面向异构并行计算系统的流水线式压缩检查点

刘勇鹏[1],王 锋[1],卢 凯[1],刘勇燕[2]
(1.国防科学技术大学计算机学院,湖南长沙 410073; 2.中国科技部信息中心,北京 100862)

摘 要: 在大规模并行计算系统中,并行检查点触发大量结点同时保存计算状态,造成巨大文件存储空间开销,以及对通信和存储系统的巨大访问压力.数据压缩可以缩小检查点文件尺寸,从而降低存储空间开销以及对通信和存储系统的访问压力.但是,它也带来额外的压缩计算开销.本文针对异构并行计算系统,提出流水线式并行压缩检查点技术,采用一系列优化技术来降低压缩引入的计算延时,包括:流水线式双重写缓存队列、文件写操作的合并、GPU 加速的流水压缩算法和 GPU 资源的多进程调度,等等.本文介绍了该技术在天河一号系统中的实现,并对所实现的检查点系统进行综合评测.实验数据表明该方法在大规模异构并行计算系统中是可行、高效、实用的.

关键词: 异构并行体系结构; 检查点; 数据压缩; 软流水线; 图形处理器
中图分类号: TP338.4 **文献标识码:** A **文章编号:** 0372-2112 (2012)02-0223-07
电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2012.02.003

Pipelined Compressed Checkpointing for Heterogeneous Systems

LIU Yong-peng[1], WANG Feng[1], LU Kai[1], LIU Yong-yan[2]
(1. *College of Computer, National University of Defense Technology, Changsha, Hunan* 410073, *China*;
2. *Information Center, Ministry of Science and Technology of China, Beijing* 100862, *China*)

Abstract: Checkpointing is an effective technique to improve the reliability of large scale parallel computing systems. Data compression is a promising technique to reduce the size of data to be saved in the files in the storage subsystem and the amount of data to go through the communication subsystem. However, compression causes a huge amount of time overhead. The time overhead is the main technical barrier of its practical usability. In this paper, we propose a parallel compressed checkpointing technique to reduce the time overhead of compression in heterogenous architectures. It integrates a number of optimization techniques, which include transmitting checkpointing data between host and GPU in buffered pipelines, aggregating file write operations, employing a pipelined parallel compression algorithm, and delegating compression operations to GPU, etc. The paper reports an implementation of the technique in the TH-1 system and the evaluation experiments with the system. The experiment data show that the technique is efficient and practically useable.

Key words: heterogenous architecture; checkpoint; data compression; pipeline; graphic processing unit (GPU)

1 引言

对大规模并行计算系统而言,故障的发生不可避免,系统固有可靠性无法满足高性能应用的运行需求[1].基于检查点的回卷恢复是一种典型的软件容错技术.基本思想是,在计算到达检查点时,保存当前的计算状态,其后发生故障时,系统恢复到最近保存的状态并继续计算.在大规模并行计算系统中,并行检查点需要保存所有并行计算进程的运行状态,导致大量计算结点同时将计算状态数据保存到文件系统,不仅造成巨大的存储空间开销,而且形成密集文件访问,对通信和文件系统造成巨大压力.数据压缩是降低检查点文件访问量的重要手段.但是,数据压缩带来额外计算开销,影响系统的正常计算性能[7,8,10].

Socket 级异构在计算密度、能效、性价比等方面优势突出,是高性能计算的一个重要趋势,在最新 Top500 前 10 名中有 4 台为 Socket 级异构系统[4].在该类异构中,协处理器提供了丰富的计算能力,为优化压缩时间提供了新的契机.

收稿日期:2010-09-20;修回日期:2011-07-27
基金项目:国家 863 高技术研究发展计划重大项目(No.2009AA01A128);高效能服务器和存储技术国家重点实验室开放基金(No.2009HSSA04);国家自然科学基金(No.60603061)

2 压缩检查点时间开销的分析

典型的 Socket 级异构并行计算系统如图 1 所示,计算结点包含 CPU 和协处理器(GPU)两种异构的计算单元.计算阵列通过互连网络与存储阵列相连,访问全局文件系统.

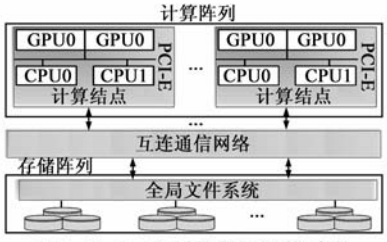


图1 Socket级异构并行计算系统

大规模并行计算系统中,当大量结点并行创建局部检查点时,形成密集文件操作,造成对互连网络和存储阵列的巨大访问压力.降低检查点文件的尺寸,不仅可以节省存储空间,而且可缓解对网络和文件系统的访问压力,是提高并行检查点技术可用性的重要途径.

本地检查点操作分为数据采集和数据保存两部分.数据采集时间 T_0 包括搜集进程状态、构造检查点数据等操作所需的时间.Socket 级异构系统中,各进程的文件吞吐率 b 基本相同.假设当进程数 p 达到 k 时,单位时间内并行发出的文件访问量等于文件系统的访问带宽 B_f,则有 $k \times b = B_f$.创建并行检查点时,如果单位时间的文件访问总量小于带宽,系统带宽不是瓶颈,文件访问时间由进程自身的检查点数据量 S 和文件访问速率(B_f/k)决定,如式(1(a)).我们称 k 为并行检查点技术的适用规模.当单位时间的文件访问总量大于等于带宽时,文件访问时间与进程数相关,受制于系统文件访问带宽,有式(1(b)).

$$T_{uc}=\begin{cases}T_0(S)+\dfrac{k\times S}{B_f}, & p<k \quad (a)\\ T_0(S)+\dfrac{p\times S}{B_f}, & p\geqslant k \quad (b)\end{cases} \tag{1}$$

引入数据压缩后,数据保存由两部分组成:数据压缩和保存压缩后数据.如果这两部分顺序执行,检查点保存时间为数据压缩时间 T_z 和保存压缩后数据时间 T_f 之和.设 δ 为数据压缩率,则压缩检查点所需时间 T_c 如式(2)所示.

$$T_c=\begin{cases}T_0(S)+T_z(S)+\delta\times\dfrac{k\times S}{B_f}, & p<\dfrac{k}{\delta} \quad (a)\\ T_0(S)+T_z(S)+\delta\times\dfrac{p\times S}{B_f}, & p\geqslant\dfrac{k}{\delta} \quad (b)\end{cases} \tag{2}$$

由式(2)可见,压缩引入额外的计算时间,但由于压缩后需要保存的数据量减少,文件访问时间相应地降低.T_{uc} 和 T_c 的对比关系如图 2 所示.由图 2 可知,压缩变相扩大了文件系统带宽,将适用规模从 k 扩大到 $k'=k/\delta$,还将检查点时间随系统规模的增长率从 S/B_f 降低为 $\delta\times S/B_f$.但是,如果顺序执行数据的压缩和保存,只有当并行进程数 p 达到一定规模时(即图 2 中的 P_0),压缩带来的时间优越性才能体现.对于文件系统带宽较大的高性能计算系统而言,P_0 的值往往很大,压缩带来的时间优越性难以实现.

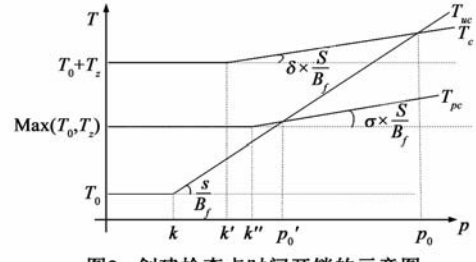


图2 创建检查点时间开销的示意图

针对上述问题,本文利用 Socket 级异构系统中的协处理器,并行处理检查点数据的压缩和保存.基本思想是将检查点数据分成 N 段,采用流水线技术,使多段数据之间的采集、压缩和保存三种操作并行执行.

假设每段数据的大小是 D,则检查点数据被分成 $N=S/D$ 段.设 $T_0(d_i)$,$T_Z(d_i)$ 和 $T_f(d_i)$ 分别为搜集、压缩和存储第 i 段状态数据的时间.在大规模并行计算系统中,检查点数据量 S 通常很大,因此,N 足够多,流水线填充和排空的时间相对整个流水线执行过程可忽略不计,流水线式压缩检查点的时间 $T_{pc}\approx\mathrm{Max}\{T_0,T_z,T_f\}$ 其中:

$$T_0=\sum_{i=1}^{N}T_0(d_i),\ T_Z=\sum_{i=1}^{N}T_Z(d_i),\ T_f=\sum_{i=1}^{N}T_f(\delta d_i).$$

当进程数 p 在适用规模内时(即 $p\leqslant k''=T\times B_f/\delta\times S$ 时,其中 $T=\mathrm{Max}(T_0,T_Z)$,文件访问不是瓶颈,检查点时间由压缩时间和数据采集时间的最大值来决定;当进程数 p 大于 k'' 时,文件访问时间开始增长,文件系统成为瓶颈,检查点时间等于文件保存的时间,从而有式(3).

$$T_{pc}=\begin{cases}\mathrm{Max}\{T_0(S),T_z(S)\}, & p<k'' \quad (a)\\ \delta\times\dfrac{p\times S}{B_f}, & p\geqslant k'' \quad (b)\end{cases} \tag{3}$$

从图 2 可以看出,流水线并行降低了压缩检查点的时间开销.而且,因为在大规模并行计算系统中,通常 $T_Z=\mathrm{Max}(T_0,T_Z)$ 且 $S/T_Z>b$,流水线并行进一步将压缩检查点技术的适用规模从 $k'=B_f/\delta\times b$ 扩大为 $k''=T\times B_f/\delta\times S$.

3 流水线式压缩检查点技术

本文针对 Socket 级异构并行系统,提出流水线式压

缩检查点的实现技术 PCCR(Pipelined Compressed Checkpoint/Restart).

PCCR 数据采集模块在操作系统内核采集目标进程的执行状态,并将采集到的检查点数据写入压缩缓存队列.为减少 CPU、GPU、文件系统之间频繁交互的通讯开销,提高文件系统的访问效率,PCCR 写缓存时还进行写合并,即将多个尺寸较小的数据块的写操作合并为一个较大的数据块的写操作.

PCCR 在用户库层实现检查点并行协议、基于 GPU 的数据压缩、文件操作等,支持缓存区尺寸、缓存队列长度、压缩窗口大小、最大匹配长度等参数的动态设置.

PCCR 采用同步并行检查点协议,进程同步完成后,各进程创建本地检查点.本地检查点创建完成后,所有进程再次同步,继续运行.

启动本地检查点操作之前,PCCR 为每个目标进程派生两个用户级进程,分别实现该目标进程的检查点数据的压缩与保存.

3.1　流水线式缓存队列

为实现数据采集、基于 GPU 的压缩以及文件操作之间的并行,本文为每个目标进程设计了两个环形缓存队列:压缩缓存队列和文件缓存队列,如图 3 所示.压缩缓存队列中的缓存区大小相等.队列头 head 指向内核层写输出的当前缓存区,队列尾 tail 指向当前待压缩缓存区,每个缓存区有 3 个状态:Empty(初始状态,没有有效数据)、Busy(包含部分待压缩数据)、Ready(数据采集完毕,可作为压缩的输入).

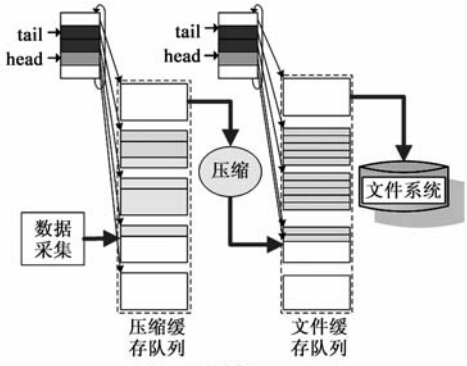


图3　双缓存环形队列

检查点过程中,PCCR 写缓存区前进行写合并判断,查看 head 指向的缓存区是否有足够的缓存空间.如果空间足够,则将数据写入当前缓冲区,标识状态为 Busy;如果空间不足,则将当前缓存区写满后,标识其状态为压缩 Ready,head 指向下一个缓冲区,并尝试将剩余内容写入下一个缓冲区.PCCR 内核将数据写入缓存区后,其一次"写文件"操作即结束,继续采集进程的其它状态.

压缩进程检测到压缩缓存队列 tail 指向的缓存区状态为 Ready 后,以缓存区为单位进行压缩.压缩完成后,缓存区状态设为 Empty,tail 指向下一个缓存区.

文件缓存队列是压缩进程的输出,文件操作进程的输入,文件操作进程将文件缓存队列中的数据写入文件系统.

通过压缩缓存队列和文件缓存队列的双重缓存,以缓存区为操作单位的压缩检查点过程分为三个阶段:数据采集、GPU 加速的压缩和写文件,如图 4 所示,彼此之间以流水线方式并行.

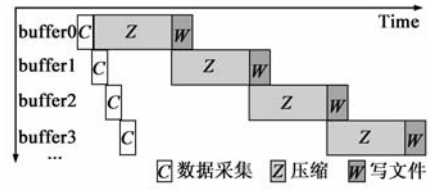


图4　压缩检查点软流水线

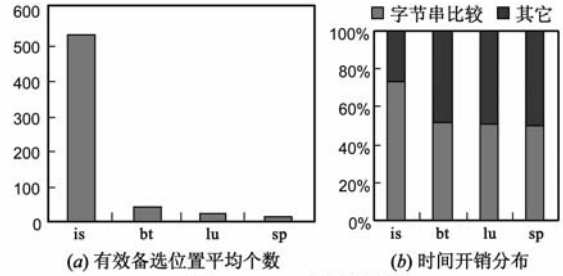
3.2　写合并

对文件系统而言,一次写大块数据要比把数据分成多块执行多次写操作的效率更高[15].对于压缩算法而言,数据块越大,压缩率通常也越高.但是,Ouyang 等人的研究表明[15],超过 60% 的检查点写文件操作的数据量小于 4KB,数据量偏小.针对这一问题,PCCR 在缓存队列的实现自然地完成了写操作的合并,将缓存区块的大小设置成较大的值,即可将较小尺寸的数据块合并为较大尺寸的数据块.由于双缓存队列结构,PCCR 写操作的合并可以在两个层次上完成,即内核写压缩缓存时和压缩进程写文件缓存时.

3.3　GPU 加速的流水压缩

Deflate[19]是目前广泛使用的一个压缩算法,它在 LZ77[13]压缩结果上进一步使用 Huffman 编码.PCCR 采用 Deflate 算法对检查点数据进行压缩,并利用 GPU 对该算法进行并行化加速.

LZ77 通过字节串匹配发现重复数据段来实现数据压缩.然而,如图 5(*a*)所示,我们将 Deflate 用于 NPB[18]的检查点数据的压缩实验表明,有效备选匹配位置的平均个数远远小于匹配窗口的大小(32768).用 Hash 表来管理窗口中的备选位置,可以剔除无效的窗口位置,



(*a*) 有效备选位置平均个数　　(*b*) 时间开销分布

图5　Deflate算法分析

是提高匹配搜索速率的有效途径之一[17,20].

但是,如图 5(*b*)所示,即使采用 Hash 表来管理窗口,LZ77 字节串比较仍占整个压缩时间的 50%以上.因此,降低字节串比较的时间开销是加速 Deflate 压缩算法的关键.窗口各个位置的字节串比较不存在数据依赖,可以利用 GPU 的 SIMD 并行计算能力进行并行加速.

PCCR 改进后的 LZ77 字节串比较过程分为输入、执行和输出三个阶段.输入将压缩缓存区的数据拷贝到 GPU 内存,执行在 GPU 上实现数据段并行匹配,输出将 GPU 的执行结果拷贝到主机内存.对一个缓存区的压缩需要多次执行字节串比较.为提高运行效率,我们采取如下技术.

(*a*)输入、输出和执行的流水并行化

为降低输入/输出传输延迟对 GPU 运行效率的影响,我们采用字节串比较流水线,实现两个待压缩缓存区之间的流水并行,如图 6 所示.

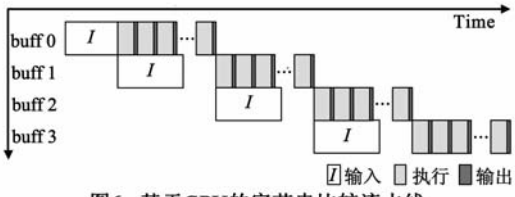


图6 基于GPU的字节串比较流水线

由于 GPU 上只能运行一个 Kernel,一次执行(即一次字节串并行比较)的输出只有 3 个字节(2 个字节表示匹配段在窗口中的偏移,1 个字节表示匹配长度),输出延迟极短,输出与其它操作的流水带来的优化不足以抵消由此引入的流水线控制开销.输入、输出和执行的并行化只体现在对当前缓存区的处理和计算结果的输出与下一个缓存区的输入之间的流水并行.为此,我们设计了当前缓存和前瞻缓存,分别用于暂存当前压缩缓存区和下一个压缩缓存区.两个缓存分别维护自己的 Empty、Busy、Ready 三种状态.

(*b*)CPU 进程分组及 GPU 的分时共享

每个 CPU 核可以运行一个并行进程,而 GPU 同一时刻只能执行一个 Kernel,无法同时为多个进程提供压缩服务.为此,我们将 CPU 进程分组,每组对应一个 GPU 服务调度进程,管理双输入缓存,实现 GPU 在组内进程之间的分时共享,如图 7 所示.

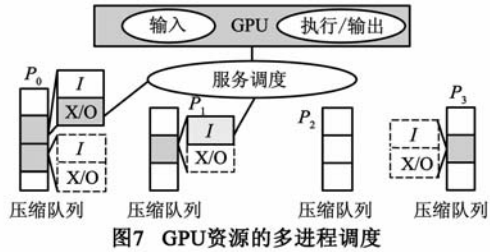


图7 GPU资源的多进程调度

(*c*) 两级服务调度算法

通过压缩流水线,一个 GPU 可以同时提供一个输入服务和一个执行/输出服务.如果前瞻缓存 Empty,即可提供输入服务,输入启动后前瞻缓存进入 Busy 状态,输入完成后,前瞻缓存切换为 Ready 状态.当前缓存 Ready 且 GPU 没有执行/输出服务未完成,即可提供执行/输出服务.当前缓存的数据全部执行/输出完成后,当前缓存设为 Empty,并切换当前缓存和前瞻缓存.

为充分流水,每个进程最多可发出两个缓存区的输入请求,两个请求分为两级,第一级优先级高于第二级.第一个请求被批准后,第二个请求升为第一级.为了 GPU 在进程之间的均衡共享,服务调度进程对同一级的输入请求采用轮转策略.由于流水线采用双输入缓存,最多只有一个压缩进程的输入就绪,可以发出执行/输出请求.

压缩进程监测压缩缓存队列 tail 状态,发现其 Ready 后,向服务调度进程请求一级输入服务.为填满流水线,同时还会监测 tail 的下一个缓存区,如果下一个缓存区 Ready,则二级请求下一个输入服务.输入申请被批准后,压缩进程将压缩缓存区拷贝到前瞻缓存.输入完成后,申请执行/输出服务,服务申请批准后,开始压缩缓存区.

PCCR 基于了 zlib[17] 设计了全新的 GPU 加速的 LZ77 字节串比较,但 LZ77 Hash 表和 Huffman 编码仍延用 zlib 的实现.

4 实验测评

天河一号是我国首台千万亿次异构并行计算系统.每个计算结点包括 2 个 Intel Xeon 4 核 CPU,32GB 内存和 1 个位于 PCIE 2.0 插槽的 ATI Radeon HD4870X2 GPU.该 GPU 包含 2 个独立的 RV770 芯片,每个芯片有 1GB GPU 内存,640 个基本运算单元.计算阵列通过两级 QDR InfiniBand 互连通信子系统与 I/O 子系统互连.I/O 子系统上构建 Lustre 全局文件系统.

在天河一号系统中,我们成功实现了 PCCR,基于 BLCR-0.8.2[2] 实现 Linux 内核层的进程检查点数据采集,基于 MVAPICH2-1.5[14] 实现 MPI 并行计算环境和同步检查点并行协议,基于 ATI Stream OpenCL SDK 2.1[11] 和 zlib-1.2.5 实现基于 GPU 的数据压缩算法.本节报告对天河一号系统中实现的 PCCR 进行的综合评测.

4.1 实验设置与评测用例

为确保实验中每个进程的检查点数据量保持不变,以便精确度量创建检查点计算时间随系统规模的变化,我们以 32 个计算结点作为一个计算组,每组运行相同的测试用例,同时对多个组进行创建检查点的操

作.

测试用例选择并行计算领域的标准测试集 NPB 3.3[18],NPROCS 设为 256,CLASS 设为 D,每个 CPU 核运行一个并行进程.

4.2　实验过程与主要结果

首先,我们选择不同的缓存区大小进行测试.为方便结果对比,本文以 1KB 缓存区配置的实验结果为基准,各实验结果除以该基准,得到一般化检查点时间.实验结果表明,压缩检查点时间随缓存区大小成 U 型变化趋势,如图 8 所示.其中,IS 测试用例在缓存区大小为 4KB 时,检查点用时最短,而其它 3 个测试用例,都是缓存区大小为 16KB 时用时最短.

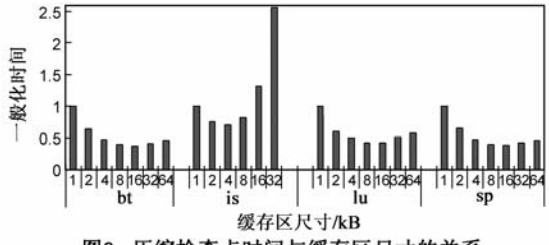


图8　压缩检查点时间与缓存区尺寸的关系

实验数据还表明,无论缓存区大小如何设置,PCCR 都在一定程度上降低了压缩检查点的时间开销.图 9 为缓存区大小取 16KB 时的优化效果,检查点时间以各测试用例的串行压缩为基准一般化.从图中可见,相比串行压缩,PCCR 时间至少降低 32.4%(在 SP 测试数据下),最多可降低 65.5%(在 IS 测试数据下).

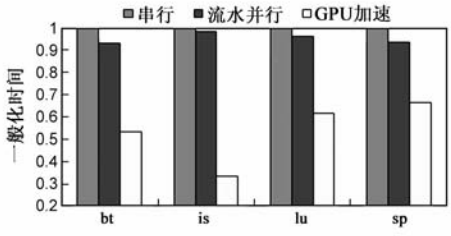


图9　压缩检查点时间

实验数据表明,在系统规模较小时(不超过 128),压缩是串行压缩检查点的主要时间开销.缓存区大小设为 16KB 时,压缩时间占串行压缩检查点时间最少比例为 90.4%(BT),而最高则占 96.3%(IS).通过 PCCR 三段流水线,检查点时间分别降低为串行压缩时的 93.1%和 98.5%,相当于最长流水段(即压缩)的 1.03 和 1.02,流水化并行效果明显.

实验数据还揭示了检查点时间是随系统规模的变化规律,以一个计算组无压缩检查点为基准一般化,结果如图 10(*a*)~(*d*)所示.

从图 10 可见,检查点时间随进程数的变化趋势与第二节的分析基本一致.如果不进行任何优化,压缩检查点的可扩展性很差,在 1024 结点(32 个计算组)规模内,串行压缩检查点时间大于无压缩检查点时间,压缩带来的数据保存时间的减少,不足以抵消压缩计算引入的时间开销.流水并行及 GPU 加速有效地降低了压缩检查点的时间开销.当规模达到一定程度时,比如在 BT 测试数据下当结点数大于 512 时,PCCR 不仅使检查点空间开销降低 8.2%,而且检查点时间比无压缩检查点短,规模为 512 时,PCCR 检查点时间比无压缩检查点降低了 17.7%,检查点时间随系统规模的增长趋势也比无压缩检查点降低了约 7.2%,增强了检查点系统的可扩展性.可见,对大规模系统而言,PCCR 在空间、时间、可扩展性方面都比传统的无压缩检查点有一定程度的优化,提高了检查点的实用性.

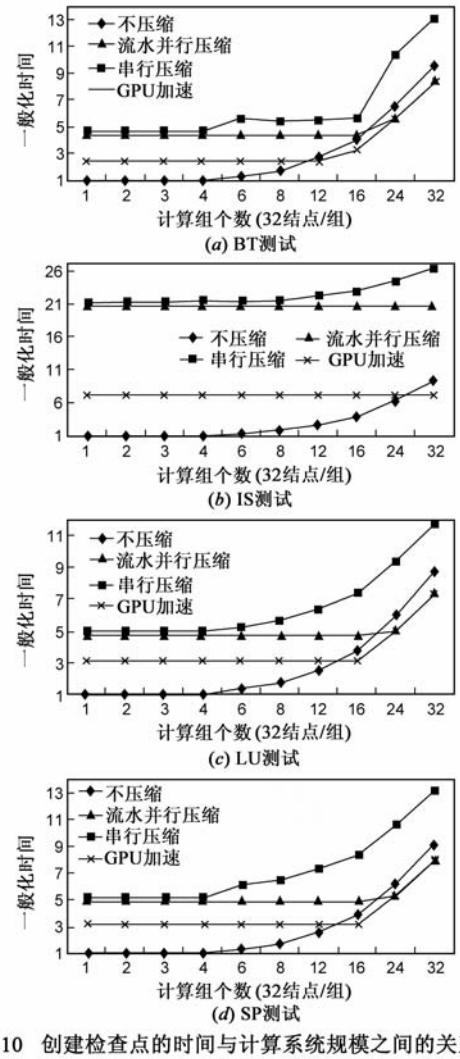


图10　创建检查点的时间与计算系统规模之间的关系

5　相关工作

数据存储技术是检查点研究的一个重要领域,包括状态保存策略、数据存储策略和存储数据管理等多个研究课题.

状态保存策略选择计算状态的一个子集作为检查

点保存的数据.主要途径有三:应用级检查点由应用自主确定需要保存的数据内容;系统级检查点保存应用的完整状态[2];编译器或用户辅助的定制检查点[3]则在编译器或用户的指导下选择需要保存的内容.BLCR[2]是一个应用广泛的系统级检查点实现,被 MVAPICH2、LAM/MPI 等许多 MPI 实现采用.本文基于 BLCR 保存应用的完整状态.实际运行中,通常需要为一个应用多次创建检查点.与每次检查点独立确定状态保存策略不同,增量式检查点利用两次检查点之间的数据相似性来确定需要保存的内容,后一次检查点只保存上次检查点之后发生变化的应用状态,从而降低检查点保存的数据量[9].

数据存储策略选择合适的时机将检查点数据保存到存储介质中.为确保数据保存的可靠性,可将采集到的数据立即写入文件系统.无盘检查点[5]将检查点数据保存在内存,提高了检查点存储的速率.多级检查点[6]利用多级存储结构,借鉴 Cache 思想,将检查点数据保存在不同的存储介质中,并采用一定的策略保证不同介质中的数据一致性.写缓存则将检查点数据缓存在内存,再按一定的写回策略存入文件系统.裴丹等人[22]对进程活跃文件内容采用写缓存方式.Ouyong 等人[15,16]利用写合并和写缓存来提高检查点性能,建立 CPU 与文件系统之间的数据缓存,由一个文件操作进程负责将结点内所有进程保存在缓存区内的数据写入磁盘.本文也采用写缓存策略.与他们不同的是,本文采用 CPU、GPU、文件系统三者之间的流水线式双重缓存队列,每个进程派生的文件操作进程将自己的检查点数据写入文件系统.另外,本文还研究了全局文件系统对检查点扩展性的影响.

数据存储管理技术按照一定的数据存储格式保存检查点数据.压缩式检查点将数据压缩后保存,旨在降低检查点数据的尺寸[8,7].Plank 等人[10]将增量与压缩相结合,进一步降低检查点的数据大小.对大规模并行计算系统而言,研究者虽然认识到压缩可以降低文件存储的空间开销,进而降低检查点操作对通讯和存储系统的访问压力.但是,压缩引入的时间开销一直困扰着压缩式检查点技术的实际应用[7,8,10].本文通过流水线式并行和对压缩算法的 GPU 加速,有效降低了压缩检查点的时间开销.

无损压缩算法包括短语式压缩和编码式压缩两大类.LZ77[13]和 Huffman 编码是这两类压缩算法的典型代表,Deflate[19]算法是 LZ77 和 Huffman 编码的融合,被 zlib[17]、gzip、zip 等压缩工具采用.本文采用 Deflate 算法压缩检查点数据.与孙圣[20]等采用专用硬件来实现加速不同,本文利用通用 GPU 加速压缩过程.目前利用 GPU 实现压缩的研究主要集中在多媒体处理等有损压缩领域[12].Wu 等人[21]利用 GPU 加速 LZ77,但他们只将数据分块,每个 GPU 线程压缩一块,实现压缩的并行,没有对 GPU 与主机之间的数据传输进行优化.与他们的并行方式不同,本文对 LZ77 中的字节串比较进行细粒度 SIMD 并行化,实现多进程之间的 GPU 资源调度和 GPU 输入与执行/输出的流水并行,充分利用了 GPU 强大的计算能力,使数据压缩的计算时间明显降低.

6 结束语

本文提出 GPU 加速的流水线式并行压缩检查点,有效降低了压缩检查点的时间开销,增强了压缩检查点在大规模异构并行计算系统中的实用性.

在作业规模、文件系统负载较小时,压缩检查点的时间开销大于无压缩检查点.根据文件系统负载情况,动态选择检查点方式,是我们后续工作的一个重点研究方向.另外,PCCR 相关参数的自适应选择,也有待进一步完善.

参考文献

[1] G Gibson, et al. Failure tolerance in petascale computers[J]. CTWatch Quarterly. 2007, 3(4): 4 – 10.

[2] Lawrence Berkeley National Laboratory. Berkeley lab checkpoint-restart (BLCR)[OL]. https://ftg.lbl.gov/CheckpointRestart/. 2010-07-11.

[3] 刘勇鹏,等.用户指导的多层混合检查点技术及性能优化[J].计算机应用研究.2008,25(7):2097 – 2099.
Liu Yongpeng, et al. User-defined hybrid checkpointing and optimization[J]. Application Research of Computers, 2008, 25(7): 2097 – 2099. (in Chinese)

[4] Top500. The 37th Top500 List[OL]. http://www.top500.org. 2011-07-10.

[5] James S Plank, et al. Diskless checkpointing[J]. IEEE Transaction on Parallel and Distributed Systems. 1998, 9(10): 972 – 986.

[6] N H Vaidya. A case for two-level distributed recovery schemes [A]. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems[C]. New York: ACM Press, 1995. 65 – 73.

[7] J Ansel, et al. DMTCP: transparent checkpointing for cluster computations and desktop[A]. 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)[C]. Washington: IEEE Computer Society, 2009. 1 – 12.

[8] J S Plank, et al. Ickp: A consistent checkpointer for multicumputers[J]. IEEE Parallel Distributed Technologies. 1994, 2(2): 62 – 67.

[9] S Agarwal, et al. Adaptive incremental checkpointing for massively parallel systems[A]. International Conference on Supercomputing (ICS'04)[C]. New York: ACM Press, 2004. 277 – 286.

[10] J S Plank, et al. Compressed Differences: An Algorithm for Fast Incremental Checkpointing[R]. Tennessee: University of Tennessee at Knoxville, 1995.

[11] Advanced Micro Devices (AMD). ATI Stream SDK v2.1 [OL]. http://developer. amd. com/gpu/ATIStreamSDK/Pages/default.aspx. 2010-07-11.

[12] Serpil Tokdemir. Digital Compression on GPU[D]. Georgia: Georgia State University, 2006.

[13] J. Ziv, et al. A universal algorithm for sequential data compression[J]. IEEE Transactions on Information Theory, 1977, 23(3):337 – 343.

[14] Network-Based Computing Laboratory (NBCL). MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE [OL]. http://mvapich.cse.ohio-state.edu/. 2010-07-11.

[15] X Ouyang, et al. Accelerating checkpoint operation by node-level write aggregation on multicore systems[A]. 2009 International Conference on Parallel Processing (ICPP'09)[C]. Washington: IEEE Computer Society, 2009. 34 – 41.

[16] X Ouyang, et al. Fast checkpointing by write aggregation with dynamic buffer and interleaving on multicore architecture[A]. 16th International Conference on High Performance Computing (HiPC'09)[C]. Kochi: IEEE Computer Society, 2009. 99 – 108.

[17] J Gailly, et al. 'zlib' version 1.2.5[OL]. http://www.zlib.net/, 2010-04-19.

[18] NAS Parallel Benchmark Team. NAS Parallel Benchmarks Version 3.3 (NPB3.3)[OL]. http://www.nas.nasa.gov/Software/NPB. 2007-08-30.

[19] RFC1951. DEFLATE Compressed Data Format Specification version 1.3[S].

[20] 孙圣.基于 FPGA 的 Deflate 算法核心模块设计[J].软件导刊,2010,9(5):63 – 64.
Sun Sheng. Core model design of deflate based on FPGA[J]. Software Guide, 2010, 9(5):63 – 64. (in Chinese)

[21] L Wu, M Storus, D Cross. CUDA Compression Project[R]. Stanford: Stanford University, 2009.

[22] 裴丹,等.WOB:一种新的文件检查点设置策略[J].电子学报,2000,28(5):9 – 12.
Pei Dan, et al. WOB: A novel approch to checkpoint active files[J]. Acta Electronica Sinica, 2000, 28(5): 9 – 12. (in Chinese)

作者简介

刘勇鹏　男,1977 年生于山西,硕士,助理研究员.主要研究方向为高性能计算、功耗管理、容错.
E-mail: liuyp@nudt.edu.cn

王　锋　男,硕士,助理研究员.主要研究方向为高性能计算、编译.