对异构机群的时间预计误差较大,但需要注意的是,在异构机群中原始程序运行时间本身也有一定波动,因此稍大的预测误差也是可以接受的.

6 结束语

本文深入研究了基于判例程序的并行程序运行时间预测问题的各个环节,设计了一种对计算任务在不同计算资源之下的运行时间预估的有效方法,能够避免传统的基于历史和基于理论分析的时间预测的许多局限,为分布式计算的任务级资源调度提供基础.1) 设计了一种获取并行程序运行时所有通信日志的方法,通过利用 MPI 库的 PMPI 接口,在 MPI 库源码中插入函数封套.2) 通过研究集合通信与一对一通信的特点,设计了一种规整化并行程序通信日志的方法,在规整化的基础上设计了一种合并日志的算法.3) 将判例技术中最核心也最困难的循环收缩识别问题转化为字符串的循环子串收缩问题,提出了一种基于后缀数组的算法,在理论和实际的时间性能方面都优于已有的最优算法.4) 重构出能够反映原始程序运行时间特征的,可按比例缩放的判例程序,解决了计算时间与通信时间的可按比例缩放模拟,设计了自动构建可执行判例程序的方法.

实验表明本文设计的方法能够比较准确地预估计算任务的运行时间,对于同构机群误差不超过 3%,对于异构机群误差不超过 10%.

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是哈尔滨工业大学计算机科学与技术学院方滨兴院士、张宏莉教授领导的课题组的同学和老师表示感谢.

References:

[1] Foster I. The grid: A new infrastructure for 21st century Science. Physics Today, 2002,55(2):42−47.

[2] Zhang W, Fang B, Hu M, Liu X, Zhang H, Gao L. Multisite co-Allocation scheduling algorithms for parallel jobs in computing grid environments. Science in China Series F—Information Sciences, 2006,49(6):906−926.

[3] Gao X, Snavely A, Carter L. Path grammar guided trace compression and trace approximation. In: Proc. of the 15th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-15). 2006. 57−68.

[4] Cardwell N, Savage S, Anderson T. Modeling TCP latency. In: Proc. of the IEEE INFOCOM 2000. 2000. 1742−1751.

[5] Sodhi S, Subhlok J. Skeleton based performance prediction on shared networks. In: Proc. of the 4th IEEE Symp. on Cluster Computing and the Grid (CCGrid 2004). Washington: IEEE Computer Press, 2004. 723−730.

[6] Cole M. Algorithmic Skeletons: Structured Management of Parallel Computations. Pitman/MIT Press, 1989. 1−42.

[7] Dikaiakos M, Rogers A, Steiglitz K. Fast: A functional algorithm simulation testbed. In: Proc. of the Int'l Conf. on Parallel and Distributed Systems. 1993.

[8] Dinda P, O'Hallaron D. An evaluation of linear models for host load prediction. In: Proc. of the 8th IEEE Int'l Symp. on High Performance Distributed Computing. 1999.

[9] Xu Q, Subhlok J. Efficient discovery of loop nests in communication traces of parallel programs. Technical Report, UH-CS-08-08, University of Houston, 2008.

[10] Xu Q, Subhlok J. Construction and evaluation of coordinated performance skeletons. In: Proc. of the 15th High Performance Computing (HiPC). 2008.

[11] Xu Q. Automatic construction of coordinated performance skeletons [Ph.D. Thesis]. University of Houston, 2007.

[12] Lu C, Reed DA. Compact application signatures for parallel and distributed scientific codes. In: Proc. of the Supercomputing 2002. 2002.

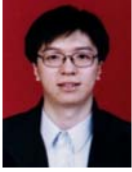
[13] Sherwood T, Perelman E, Hamerly G, Calder B. Automatically haracterizing large scale program behavior. In: Proc. of the10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X). 2002.

[14] MPI. http://www.mcs.anl.gov/research/projects/mpi

[15] Kärkkäinen J, Sanders P, Burkhardt S. Linear work suffix array construction. Journal of the ACM, 2006,53(6):918−936.

© 中国科学院软件研究所 http://www.jos.org.cn

[16] Bender MA, Farach-Colton M. The LCA problem revisited. In: Proc. of the Latin American Theoretical Informatics. 2000. 88−94.
[17] Fischer J, Heun V. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In: Proc. of the Int'l Symp. on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. LNCS 4614, Springer-Verlag, 2007. 459−470.



张伟哲(1976－),男,黑龙江哈尔滨人,博士,副教授,主要研究领域为网络计算,网络安全.

张元竞(1985－),男,硕士生,主要研究领域为网络计算,网络安全.

张宏莉(1973－),女,博士,教授,博士生导师,主要研究领域为网络计算,网络安全,拓扑发现.

www.jos.org.cn
www.jos.org.cn
© 中国科学院软件研究所 http://www.jos.org.cn