软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(6):1361−1375 [doi: 10.3724/SP.J.1001.2013.04325]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

静态分析面向异构系统的应用级 Checkpoint 设置问题*

贾 佳[1,2], 杨学军[1], 马亚青[3]

[1](国防科学技术大学 计算机学院 并行与分布处理国家重点实验室,湖南 长沙 410073)

[2](北京系统工程研究所,北京 100101)

[3](中国北方车辆研究所,北京 100072)

通讯作者: 贾佳, E-mail: morpheux@163.com

摘 要: 应用级 checkpointing 是一种在大规模科学计算领域中备受关注的容错技术,该技术由用户程序员选择在适当的地方保存关键数据,从而降低了容错开销.选择合适的 checkpointing 位置、减小全局 checkpoint 保存数据量是优化应用级 checkpointing 技术的关键问题.对于近年来推出的带有通用 GPU 的异构系统上的应用级 checkpointing 技术,也同样面临上述问题.针对异构系统体系结构和程序特征,对面向异构系统的应用级 checkpointing 技术的检查点设置进行了静态分析,提出两套不同机制的检查点设置方法:同步及异步检查点设置方法,并分别就 checkpointing 优化设置问题对其进行数学建模和求解.最后,通过实验验证并评估了所提出的两种方法的性能.

关键词: 应用级 checkpointing;异构系统;通用 GPU;同步检查点设置;异步检查点设置

中图法分类号: TP306 **文献标识码**: A

中文引用格式: 贾佳,杨学军,马亚青.静态分析面向异构系统的应用级 Checkpoint 设置问题.软件学报,2013,24(6):1361−1375. http://www.jos.org.cn/1000-9825/4325.htm

英文引用格式: Jia J, Yang XJ, Ma YQ. Static analysis for the placement of application-level Checkpoints on heterogeneous system. Ruan Jian Xue Bao/Journal of Software, 2013,24(6):1361−1375 (in Chinese). http://www.jos.org.cn/1000-9825/4325.htm

Static Analysis for the Placement of Application-Level Checkpoints on Heterogeneous System

JIA Jia[1,2], YANG Xue-Jun[1], MA Ya-Qing[3]

[1](National Laboratory for Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha 410073, China)

[2](Beijing Institute of System Engineering, Beijing 100101, China)

[3](China North Vehicle Research Institution, Beijing 100072, China)

Corresponding author: JIA Jia, E-mail: morpheux@163.com

Abstract: Application-Level checkpointing is a widely concerned technique used in large-scale scientific computing fields, and programmers to choose the appropriate place to save crucial data: henceforth, the fault-tolerant overhead can be reduced. There are two key issues in adopting this technique: find the proper place and reduce the scale of global checkpoints saving datum. The same problem is encountered when emerging heterogeneous systems with general purpose computation on GPUs. Towards architecture of heterogeneous system and characterization of application, this paper performs static analysis for the checkpointing configurations and placements, and two novelty approaches are proposed: 'synchronous checkpoint placement' and the 'asynchronous checkpoint placement'. The placement problem of checkpoints can be mathematically modeled and solved. Finally, their performances are evaluated via conducting experiments.

Key words: application-level checkpointing; heterogeneous system; general purpose computation on GPU; synchronous checkpoint placement; asynchronous checkpoint placement

* 基金项目: 国家自然科学基金(60921062, 61003087)

收稿时间: 2011-08-19; 修改时间: 2012-01-15; 定稿时间: 2012-08-20

大规模科学计算一直是推动高性能计算机迅速发展的主要动力.今天的科学家更加依赖于高性能计算机处理空前庞大的数据集实现空前复杂的模拟仿真.当前,随着 GPGPU(general purpose computation on graphic processing units)[1]性能的不断提高,利用 CPU 和 GPU 构建的异构系统已经成为高性能计算机[2,3]领域的研究热点,我国首台千万亿次计算机天河(Tianhe-1A)[4]就是一个典型案例.而在最新公布的 Top500 前 5 位的高性能计算机中,采用异构系统的计算机除了天河外还有另一台我国的"星云(Nebulae)"和惠普的 TSUBAME 2.0,由此可见今后的发展趋势和研究热度.

直到目前为止,提高计算机性能的主要手段仍是增加处理器数目,因此,高性能计算机的规模迅速扩大.当前,世界上最快的计算机——日本的"京"由 68 544 个 SPARC64 VIIIfx 处理器组成,每个处理器均内置 8 个内核,总内核数量高达 548 352 个[4].然而,系统规模的急剧扩大,导致系统的平均故障时间间隔(MTBF)大幅降低.相关数据显示,当一台大规模的 IBM/LLNL ASCI White 系统在持续运行过程中,其 MTBF 最多不超过 40 个小时[5],而一台拥有 8 000 个节点的 Google 集群系统,其单节点失效率达到了 2%~3%/年[6].那么换一种说法,其 MTBF 也仅仅为 36 小时[7].另一方面,科学计算程序往往需要连续运行几天甚至几个月,例如,Blue Gene/L 上的蛋白质折叠(protein-folding)程序需要运行好几个月[8].所以,这些大规模科学计算程序必须具备容忍硬件故障的能力.当前,随着硬件功能的不断丰富和软件开发环境的逐渐成熟,GPU 开始被应用于通用计算领域,协助 CPU 加速程序的运行.为了追求高性能,GPU 往往包含成百上千个核心运算单元,高密度的计算资源,使其在性能远高于 CPU,而这一特性也使得 GPU 的失效率远大于 CPU.由于商用 GPGPU 容错能力较弱,所以,由 CPU 和 GPU 构建的大规模异构并行系统的可靠性问题更为尖锐,尚缺乏实用的容错手段.针对这种情况,我们将同构系统中最为广泛应用于大规模科学计算领域的容错技术[9,10]应用级 checkpointing 技术[11,12]应用于异构系统.

在执行大规模计算这种长时间运行的异构程序时,需要进行多次 checkpointing.为了减小总的容错开销,我们希望在程序中合理地设置各 checkpoint 的位置,使得程序执行期间发生的全局 checkpointing 开销尽可能小.这就是所谓的多 checkpoint 的优化设置问题.目前,对于面向异构系统的应用级 checkpointing 技术尚无一套完整和高效的设置方法.本文针对这一问题展开研究,根据异构系统的特点提出同步和异步检查点设置方案,并分别就 checkpoint 设置问题对这两种方案进行讨论;通过分析异构系统程序特征及各部件 MTBF 等相关约束条件,以合理设置 checkpointing 次数;同时,为选取最小 checkpoint 数据保存点,对 checkpointing 时间间隔进行偏移量设置;最后,根据上述分析结果建立异构系统应用级 checkpointing 最优设置的数学模型并求解.

本文第 1 节分析异构系统体系结构及程序特征,并提出 checkpointing 优化设置所需解决的问题.第 2 节针对文中提出的同步及异步检查点设置方案对 checkpointing 优化设置问题进行建模和求解.第 3 节实验验证及评估本文提出方法的性能.第 4 节介绍相关工作.第 5 节总结全文并进行展望.

1 背景及问题

传统的同构系统下,checkpointing 的主要开销是将 checkpoint 数据写入稳定存储器的时间,系统的失效率和 I/O 传输速度是其主要约束条件;而在异构系统下,由于具有与同构系统不同的特性,对于影响 checkpointing 开销的约束条件相比同构系统也更多更为复杂.为了便于我们对问题精确的描述,需要对异构系统体系结构及程序特征进行研究.

1.1 异构系统体系结构

一个典型的异构系统体系结构如图 1 所示,它由一个 CPU 和一个 GPU 组成,它们之间通过 PCI-E 总线相连,各自的存储系统可以通过 DMA 操作进行大块的数据传输.当一个异构程序在异构系统上运行时, CPU 和 GPU 之间是一种主从关系,CPU 负责主控程序的执行,以 kernel 调用的方式启动 GPU 进行异步运算,并通过 Read 和 Write 操作与 GPU 进行数据传输;GPU 响应 CPU 发起的 kernel 调用,并通过 SIMT 的方式启动大量的轻量级线程完成 kernel 计算.

由此可见,CPU 与 GPU 之间存储空间是分离的,且 GPU 显存空间明显无法满足大规模计算的需求,因此在 GPU 进行 checkpointing 时,其 checkpoint 数据最终必须都要通过数据的拷贝传输转存到 CPU 端.这就使得异构

系统在 checkpointing 时不仅需要分析系统主存的 I/O 带宽对于其开销的影响,还需要将 CPU-GPU 之间传输带宽的因素考虑进去,这些都是对异构系统 checkpointing 的整体开销产生巨大影响的约束条件.

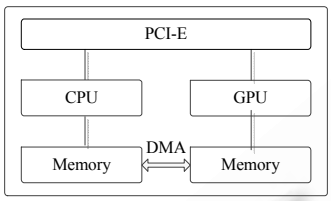

Fig.1 Architecture of heterogeneous system

图 1 异构系统体系结构

1.2 异构系统程序特征

目前比较热门的几类异构系统编程模型[13]主要包括 AMD 公司的 Brook+[14]、Nvidia 公司的 CUDA[15]和业界制定的统一编程标准 OpenCL[16].本文中,我们在分析异构程序特征时,将各公司引入的面向其特定体系结构的细节特征隐去,只抽取异构程序执行方式的相关特征.本文选取 CUDA 为研究对象,因此文中都统一采用 CUDA 专用术语[17]来描述.

在异构系统中,程序分为 CPU 运行代码和 GPU 运行代码两部分,即流级代码和核心级代码两部分[18]:流级代码运行于 CPU 上,负责组织数据,配置并调用核心(kernel)函数的执行;而核心级代码则全部运行于 GPU 上,负责具体的计算.而通过图 2 的程序代码可以看出,CUDA 程序在通过 CPU 启动 GPU 计算时,主程序首先在 CPU 端初始化矩阵 *ac*,*bc* 和 *cc*,并事先将它们的全部元素在 GPU 上分配好地址空间,再在 kernel 进行有效计算前将这些元素全部拷贝到 GPU 中,最后由 CPU 调用执行计算.由此可见,CPU 启动 GPU 计算时的参数传递是通过数据拷贝的传值方式来实现的,即在 kernel 出入口处都存在数据传输,如图 3 所示.

需要特别注意的是,由异构系统的编程模型可以看出,CPU 和 GPU 之间的数据传输只发生在 kernel 计算开始前和结束后的入口和出口处,如图 3 所示,在 GPU 进行 kernel 计算的过程中是不能中途进行 CPU-GPU 通信的.因此在程序某一计算段过程中,CPU 必须要等待 GPU 上 kernel 计算完成时才能进行通信.有别于传统的同构系统,在异构系统中使用应用级 checkpointing 技术时,由于异构程序这一特征的约束,在计算过程中无法插入 checkpoint,只有在 kernel 外,即 kernel 之间才能执行 checkpointing.

```
//Main on CPU
  float *ac, *bc, *cc;
  // allocate space on the GPU
  cudaMalloc((void**)&ac,sizeof(float)*n*n);
  ...
  // transfer array to GPU
  cudaMemcpy2D(ac,sizeof(float)*n,a,...,
       cudaMemcpyHostToDevice);
  ...
  // execute the kernel
  matMultCUDA<< blocks*n,threads>>
    (ac,n,bc,n,cc,n,n);
  // transfer stream to CPU
  cudaMemcpy2D(c,sizeof(float)*ldc,cc,...,
      cudaMemcpyDeviceToHost);
   ...
```

Data Transfer

```
//Kernel on GPU

__global__ static void
  matMultCUDA(const float*a,
  size_t lda,const float*b,size_t ldb,
  float*c,size_t  ldc, int n)
  { // get the thread index
     const int tid=threadIdx.x;
     const int bid=blockIdx.x;
     const int idx=bid*blockDim.x+tid;
     ...
       // perform operations
        c[row*ldc+column]=...;
  }
```

Fig.2 Matrix multiply on CUDA

图 2 CUDA 上的矩阵乘算法

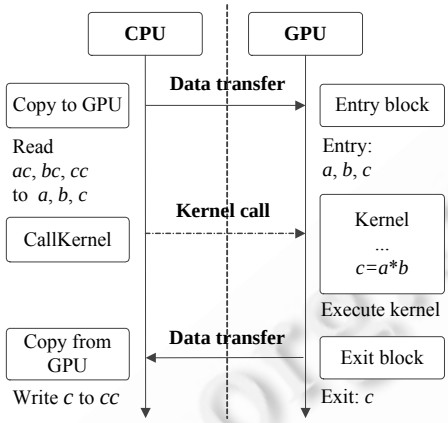

Fig.3 Program process on heterogeneous system

图 3 异构系统程序流程

此外,由图 3 可以看出,在计算矩阵乘时,CPU 端的矩阵 c 直到 GPU 端计算结束返回结果后才会更新,在计算过程中保持不变,且 CPU 处于空闲状态.由于 CPU 与 GPU 的存储空间是分离的,这也使得 CPU 与 GPU 在计算过程中互不干扰,且可各自独立运算,从而使得异构系统程序不仅具备同步执行机制,也可以实现异步执行机制.这也为我们对异构系统应用级 checkpointing 的优化设置问题分析提供了新的思路和方法.

1.3 问题提出

根据前文对于异构系统体系结构和程序特征的分析,可以看出,异构系统 checkpointing 的开销主要分为两部分:将 checkpoint 数据写入稳定存储器的时间以及 CPU 与 GPU 之间的数据传输时间.将 checkpoint 数据进行传输以及写入存储器的时间与 checkpoint 数据量、CPU-GPU 传输带宽及系统 I/O 速度有关.不同系统的 I/O 速度各异,不同应用程序产生的 checkpoint 数据量也千差万别,不方便直接比较 checkpoint 开销.但是大规模科学计算应用产生的 checkpoint 数据量通常远大于单位时间内系统 I/O 能够传输的数据量,这时,写 checkpoint 数据的时间会近似地正比于 checkpoint 数据量,同时,checkpoint 的数量也直接影响到总体 checkpoint 开销.为了使解决方案超越具体系统与应用程序的差异,我们近似地认为并行程序执行期间的总 checkpoint 开销正比于总的 checkpoint 数据量以及 checkpointing 的次数.这样,多 checkpoint 优化设置的目标就变成了设置最优的 checkpointing 位置、合理地控制和减少 checkpoint 的数量,使得总 checkpoint 数据量的最小化.

多 checkpoint 优化设置问题中的 checkpoint 位置有两种基本的产生途径,并由此衍生出两个优化问题:

(1) 程序员在异构系统程序中任意插入了 N 个 checkpoint 指令.然而,保证程序正确的执行完成可能并不需要全部 N 次 checkpoint.为了减小不必要的容错开销,我们要从这 N 次 checkpoint 中选取 $M(M\leqslant N)$ 个,使得总的 checkpointing 次数最小;

(2) 程序员需要在一个异构系统程序中插入 checkpoint 指令.我们通过分析异构系统程序特点找出若干个合理的 checkpoint 位置,并使得在这些位置处 checkpoint 的数据总量最小.

下面我们针对这两个问题对异构系统 checkpoint 设置问题进行数学建模并求解.

2 全局 checkpointing 开销最小化

2.1 基于同步的checkpointing

2.1.1 同步检查点设置

在异构系统程序运行过程中,CPU 与 GPU 具备同步与异步两种执行行为,基于这两种执行机制的特性,我们为检查点的设置方法分别进行了规划和设计,并基于检查点优化设置问题对其进行分析.

同步机制的检查点设置方法,就是在程序执行的某一点处对 CPU 和 GPU 同步进行 checkpoint 的插入设置,并在 checkpointing 时暂停在 CPU 与 GPU 端计算过程,并在两端检查点数据全部保存结束后继续计算.图 4 给出了同步的检查点设置以及执行的基本流程.由于 kernel 执行过程中无法中断,且只在 kernel 之间进行通信,由此可见,GPU 端对于检查点设置的要求更高.因此,我们以 kernel 为粒度进行设置,将检查点统一设置在 kernel 执行前的位置,CPU 端检查点同步设置在对 kernel 进行启动执行前.当 GPU 端检查点数据传输到 CPU 端且全部检查点数据保存结束后,CPU 才开始调用 kernel 继续进行计算.

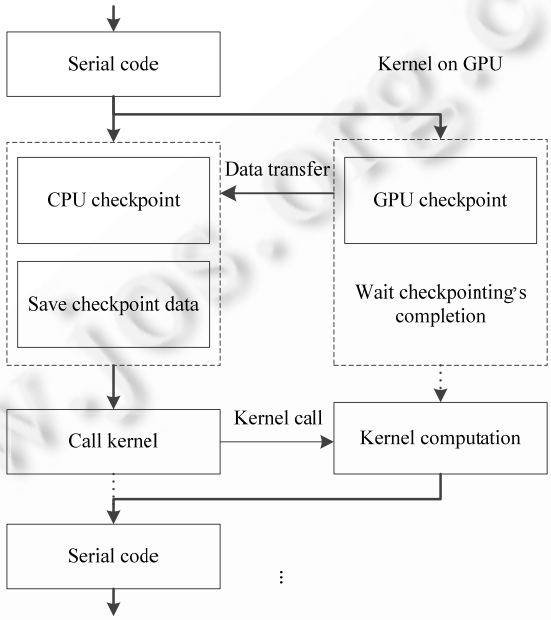

Fig.4 Process of synchronous checkpointing

图 4 同步执行 checkpointing 的流程

检查点设置时机的依据之一就是系统的 MTBF.在异构系统下,完整系统主要包含 CPU 与 GPU 两个部件,且所有计算也都在这两个部件上完成;但是在同步机制的检查点设置方式下,CPU 与 GPU 是被看成一个整体来考虑的.因此,为了突出异构系统与同构系统的区别,同时为了便于分析与对 GPU 的重点关注,我们只考虑加入 GPU 的平均故障间隔时间后对系统的整体 MTBF 进行分析.那么,对于异构系统的整体 MTBF 就主要综合 CPU 和 GPU 两个部件的平均故障间隔时间来进行分析.为了便于讨论,我们对各个 MTBF 进行定义:

- $MTBF_{CG}$:异构系统总体平均故障间隔时间;
- $MTBF_{CPU}$:CPU 的平均故障间隔时间;
- $MTBF_{GPU}$:GPU 的平均故障间隔时间.

通过体系结构分析可以看出,异构系统中,CPU 和 GPU 是以串联方式进行构成整体系统的,则根据经典可靠性理论,如果系统中存在 N 个部件,则系统 MTBF 为[19]

$$MTBF=\frac{1}{\frac{1}{MTBF_1}+\frac{1}{MTBF_2}+\ldots+\frac{1}{MTBF_N}} \tag{2-1}$$

那么,异构系统的整体 $MTBF_{CG}$ 为

$$MTBF_{CG}=\frac{1}{\frac{1}{MTBF_{CPU}}+\frac{1}{MTBF_{GPU}}} \tag{2-2}$$

2.1.2 数学模型

在异构程序中,我们可以将每个 kernel 看做是循环中最小的一个基本块,checkpoint 只能插于基本块的分界处,每个 kernel 外插入一个 checkpoint,因此在 GPU 端,我们将 checkpoint 插在 kernel 的出口和入口处.本文统一将 checkpoint 插在 kernel 的入口处.如图 5 所示,GPU 端一次循环中最多有 N 个 kernel,那么其可插入的 checkpoint 也为 N 个.其中,虚线表示 checkpoint 指令执行及 checkpoint 数据的保存过程,实线代表程序的正常执行过程.

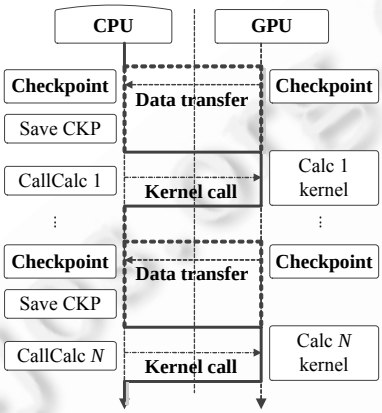

Fig.5 Synchronous checkpoint placement on CPU-GPU

图 5 同步设置 CPU-GPU 端 checkpoint

一般情况下,kernel 计算的时间开销都会小于系统 MTBF,即使出现一个规模非常大的 kernel 也可以通过程序员手动分割来改变其规模,因此这种情况不予考虑.由于异构程序的特点,我们规定将检查点插于 kernel 之间,那么针对一个需要插入多个检查点的大规模计算程序,当其存在 i 层循环,并且一个循环中存在 j 个 kernel 时,我们可以将循环展开,即存在 $N=i\times j$ 个 kernel.由于同步检查点的设置位置由 GPU 为标准来设置,并且已知程序循环次数并且固定循环中检查点的插入位置,因此我们可以直接将异构系统同步检查点设置问题转化为与问题(1)同样的问题来分析.即当 GPU 端的 kernel 数目为 N 个,那么程序员最多会在程序中 CPU 和 GPU 端同步插入 N 个 checkpoint 指令来对 CPU 端数据及每个 kernel 的状态数据都进行保存.但这是没有必要的,因此我们要从这 N 次 checkpoint 中选取 $M(M\leqslant N)$个,使得总的 checkpoint 数量最小,即总的 checkpointing 开销最小.

在这个问题中,每一个 checkpoint 指令要么保留,要么去掉.因此,这是一个类似于 0-1 整数规划的优化问题.

首先用一个长度为 N 的数组 $x[N]$表示每个同步设置的 checkpoint 指令的去留:若 $x[i]=0$,则表示第 i 个 checkpoint 指令要被去掉;若 $x[j]=1$,则表示第 j 个 checkpoint 指令会被保留.我们再用一个长度为 N 的数组 $C[N]$表示程序员选择的 N 个 checkpoint 位置的数据量和执行时间信息.$C[N]$的每一个元素都是一个 *C_struct* 结构体,*C_struct* 包括两个域:数据量大小 *size* 和程序执行到该位置处的执行时间 *time*,如图 6 所示.其中,程序执行到每个 checkpoint 位置的时间 $C[i].time$ 可由专门的 profiling 工具——NVIDIA's CUDA Visual Profiler[20]获得.

```
struct C_struct{
    double size
    double time
}C[N];
```

Fig.6 Structure of *C_Struct*

图 6 *C_Struct* 的数据结构

假设连续两次同步 checkpointing 的时间间隔为 T_C.$T_C<MTBF_{CG}$,那么这个问题可抽象为类似于 0-1 整数规

划的数学模型.其中,B 为 CPU 端存储 checkpoint 数据到硬盘的 I/O 带宽,B_2 为 CPU-GPU 通信带宽,其单位都为 GB/s.

$$\min\sum_{i=1}^{N}\left(\frac{C[i].size\times x[i]+G[i].size\times y[i]}{B}+\frac{G[i].size\times y[i]}{B_2}\right)$$

约束条件:

$$\begin{aligned}&(1)\quad \sum_{i=1}^{N}x[i]=M;\\&(2)\quad M\leqslant N;\\&(3)\quad \text{if }\sum_{i=a}^{b}x[i]=2,\ \text{then } C[a].time-C[b].time\leqslant T_C,i,a,b\in[1,N];\\&(4)\quad x[i]=0\text{ or }1,i=1\sim N.\end{aligned}\tag{2-3}$$

上述模型(2-3)中有 4 个约束条件:条件(1)和条件(2)表示总的同步 checkpoint 数量不超过 N,条件(3)要求选取的 M 个 checkpoint 中任意相邻两个的时间间隔必须小于或等于 T_C,条件(4)表示原有 N 个 checkpoint 位置中的每一个都有可能被保留或被删除.

2.2 基于异步的checkpointing

2.2.1 异步检查点设置

在传统的checkpointing思想下进行checkpointing时,CPU和GPU是同步执行的,需要考虑综合CPU及GPU两个部件后的系统整体 MTBF.而在串联系统上,系统整体 MTBF 必定小于任意单一部件的 MTBF,因此使得同步执行 checkpointing 的次数远比只为单一部件进行 checkpointing 的次数多,其容错开销也相对会大很多.另外,由前文分析得出的结论可以看出,GPU 在进行 kernel 计算时是不可中断的,那么即使推导出 checkpoint 的理论最优设置点,也需要根据当前 kernel 计算时间来具体判定,在分析上更为复杂.由于同步模式下需要同时保存 CPU 和 GPU 上的 checkpoint 数据,这就必然存在更多的存储时间开销和容错开销.而异步的应用级 checkpointing 则利用异构系统异步执行机制,分别为 CPU 和 GPU 进行 checkpointing,更有效地利用了 CPU 的空闲时间,最少化地为 CPU 和 GPU 设置 checkpoint 点,也使得应用级 checkpointing 技术在异构系统的应用更加灵活高效.图 7 给出了异构系统在一个计算段上为 CPU 和 GPU 异步设置检查点的位置,以及异步执行 checkpointing 的基本流程.

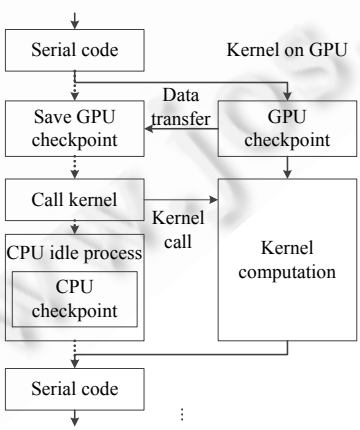

Fig.7 Process of asynchronous checkpointing

图 7 异步执行 checkpointing 的流程

在异步执行 checkpointing 时,我们分别考虑 CPU 和 GPU 两个部件的 MTBF,针对它们各自的 MTBF 以及 checkpoint 数据保存量等相应的约束条件,为 CPU 和 GPU 分别设置 checkpoint.由于异构系统中 GPU 负载了绝

大部分计算,且一般来说 $MTBF_{GPU} \leqslant MTBF_{CPU}$,那么可得出结论:如果分别为 CPU 和 GPU 进行 checkpointing,那么 GPU 端 checkpoint 数量必然多于 CPU 端,其容错开销也远大于 CPU 端,因此对 GPU 中检查点进行优化设置的需求也更大.而且在异步执行模式下,我们利用 CPU 的空闲时间对 CPU 端程序进行 checkpointing,其检查点保存的时间开销将被隐藏于 kernel 计算的时间开销之下,由于检查点保存的时间开销远小于 MTBF 且 GPU 端计算完全不受影响,使得在异步检查点设置下 CPU 端的 checkpointing 开销近乎可忽略不计,这也使得对 GPU 端检查点的最优化设置可以取得最优的容错性能.同时,依然对 CPU 端检查点进行优化设置,令 CPU 端 checkpoint 数据保存量最小,也使系统整体的全局 checkpointing 开销最小化.

2.2.2 数学模型

异步检查点设置与同步检查点设置的分析方法类似,只是需要多考虑一个部件,其每个部件的设置思路都与同步检查点设置问题相同.

在异步检查点设置时,我们同样已知程序循环次数并且固定循环中检查点的插入位置,因此直接将这一问题转化为前面提出的问题(1)同样的问题来分析.即当 CPU 端最多插入 N 个 checkpoint,而 GPU 端的 kernel 数目为 K 个,那么程序员最多会在 GPU 端插入 K 个 checkpoint 指令来对每个 kernel 都进行保存.但这是没有必要的,因此我们要从这 N 和 K 个 checkpoint 中选取 M 和 M'($M \leqslant N, M' \leqslant K$)个,使得总的 checkpoint 数据量最小,即总的 checkpointing 开销最小.图 8 给出了异步检查点设置后的程序执行流程,其中,GPU 端一次循环中最多有 K 个 kernel,那么其可插入的 checkpoint 也为 K 个,而 CPU 端 checkpoint 与 GPU 端 checkpoint 异步设置,我们对其任意插入 N 个 checkpoint.图中虚线表示 checkpoint 指令执行及 checkpoint 数据保存过程,实线代表程序正常执行过程.

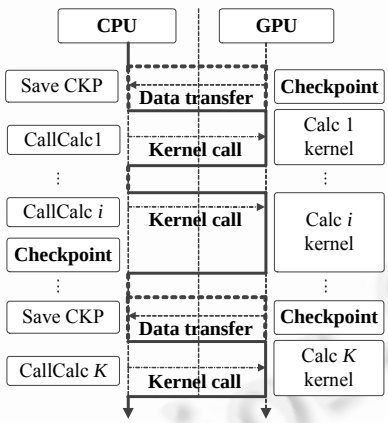

Fig.8 Asynchronous checkpoint placement on CPU-GPU

图 8 异步设置 CPU-GPU 端 checkpoint

在这一问题中,分别在 CPU 和 GPU 上的不同位置设置 checkpoint,每一个 checkpoint 指令要么保留,要么去掉,也同样是一个类似于 0-1 整数规划的优化问题.

与同步检查点设置模型类似,我们首先用一个长度为 N 的数组 $x[N]$表示每个 CPU 端 checkpoint 指令的去留:若 $x[i]=0$,则表示第 i 个 checkpoint 指令要被去掉;若 $x[k]=1$,则表示第 k 个 checkpoint 指令会被保留.同样,用一个长度为 K 的数组 $y[K]$表示每个 GPU 端 checkpoint 指令的去留:若 $y[j]=0$,则表示第 j 个 checkpoint 指令要被去掉;若 $x[l]=1$,则表示第 l 个 checkpoint 指令会被保留.我们再用一个长度为 N 的数组 $C[N]$表示程序员在 CPU 端选择的 N 个 checkpoint 位置的数据量和执行时间信息,用一个长度为 K 的数组 $G[K]$表示程序员在 GPU 端选择的 K 个 checkpoint 位置的数据量和执行时间信息.与前文相同,$C[N]$与 $G[K]$的每一个元素都是一个类似 *C_struct* 的结构体,都包括数据量大小 *size* 和程序执行到该位置处的执行时间 *time* 这两个域.其中,程序分别在 CPU 和 GPU 端执行到每个 checkpoint 位置的时间 $C[i].time$ 和 $G[j].time$ 也都可由 profiling 工具获得.

假设 CPU 端连续两次 checkpointing 的时间间隔为 T_C,GPU 端连续两次 checkpointing 的时间间隔为 T_G,$T_C<MTBF_{CPU}$,$T_G<MTBF_{GPU}$.那么,这个问题可抽象为类似于 0-1 整数规划的数学模型,其中,B 为 CPU 端存储 checkpoint 数据到硬盘的 I/O 带宽,B_2 为 CPU-GPU 通信带宽,其单位都为 GB/s.

$$\min\left(\sum_{i=1}^{N}\frac{C[i].size\times x[i]}{B}+\sum_{j=1}^{K}\left(\frac{G[j].size\times y[j]}{B}+\frac{G[j].size\times y[j]}{B_2}\right)\right).$$

约束条件:

$$(1)\quad \sum_{i=1}^{N}x[i]=M,\sum_{j=1}^{K}y[j]=M';$$

$$(2)\quad M\leqslant N,M'\leqslant K;$$

$$(3)\quad \text{if }\sum_{i=a}^{b}x[i]=2,\text{ then }C[a].time-C[b].time\leqslant T_C,\ i,a,b\in[1,N],\qquad (2\text{-}4)$$

$$\text{if }\sum_{j=a}^{b}y[j]=2,\text{ then }G[a].time-G[b].time\leqslant T_G,\ j,a,b\in[1,K];$$

$$(4)\quad x[i]=0\text{ or }1,i=1\sim N,y[j]=0\text{ or }1,j=1\sim K.$$

与之前的同步机制检查点设置模型相同,模型(2-4)中也有 4 个约束条件:条件(1)和条件(2)表示 CPU 端与 GPU 端总的 checkpoint 数量不超过 N 和 K;条件(3)要求在 CPU 端选取的 M 个 checkpoint 中任意相邻两个的时间间隔必须小于或等于 T_C,在 GPU 端选取的 M'个 checkpoint 中任意相邻两个的时间间隔必须小于或等于 T_G;条件(4)表示 CPU 端原有 N 个 checkpoint 位置中的每一个都有可能被保留或被删除,而 GPU 端原有 K 个 checkpoint 位置中的每一个都有可能被保留或被删除.

2.3 偏移量设置

在前文中,我们分别对同步及异步机制 checkpoint 的设置进行了讨论,并求解了 T_{CG},T_C 和 T_G 的最优解的数学模型.然而,对于全局最优的 checkpoint 设置的数学模型而言,在达到全局最优 checkpoint 时间间隔处设置的 checkpoint 却并不一定是最为合适的.对于个别 checkpoint 的设置位置而言,其所在 kernel 处需要保存的状态数据有可能是非常大的.

一个异构程序中有许多 kernel,对于不同的 kernel,所需保存的 checkpoint 数据量大小也是不同的,因此,在一定可容忍的范围内选取保存数据量最小的 checkpoint 位置,可以更好地减小全局 checkpointing 开销.

在异步检查点设置中,我们不仅需要针对问题(1)对 CPU 和 GPU 各自选取 checkpoint 进行分别设置,还需要在达到问题一约束条件的同时对问题(2)更进一步的讨论,再根据问题(1)选出的 CPU 及 GPU 两端检查点位置的一定范围内对其中可以进行检查点保存的位置进行分析,选取出 CPU 和 GPU 端各自存在的数据保存量最小的 checkpoint 的位置,这是一个偏移量的设置问题.需要对连续两次 checkpointing 之间的时间间隔进行偏移量设置,更进一步地对 checkpoint 数据保存量进行优化设置.

由于可以通过偏移量设置为 CPU 和 GPU 分别选取更小的 checkpoint 数据保存位置,因此偏移量设置的方法对于异步机制 checkpoint 设置的优化更为有效.在异构程序中,其主要计算集中于 GPU,且 CPU 端状态数据保存量的大小在很大程度上取决于当前对应的 kernel 所有的活跃变量数据量的大小,所以我们在对同步机制 checkpoint 进行偏移量设置时,着重针对 GPU 的 checkpoint 设置来分析,同样可以取得不错的优化效果.

图 9 中每一个竖条代表一个 kernel 具有的活跃变量的大小,横轴表示执行时间,纵轴表示活跃变量数据量的大小,深色竖条代表Δt 范围内的 kernel,灰色竖条表示Δt 范围外的 kernel.可以看到,在偏移量Δt 的范围内,总共有 6 个 kernel,其中,第 5 个 kernel 的活跃变量数据量明显低于其余 5 个,因此,我们将 checkpoint 设置在这一 kernel 之后,由此极大地减小了当前 checkpoint 的数据保存量.

偏移量的范围主要是依据程序员针对各个不同的应用的特点来进行判定并设置的.对于某些 kernel 与 kernel 之间活跃变量的数据量差异明显的应用而言,通过对全局最优 checkpoint 时间间隔进行偏移量设置,为 checkpoint 位置进行一定范围的调整,可以取得极大的优化效果.

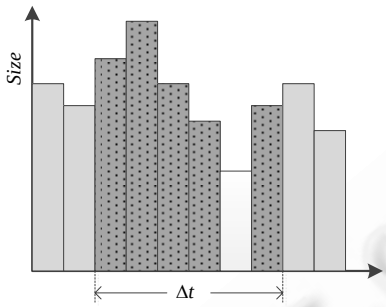

Fig.9 Checkpoint selection in Δt

图 9 偏移量Δt中的 checkpoint 位置选取

2.4 模型求解

模型(2-3)、模型(2-4)的第 3 个约束条件不能转化为 0-1 整数规划的标准约束条件,因此它们并不是 0-1 整数规划模型.但是,我们仍然可借助求解 0-1 整数规划问题的隐枚举法求解这些模型.由于模型相似,解法上可通用,这里我们只列出模型(2-4)的求解算法.

模型(2-4)的求解算法如图 10 所示.

算法. 含偏移量设置的模型(2-4)的求解算法.

输入:M,M',N,K,T_C,T_G和数组 $C[N],G[K]$;

输出:数组 $C[N],G[K]$和总 checkpoint 数据量的最小值.

1. 任选一个可行解 $x[N],y[K]$,计算与之对应的目标函数值 P,Q,增加一个过滤条件:

$$\sum_{i=1}^{N}\frac{C[i].size\times x[i]}{B}\leqslant P,\sum_{j=1}^{K}\left(\frac{G[j].size\times y[j]}{B}+\frac{G[j].size\times y[j]}{B_2}\right)\leqslant Q$$

2. **if** 第 1 枚举点对应的目标函数值 $P1,Q1$ 不满足新增的过滤条件 **then**
   转第 3 步;
   else //即满足过滤条件
   if 第 1 枚举点不满足模型(2-4)的任意一个约束条件 **then**
   转第 3 步;
   else //即全部都满足
   修改新增的过滤条件:$P=P1,Q=Q1$;
   转第 3 步;
   endif
   endif
3. **while** 还有枚举点未经考察
   if 下一个枚举点对应的目标函数值 $P2,Q2$ 不满足新增的过滤条件 **then**
   重复第 3 步,考察下一个枚举点;
   else //即满足过滤条件
   if 当前枚举点不满足模型(2-4)的任意一个约束条件 **then**
   重复第 3 步,考察下一个枚举点;
   else //即全部都满足
   修改新增的过滤条件:$P=P2,Q=Q2$;
   重复第 3 步,考察下一个枚举点;
   endif
   endif
   endwhile

Fig.10 Algorithm for solving model (2-4)

图 10 模型(2-4)的求解算法

3 实现与评估

3.1 实验方法

为了验证异构系统同步及异步 checkpointing 执行机制的可行性并评估优化效果,我们选取了 CUDA 平台的 Swim,Mgrid 等 6 种算法进行实验.通过代码修改,在 CUDA 上实现了应用级 checkpointing 技术及其同步和异步的执行机制.同时,基于前文提出的设置方法,根据加入偏移量设置的数学模型(2-3)、模型(2-4)的约束条件对检查点插入位置的设置进行了分析及优化处理.需要说明的是,由于本文所选取的实例规模都不是很大,为了达到更好的实验效果,我们通过修改代码增加了算法规模,并以此为依据设定及模拟相应的 CPU 的 $MTBF_{CPU}$ 和 GPU 的 $MTBF_{GPU}$;同时,根据公式(2-2)算出系统 $MTBF_{CG}$,以验证我们方法的优化效果.本实验中,我们设 $MTBF_{CPU}$=400000ms,$MTBF_{GPU}$=240000ms,那么根据计算得出系统 $MTBF_{CG}$=150000ms.基于工程经验,一般认为 *MTBF*/2 是两次 checkpointing 之间的最优时间间隔,本文实验也以此为标准对 checkpoint 进行设置.

作为对比,我们统计出同一算法在同步和异步检查点设置策略下所需设置的最优 checkpoint 数量,并对每种算法基于同步及异步机制执行一次的 checkpointing 的开销及全局 checkpointing 开销进行测试和对比,得出 6 组数据,并以此来对我们方法的优化效果进行评估.

此外,我们在实验中没有加入 restart 机制的开销评估.由于实验的目的是为了评估 checkpointing 优化设置的性能,且本文的方法与恢复机制没有关系,因此为了更为简单直观地验证和突显本文所提出的优化设置策略的性能,就只对全局 checkpoint 数据保存开销进行了对比和评估.

实验平台是由 i945+GTX295 搭建的 CUDA 服务器,四核 CPU 每核主频都是 2.66GHz,GPU GTX295 中有 30SM(Stream Multiprocessor),每个 SM 中有 8 个核,主存为 4GB,CPU 与 GPU 之间传输带宽为 4GB/s,在 LinuxSUSE 11.2 下进行代码修改及测试.

3.2 实验结果

本节对 CUDA 平台同步及异步机制的应用级 checkpointing 技术时间开销进行评估,并对这两种机制进行了对比.针对每个应用程序测试了一下 6 组数据:

- 同步检查点设置下 checkpoint 的数量;
- 异步检查点设置下 checkpoint 的数量;
- 同步设置的一次 checkpoint 的数据保存时间;
- 异步设置的一次 checkpoint 的数据保存时间;
- 无故障情况下带有同步执行的 checkpointing 的程序整体运行时间;
- 无故障情况下带有异步执行的 checkpointing 的程序整体运行时间.

表 1 给出了各个程序在同步与异步检查点设置情况下,分别插入 checkpoint 的数量.其中,*Sync_CKP* 是代表同步设置下 checkpoint 的数量,*Aysnc_CKP* 是异步设置下 checkpoint 的数量.可以看出,由于 GPU 的 *MTBF* 小于 CPU 的 *MTBF*,异步设置下,GPU 端 checkpoint 数量要多于 CPU 端;但无论 CPU 或 GPU 端 checkpoint 数量都要小于同步设置下的 checkpoint 数量.

Table 1 Number of checkpoints

表 1 检查点数量

	Sync_CKP	*Async_CKP*	
	CPU-GPU	CPU	GPU
Swim	14	5	9
Mgrid	56	21	35
FFT2D	11	4	7
BackProp	66	24	41
BFS	28	10	17
Gaussan	52	19	32

表 2 列出了各程序在带有两种执行机制的 checkpointing 的整体运行时间开销.*Sync_CKP* 是带有同步设置

应用级 checkpointing 的程序的总时间开销,*Async_CKP* 是带有异步设置应用级 checkpointing 的程序的总时间开销.可以看出,同步检查点设置下的程序总体时间开销全都要大于异步检查点设置下的程序执行时间,其中差距最大的 Mgrid 达到了 357 889ms,最小的 FFT2D 为 15 438ms.这是由于 FFT2D 的程序规模较小,且 checkpoint 数据保存量较小,使得每次 checkpointing 的时间开销较小造成的.

Table 2 Time overhead of checkpointing

表 2 检查点时间开销

	Sync_Ckp (ms)	*Async_Ckp* (ms)
Swim	1 261 289	1 148 858
Mgrid	4 756 304	4 398 415
FFT2D	881 425	865 986
BackProp	5 235 780	5 042 400
BFS	2 197 076	2 125 882
Gaussan	3 930 368	3 912 090

图 11 给出了在同步与异步两种设置情况下各个程序执行一次 checkpointing 的开销柱状图.可以更为直观地看到,单一异步设置的 checkpoint 在保存时的时间开销要远小于同步的 checkpoint 保存时间.BFS 算法在异步设置的单次 checkpointing 时间比同步设置的单次 checkpointing 时间开销减小了 57.3%,而减小最少的 FFT2D 也达到了 33.5%.可见,即使是单次检查点,在异步设置下的开销也要明显小于同步设置时的开销.

图 12 给出了同步与异步设置下不带原始程序执行时间的全局 checkpointing 的开销柱状图.可以更为直观地看到,对于同一个程序,其异步设置的全局 checkpointing 的时间开销要远小于同步设置的 checkpointing.其中,开销差距最为明显的 BFS 算法在异步设置时的全局 checkpointing 的时间比同步设置的全局 checkpointing 时间少 73.3%,而差距最小的 FFT2D 也达到了 58.4%.由此可见,异步设置的 checkpointing 相对于同步设置 checkpointing 在开销上有显著的优势.

通过实验可以看出,隐藏了 CPU 端 checkpoint 数据保存开销后的异步设置 checkpointing 相对于同步设置 checkpointing 在开销上的优势十分明显.从两种设置方法下单次 checkpoint 数据保存时间开销的对比就可以看出差距;同时,在同步设置比异步设置情况下的 checkpoint 数量也要多很多,这就必然导致了两者在全局 checkpointing 时间开销上的巨大差距.由此可以看出,异步设置的应用级 checkpointing 技术更加适用于异构系统,其对于异构系统容错技术的开销优化和性能提升也有十分巨大的作用.在 6 种算法中,FFT2D 相对其余 5 个算法在两种设置方法对比时的效果最小.这是由于其计算规模不大,且 checkpoint 的数据保存量较小的原因引起的.可见,在需要保存大量状态数据的大规模应用中,我们提出的异步检查点设置方法对于 checkpointing 性能的提升将有更为显著的体现.

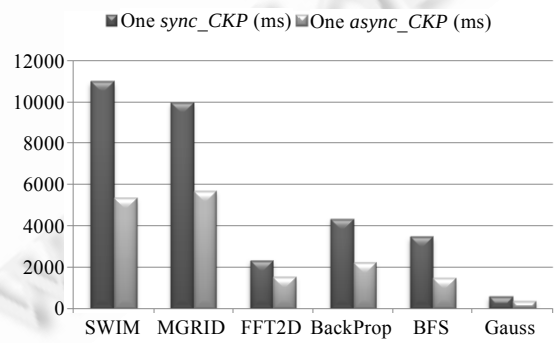

Fig.11 Time overhead of one *sync* & *async* checkpointing

图 11 一次同步与异步 checkpointing 时间开销

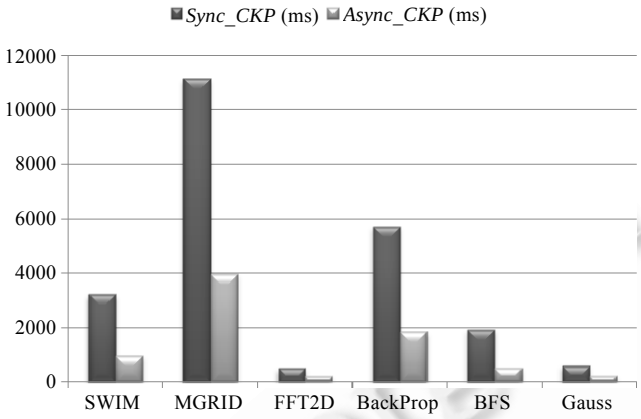

Fig.12 Time overhead of global *sync* & *async* checkpointing

图 12 同步与异步设置下全局检查点时间开销

4 相关工作

本文的研究是面向异构系统的应用级 checkpointing 技术的优化设置方法,而传统同构系统的应用级 checkpointing 优化设置方法研究已久.常用的优化 checkpointing 的标准是检查点的平均响应时间及其有效性.Krishna 等人就曾在 1984 年提出一种有效性的权衡公式——系统故障的临界收益累计和所有用户的平均损失的比[21].本文采用最少的 checkpoint 数据保存开销为优化标准,用它近似表示最小的 checkpointing 开销.

相关工作还有使用了 graph-theoretic 方法作为指导程序员进行 checkpoint 的插入[22],通过标识出最优的 checkpoint 位置,使得最大 checkpoint 时间、期望的 checkpoint 时间或期望运行时间都可以被最小化.还有些研究者也提出了类似的方法[23,24].然而,这些方法都是针对传统的同构系统开发的,并未出现一套比较有效地针对异构系统应用级 checkpointing 技术的设置方法.

随着 GPU 进入通用计算领域乃至高性能计算领域,面向 GPU 的容错技术逐渐被各大厂商和相关研究者关注,但是目前,异构系统最常用的容错技术还主要是硬件冗余和软件冗余技术.Sheaffer 和 NVIDIA 研究小组的 Luebke 等人在 2007 年设计了一种面向超级计算领域的 GPGPU 可靠性的硬件冗余和恢复机制[25],George 等人也在 2008 年 DSN 会议上提出了基于硬件双模冗余和同步数据流技术的异构系统的容错方案[26].在软件冗余方面,Dimitrov 等人提出了一种冗余执行 kernel 的方式来对 GPU 上发生的瞬时故障进行故障检测[27].在已有的异构系统容错技术中,对于应用级 checkpointing 技术的研究还很少,也尚未出现一套针对这一技术的优化设置方法.本文基于 CUDA 平台展开研究,将应用级 checkpointing 技术应用于异构系统,并对其优化设置问题进行静态分析,提出两套不同机制设置方法,有效地减小了容错开销,提高了容错性能.

5 总 结

当前,异构系统计算机逐渐进入高性能计算领域,面向异构的容错技术研究必将成为该领域的研究热点之一.但异构系统的容错技术由于异构系统体系结构及异构程序的一些特性及约束使得容错开销极为巨大,这也为如何降低面向异构系统的容错开销提供了更大的研究和优化空间;与此同时,这也导致面向异构系统的容错优化方法有别于传统同构系统中的优化方法,使得研究者可以发散思维,拓展并提出更有创新性和更有意义的容错方法.

本文针对异构系统的特性分析,提出了面向异构系统的同步及异步的应用级 checkpointing 的优化设置方法.仅就实现上来说,同步检查点设置方法更易于实现,对于程序员来说负担更小,因此同步检查点设置方法更适用于程序规模小、容错开销小的应用程序;而异步检查点设置方法则更适用于大规模应用程序.我们通过实验详细评估了这两套方法对 6 个典型应用程序的优化效果,结果表明,异步检查点设置方法相对于同步检查点

设置方法更适用于异构系统,该方法可以有效降低全局 checkpointing 的开销,更有效地提升了异构系统应用级 checkpointing 技术的性能.未来的工作将进一步研究面向异构系统的容错技术,并对新的容错机制及容错优化方法展开深入探讨.

References:

[1] Luebke D, Harris M, Krüger J, Purcell T, Govindaraju N, Buck I, Woolley C, Lefohn A. GPGPU: General purpose computation on graphics hardware. In: Proc. of the ACM SIGGRAPH 2004 Course Notes (SIGGRAPH 2004). New York: ACM Press, 2004. 33. [doi: 10.1145/1103900.1103933]

[2] Fan Z, Qiu F, Kaufman A, Yoakum-Stover S. GPU cluster for high performance computing. In: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing (SC 2004). Washington: IEEE Computer Society, 2004. 47. [doi: 10.1109/SC.2004.26]

[3] Dally WJ, Hanrahan P, Erez M, Knight TJ. Merrimac: Supercomputing with streams. In: Proc. of the Supercomputing Conf. (SC 2003). 2003. 35−42. [doi: 10.1109/SC.2003.10043]

[4] TOP500 supercomputing site. http://www.top500.org

[5] Read DA, Lu CD, Mendes CL. Reliability challenges in large systems. Future Generation Computers System, 2006,22(3):293−302. [doi: 10.1016/j.future.2004.11.015]

[6] Brown A, Patterson DA. Embracing failure: A case for recovery-oriented computing (ROC). In: Proc. of the High Performance Trans. on Processing Symp. 2001.

[7] Bosilca G, Bouteiller A, Cappello F, Djilali S, Fedak G, Germain C, Herault T, Lemarinier P, Lodygensky O, Magniette F, Neri V, Selikhov A. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In: Proc. of the 2002 ACM/IEEE Conf. on Supercomputing. Baltimore: IEEE Computer Society Press, 2002. [doi: 10.1109/SC.2002.10048]

[8] Bronevetsky G, Marques D, Pingali K, Stodghill P. Automated application-level checkpointing of mpi programs. In: Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP). 2003. 84−94. [doi: 10.1145/966049.781513]

[9] Elnozahy EN, Alvisi L, Wang YM, Johnson DB. A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys, 2002,34(3):375−408. [doi: 10.1145/568522.568525]

[10] Plank JS, Li K, Puening MA. Diskless checkpointing. IEEE Trans. on Parallel Distributed Systems, 1998,9(10):972−986. [doi: 10.1109/71.730527]

[11] Ramkumar B, Strumpen V. Portable checkpointing for heterogeneous architectures. In: Proc. of the 27th Int'l Symp. on Fault-Tolerant Computing (FTCS'97). Washington: IEEE Computer Society, 1997. 58. [doi: 10.1109/FTCS.1997.614078]

[12] Schulz M, Bronevetsky G, Fernandes R, Marques D, Pingali K, Stodghill P. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In: Proc. of the Supercomputing 2004. 2004. [doi: 10.1109/SC.2004.29]

[13] Kapasi UJ, Rixner S, Dally WJ, Khailany B, Ahn JH, Mattson P, Owens JD. Programmable stream processors. IEEE Computer, 2003,36(8):54−62. [doi: 10.1109/MC.2003.1220582]

[14] Advanced Micro Devices, Inc. AMD brook+. http://ati.amd.com/technology/streamcomputing/AMDBrookplus.pdf

[15] Kirk D. NVIDIA CUDA Software and GPU Parallel Computing Architecture. New York: ACM Press, 2007. 103−104. [doi: 10.1145/1296907.1296909]

[16] Open computing language. http://www.khronos.org/

[17] CUDA technical training volume I/II. Prepared and Provided by NVIDIA, 2008.

[18] NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 2.1, Beta, 2008.

[19] Zou FX, Zhang XP. Basic Technology for Fault-Diagnosis and Reliability in Computer Application Systems. Beijing: High Education Press, 1999 (in Chinese).

[20] Compute visual profiler 4.0 for NVIDIA CUDA user guide. DU-05162-001_v04. 2011.

[21] Krishna CM, Shin KG, Lee YH. Optimization criteria for checkpoint placement. Communications of the ACM, 1984,27(10):1008−1012. [doi: 10.1145/358274.358282]

[22] Chandy KM, Ramamoorthy CV. Rollback and recovery strategies for computer programs. IEEE Trans. on Computers, 1972,21(6): 546−556. [doi: 10.1109/TC.1972.5009007]

[23] Toueg S, Babaoğlu Ö. On the optimum checkpoint selection problem. SIAM Journal on Computing, 1984,13:630−649. [doi: 10.1137/0213039]

[24] Upadhyaya SJ, Saluja KK. An experimental study to determine task size for rollback recovery systems. IEEE Trans. on Computers, 1988,37(7):872−877. [doi: 10.1109/12.2235]

[25] Sheaffer J, Luebke D, Skadron K. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In: Proc. of the Graphics Hardware 2007. 2007.

[26] George N, Lach J, Gurumurthi S. Towards transient fault tolerance for heterogeneous computing platforms. In: Proc. of the 38th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN 2008). 2008.

[27] Dimitrov M, Mantor M, Zhou H. Understanding software approaches for GPGPU reliability. In: Proc. of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2), Vol.383. New York: ACM Press, 2009. 94−104. [doi: 10.1145/1513895.1513907]

附中文参考文献:

[19] 邹逢兴,张湘平.计算机应用系统的故障诊断与可靠性技术基础.北京:高等教育出版社,1999.

贾佳(1981－),男,北京人,博士生,主要研究领域为计算机体系结构,容错技术.
E-mail: morpheux@163.com

杨学军(1963－),男,博士,教授,博士生导师,主要研究领域为并行计算机体系结构与编译,容错并行算法,流处理器体系结构与编译.

马亚青(1975－),女,高级工程师,主要研究领域为控制工程.