并行复算：一种面向高性能计算的新的容错方法

王攀峰 杜云飞 富弘毅 杨学军 周海芳

（国防科技大学计算机学院并行与分布处理国家重点实验室 长沙 410073）

摘 要 Checkpointing是高性能计算领域最常用的容错技术。但是，当处理器数目变大时，这种技术的性能迅速恶化。提出一种在并行计算中容忍单进程故障的新方法：并行复算。这种方法的主要特征是利用冗余处理器的计算能力而不是冗余磁盘的存储能力实现低开销的容错。还提出这种方法的一个优化方法，将并行复算与checkpoint技术相结合，以进一步减小容错开销，并通过举例说明如何开发一个基于并行复算以及其优化方法的并行程序。最后通过实验对该方法进行评估。结果显示，当处理器数目变大时，并行复算的开销低于checkpointing，其优化方法能提供优于并行复算的性能。

关键词 高性能计算，容错，并行复算

Parallel Recomputing: A New Approach for Fault-tolerant High Performance Computing

WANG Pan-feng DU Yun-fei FU Hong-yi YANG Xue-jun ZHOU Hai-fang

(National Laboratory for Paralleling and Distributed Processing, College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract Checkpointing is the most commonly used scheme for tolerating faults in high-performance computing systems. But this scheme has its performance limitation when the number of processors becomes much larger. The paper proposed a new approach called parallel recomputing for tolerating a single process failure in parallel computing. The main feature of our approach is that it utilizes the computing power of the redundant processor instead of the storage capacity. The paper also presented an optimization of this approach which is a combination of parallel recomputing and checkpointing, and then illustrated how to incorporate parallel recomputing and its optimization into a parallel program. Experimental results demonstrate that the overhead of parallel recomputing is less than checkpointing when the number of processors becomes large, and its optimization can provide a better performance than parallel recomputing.

Keywords High-performance computing, Fault tolerance, Parallel recomputing

1 前言

科学计算一直是推动高性能计算机发展的主要动力。今天，科学家们越来越依赖高性能计算机处理空前庞大的数据集和实现空前复杂的模拟仿真。而直到目前为止，提高计算机性能的主要手段仍然是增加处理器数目，因此高性能计算机的规模迅速扩大。世界上最快的计算机——IBM Blue Gene/L更是拥有212,992个处理器。然而，系统规模的急剧扩大导致系统的平均无故障时间（MTBF）大幅降低。很多高性能计算机的MTBF只有几小时或者十几小时。像Blue Gene/L这样处理器数目超过10万的大规模系统，MTBF甚至会降到只有几十分钟甚至更短[1]。另一方面，很多科学计算程序往往需要连续运行几天甚至几个月，例如IBM Blue Gene上的蛋白质折叠程序需要运行好几个月[2]。很多高性能计算机的MTBF已经变得比运行与该系统上的科学计算应用的执行时间更短，因此这些科学计算应用必须具备容忍硬件故障的能力。

1.1 故障模型

为了解决上述问题，需要首先定义并行计算的故障模型。最常用的故障模型有两类：fail-stop模型和Byzantine模型[3]。在Fail-stop模型中，当某一个进程发生故障时，该进程停止运行，但不会引起系统中其他进程发生错误的状态变化。这种故障主要指并行计算中结点死的情况。在Byzantine模型中，当某一个进程发生故障时，故障进程会引起其他进程发生错误的状态变化，比如发送错误的数据等。这类故障很难被检测，本文主要讨论如何容忍fail-stop故障。

由于MPI是当前编写高性能应用的事实标准，本文集中讨论针对MPI程序中fail-stop类型的进程故障的容错方法。

1.2 现有的解决方案

Fail-stop类型的进程故障的解决方法主要分为2类：message-logging技术和checkpointing技术。checkpointing技术是在程序执行期间将计算状态周期性地保存到可靠存储

到稿日期：2008-04-29 本文受国家自然科学基金项目（60621003和60603081）资助。

王攀峰 博士生，主要研究方向为高性能计算和容错，E-mail：wpfeng@nudt.edu.cn；**杜云飞** 博士生，主要研究方向为高性能计算和容错；**富弘毅** 博士生，主要研究方向为高性能计算和容错；**杨学军** 教授，博士生导师，主要方向为高性能计算、系统结构等；**周海芳** 博士，副研究员，主要研究方向为高性能计算和图像处理等。

器上，存为一个 checkpoint 文件。如果某个进程失效，所有其他进程都必须立即终止执行，然后重新启动并通过读入最近保存的 checkpoint 文件将计算状态恢复到故障点之前的最近一个保存点处，最后计算从该保存点处继续执行[4-7]。message-logging技术要求每个进程保存它的计算状态和它所发出的每个消息的副本。如果任何进程失效，只有发生故障的进程上的计算会重新执行。无故障进程既不停止也不回滚，但它们可能要重发已经发过的消息来帮助重启的新进程恢复到失效进程在发生故障时的状态[3,8-10]。与 checkpointing 方法相比，message-logging 减小了恢复的开销，但是实际应用中，记录消息和重发消息的开销比较大，限制了该方法的实用性[3,11]。当前高性能计算领域广泛应用的还是 checkpointing 技术。

传统的 checkpointing 技术在应用执行期间将应用进程空间的所有栈、堆和寄存器信息保存到可靠的存储器上，其开销的主要部分是将 checkpoint 文件写入可靠存储器所花的时间[12]。当问题规模和机器规模变大时，这种开销会很快增大，甚至不可接受。通过提高读写速度可以减小这种开销，例如 diskless checkpointing 技术，它用高速的内存代替低速的磁盘来存放 checkpoint 文件[1,12-14]。由于内存空间有限，为了减小存储文件的尺寸，这种方法往往要对 checkpoint 数据进行额外的编码(压缩)和解码运算。通过选择合适的 checkpointing 时机也可以减小开销，例如 application-level checkpointing，它由用户指定 checkpointing 的时机，可以选择最少的信息进行保存[3,15]。尽管这些优化手段可以在某种程度上改善 checkpointing 技术的性能，它们仍然有一些固有的性能局限。首先，即使在无故障情况下，checkpointing 的开销也是不可避免的，而且开销还可能比较大。第二，所有失效进程上的任务都在一个重启的进程上重算，其它无故障进程只能空闲，这不仅浪费了计算时间，而且使得恢复时间受限于故障和前一个 checkpoint 之间的间隔[16]。

1.3 本文方法的简介

本文提出一种开发容错并行程序的新方法：并行复算(Parallel Recomputing，简称 PR)。这个方法的主要特点是利用多个无故障进程的并行的计算分配给故障进程的任务。并行复算一方面可以减小保存状态的固有开销，另一方面通过并行计算加快了故障恢复过程。

本文的组织如下。第 2 节介绍并行复算方法。第 3 节举例说明如何用开发基于并行复算的容错的 MPI 程序。第 4 节提出并行复算的优化方法，该方法结合了应用级 checkpointing 的思想。第 5 节给出实验结果。最后总结全文并讨论未来工作。

2 并行复算

为了聚焦研究内容，我们对 MPI 做了 2 个通常的假设：1) 存在一个可靠的消息传输层，故障只发生在计算期间；2) MPI 负责故障检测(已经有这样的 MPI 实现，如 FT-MPI[17])。

我们将一个并行程序的执行分为 2 部分：串行段和并行段。在串行段，只有一个进程执行，其它进程空闲。在并行段，所有进程计算各自的任务。如果故障发生在不同段，并行复算的处理方法也不同。为了容忍发生在串行段的故障，采用多模冗余的容错思想，将计算复制到多个进程上，然后比较计算结果，实现容错。在 MPI 程序中很容易将串行段复制到多个进程上，例如将“if(myid. == 0) then”修改为“if(myid <3) then”。多个计算结果的比较所需的开销一般是非常小的，基本可以忽略。并行段通常占据计算时间的绝大部分，因此故障主要在并行段发生。我们主要介绍如何处理发生在并行段中的故障。

在并行段中如果用并行复算来容忍一个进程失效故障，若进程 P_0 失效，所有无故障进程继续执行，并在通信之前分担 P_0 上的负载。最终的结果由所有无故障的进程给出。即使在恢复阶段，所有无故障的进程也是并行工作的，没有计算能力的浪费，如图 1 所示。

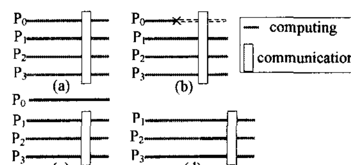


图 1 用并行复算容单进程故障

相对 checkpointing 而言，并行复算方法有 3 个特点：

• 利用计算能力而不是存储能力容错。相对于存储器而言，处理器的可靠性更高，并且性能提高更快。

• 基于并行复算的容错程序的故障恢复过程由该程序自动启动，而 checkpointing 技术要求故障恢复过程由操作员启动。

• 无故障情况下开销小。在这个例子，额外开销就是 MPI 感知故障所需的时间。若采用类似于 FT-MPI 的 heartbeat 机制检错，这个时间是非常短的。

3 基于并行复算的容错程序的设计

3.1 一般方法

定义 1 恢复段是通信与通信之间、通信与程序初始或结束之间的非空的计算块。

并行程序的执行可以看作是恢复段的序列。恢复段的数目是每个进程上通信操作数的最大值加 1，各进程的段数相同，具有相同标号的恢复段的计算相互独立且并行执行。如图 2 所示，通信操作数为 2，程序被划分为 3 个恢复段，S_i^k 是进程 P_i 的第 k 个恢复段。

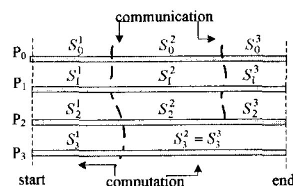


图 2 并行程序的恢复段

定义 2 并行恢复段中多个进程处于忙状态；串行恢复段中只有一个进程处于忙状态。

为了容忍发生在串行恢复段中的故障，我们将计算简单地复制到多个空闲进程上，最终的全局计算结果通过比较这些进程的局部结果得到。在并行恢复段，每个进程要在恢复段入口处保存某些能到达这个入口处的变量。这样，如果在该恢复段中发生故障，可以通过这些变量完成这一段的恢复。

在每个恢复段的末尾，基于并行复算的容错程序执行故障检测。如果检测到故障，所有无故障的进程执行恢复程序，并且在恢复之后继续执行。

3.2 一个基于并行复算的容错程序的实例

我们以 NPB EP 为例，给出基于并行复算的容错程序详细的设计方法。原始 NPB EP 程序的伪码如图 3 所示（要了解更细节的程序，请参看 NPB 源码）。

```
Step 1. Data partitioning
  Distribute L loops to N processes: np_i = L/N
  Compute the start loop: k_offset = myrank * np_i
Step 2. Main loop
  for k = k_offset to k_offset + np_i
    Compute result: R_i
  end
Step 3. Collect global result
  Call MPI_Allreduce(…)
  R = R_0 + R_1 + … + R_{N-1}
```

图 3 NPB EP

如图 3 所示，我们认为 EP 程序中只包含一次通信，整个程序被分为 2 个恢复段。当进入第二个恢复段，每个进程通过 MPI_Allreduce 获得最终结果，但只有 P_0 输出结果。第 2 个恢复段是串行段，我们用多个进程输出最终结果实现容错。对可能发生在第 1 个恢复段（并行恢复段）中的故障的检测和处理发生在该段结束通信之前。我们在 step 3 的 call MPI_Allreduce 语句之前检测和处理故障。相应的容错 EP（简称为 PR EP）的伪码如图 4 所示。

```
Step 1. Data partitioning
  Distribute L loops to N processes: np_i = L / N
  Compute the start loop: k_offset = myrank * np_i
  Set recovery flag: recovery = false
  Backup the current MPI communicator: call MPI_COMM_dup
(…)
Step 2. Main loop
  for k = k_offset to k_offset + np_i
    Compute result: R_i
  end
Step 3. Detect and process a single failure
  if (P_j fails and recovery = false) then
    Save the current result: R_i' = R_i
    Set recovery flag: recovery = true
    Change the number of processes: N = N-1
    Distribute np_j loops of P_j to N-1 surviving processes:
    np_i = np_j / (N-1)
    Compute the new start loop: k_offset = k_offset + …
    Recovery the MPI communicator: comm = oldcomm
    goto step 2
  endif
Step 4. Collect the global result
  if (recovery = true) then
    Combine two results: R_i = R_i + R_i'
  endif
  Call MPI_Allreduce(…)
  R=R_0+R_1+…+R_{N-1}
```

图 4 PR EP

图 4 中，在第 1 个恢复段末尾，也即第 2 步各进程完成各自的计算之后，检测并处理故障。如果没有发生故障，程序的执行与原始程序完全一样。如果进程 P_j 发生故障，剩下的 $N-1$ 个无故障进程首先保存各自当前的计算结果，然后获取 P_j 上负载的 $1/(N-1)$，通过并行计算完成 P_j 没有完成的工作。在第 4 步，如果前面发生过故障，则通过收集剩余无故障进程上 2 次的结果获得最终计算结果。由于 MPI 通信域会被进程故障破坏，我们在第 1 步备份它，在第 3 步恢复。

3.3 并行复算与 checkpointing 的比较

记处理器数为 N；NPB EP 的全部执行时间为 T，包括串行部分 T_S 和并行部分 T_P，即 $T=T_S+T_P$。当我们把 NPB EP 升级到 PR EP 后，PR EP 中会出现一些额外开销：1）固有开销 T_S'，主要是备份 MPI 通信域的开销；2）容错开销 T_S^* 和 T_P^*，故障恢复过程中的串行计算。并行复算 EP 的全部执行时间如图 5 所示。

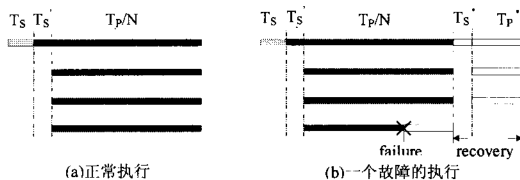


图 5 PR EP 的执行

如图 5 所示，如果没有故障，并行复算 EP 的额外开销（O_{PR}^0）是

$$O_{PR}^0=T'_S \tag{1}$$

如果发生 1 个故障，并行复算 EP 的额外开销（O_{PR}^1）是

$$O_{PR}^1=T'_S+T_S^*+T_P^*\approx T'_S+T_S^*+\frac{T_P}{N(N-1)} \tag{2}$$

其中，T_S^* 主要是图 4 中第 3 步的时间。组成开销的 3 部分中，T_S' 和 T_S^* 通常是小项；占主要部分的是 $\frac{T_P}{N(N-1)}$，它随着 N 增大而迅速减小。我们总结并行复算 EP 的特征如下：

- 无故障时，RP EP 的开销可忽略。
- 发生 1 个故障时，开销随着参与计算的处理器数目增大而减小。
- 无故障进程上从发生故障和恢复之间的计算没有浪费，不需在恢复后再重新计算。

采用 checkpointing 技术时 EP 的执行情况如图 6 所示。故障发生在 t_1，故障前的最后一次 checkpointing 完成在 t_0，恢复过程在 t_2 时刻启动。$t_a=t_1-t_0$，$t_b=t_2-t_1$，t_b 表示故障和恢复处理之间的延时。I 是 checkpoint 间隔，$k=\lfloor T/I \rfloor$，表示 checkpoint 的次数。C 表示一次 checkpointing 的时间，R 表示一次恢复的时间。通常，C 和 R 都随着问题规模而增大。恢复之后，t_0 和 t_2 之间的计算要重算。

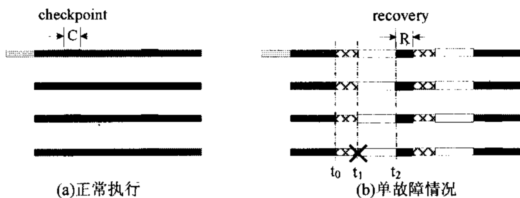


图 6 带 checkpointing 的 EP 的执行

无故障时，额外开销为 $O_{ckpt}^0=k\times C$。1 个故障时，额外开销为 $O_{ckpt}^1=k\times C+R+t_a+t_b$。对 EP 而言，基于 checkpoin-

ting 的容错方法有如下特征：

• 不管有没有故障，额外开销都相当大，且随问题规模增大。

• 发生 1 个故障时，如果考虑前一次 checkpoint 和恢复之间浪费掉的计算，容错开销更大。

• 无故障进程上从发生前一次 checkpoint 和恢复之间的计算被浪费掉了，要在恢复时再重新执行一次。

3.4 多个故障的情况

上述并行复算的容错可以推广到多个故障的情况。如果一个并行复算段中发生了 S 个故障，则要将分配给这 S 个故障进程的任务再划分给无故障进程。一种简单的划分方法是针对每个故障进程的任务分别进行划分，也就是重复 S 次容单故障时的划分过程。

4 并行复算与 checkpointing 的结合

根据前面的分析，并行复算容单故障的开销随着 N 增大而减小。然而，如果问题规模非常大，复算的开销可能仍然较大。受到 checkpointing 技术的启发，我们可以通过保存计算的中间状态进一步减小复算的工作量。图 7 给出了带 checkpointing 的 PR EP 的伪码(简称为 PRC EP)。

```
Step 1. Data partitioning
  Distribute L loops to N processes: np_i = L / N
  Compute the start loop: k_offset = myrank * np_i
  Set recovery flag: recovery = false
  Backup the current MPI communicator: call MPI_COMM_dup (…)
Step 2. Main loop
  for k = k_offset to k_offset + np_i
    Compute result: R_i
    Checkpointing one every ckpt loops:
      if (k mod ckpt == 0) then
      Save the current result: R_i'' = R_i
      Save the current k: kk = k
    endif
  endfor
Step 3. Detect and process a single failure
  if (process j fails and recovery = false) then
      Save the current result: R_i' = R_i
      Set recovery flag: recovery = true
      Change the number of processes: N = N-1
      Distribute np_j loop of process j to N-1 surviving processes:
      np_i = (np_j - kk)/(N-1)
    Compute the new start loop: k_offset = k_offset + kk + …
     Recovery the MPI communicator: comm = oldcomm
     goto step 2
endif
Step 4. Collect the global result
  if (recovery = true) then
    Combine three results: R_i = R_i + R_i' + R_i''
  endif
  Call MPI_Allreduce(…)
  R = R_0 + R_1 + … + R_{N-1}
```

图 7 带 checkpointing 的 PR EP (PRC EP)

并行程序中，迭代通常占据了大部分时间。因此我们可以通过在第 2 步的循环中插入 checkpointing 语句来减小复算的工作量，如图 7 中斜体字所示。第 2 步中所需保持的变量只有 2 个，即局部计算结果 R_i 和循环次数 k，但是却有可能显著减小复算的计算量。

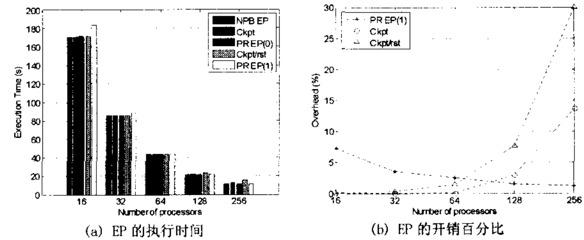
5 实验评估

我们通过 2 组实验评估 PR 和 PRC 这两种容错方法的性能。首先，在 256 结点的 Itanium cluster 系统上评估 NPB kernels EP，CG，FT 和它们的并行复算版本程序，每个结点有 2 个 3.2 GHz CPUs 和 4 GB 内存。操作系统是 Redhat Linux Advanced Server 4。第二组，在 4 结点的 XEON cluster 上比较 PR EP 和 PRC EP，每个结点有 3.2 GHz CPUs 和 4GB 内存，操作系统是 Redhat Linux 9。

我们采用 MPICH2-1.0.5 作为并行计算平台，通过杀死进程的方法模拟 fail-stop 类型的故障。我们修改 MPICH2-1.0.5，增加了 2 个全局变量 *HasFailed* 和 *FailedRank*，使得进程能通过故障检测接口获取失效进程的 rank 号。

5.1 PR 与 Checkpointing 的性能比较

本实验的问题规模都是 class D。不同版本的执行时间如图 8 所示。PR EP(0)表示 PR EP 在无故障情况下的结果，PR EP(1)表示 PR EP 在 1 个故障情况下的结果。Ckpt 表示 NPB EP 做 1 次 checkpoint，Ckpt/rst 表示 NPB EP 做 1 次 checkpoint 和 1 次 recovery。后面我们采用类似的表示方法。Ckpt/rstd 的执行时间没有包含前 1 次 checkpoint 和恢复之间浪费掉的计算时间，即图 6 中 t_0 到 t_2 的时间。

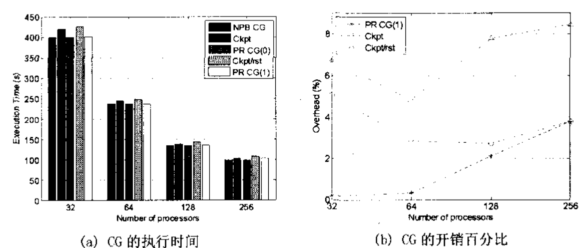


(a) EP 的执行时间 (b) EP 的开销百分比

图 8 EP 的实验结果

无故障时，PR EP 的执行时间几乎等于 NPB EP。由于 1 个处理器上 checkpoint file 只有大约 2.5MB，checkpointing 的开销在 N 较小时是非常低的。但全部 checkpoint files 随着 N 增大而增加。当 N 增大到 256 时，Ckpt 的开销为 13.65%，Ckpt/rst 的开销为 29.56%。相对地，PR EP(1)的开销随着 N 增大而减小，用 256 处理器时开销仅为 1.19%。

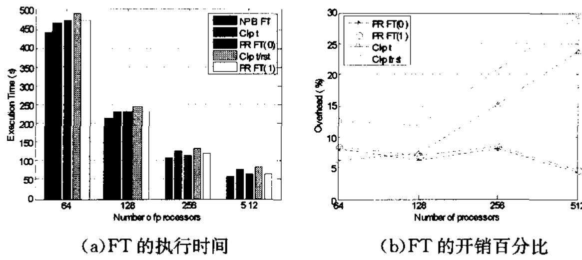
CG 的实验结果如图 9 所示。ckpt/rst 开销并不大，因为 CG 执行时间非常长。用 256 处理器时，ckpt 开销为 3.77%，ckpt/rst 开销为 8.44%。无故障时，PR CG 与原始 CG 的执行时间几乎相同。随着处理器数目增加，容错开销增大，因为并行复算的通信开销增大了。用 256 处理器时，开销为 3.74%。



(a) CG 的执行时间 (b) CG 的开销百分比

图 9 CG 的实验结果

FT 的实验结果如图 10 所示。ckpt/rst 开销随着处理器数目增加而增大。用 512 处理器时，checkpoint 开销为 23.82%，ckpt/rst 开销为 29.3%。1 个故障时，我们用 2 个进程并行恢复，开销低于 10%。无故障时，PR FT 与 NPB FT 的执行时间几乎相同。

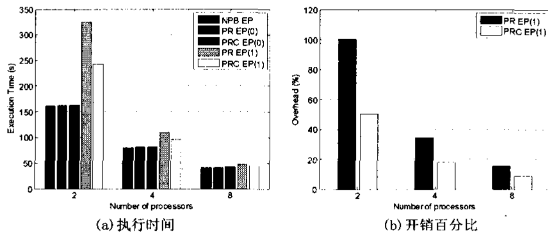


(a)FT 的执行时间　　(b)FT 的开销百分比

图 10　FT 的实验结果

5.2　PR 与 RPC 的性能比较

这组实验中 EP 的规模为 class B，某个进程在主循环的正中间被杀掉。在 PRC 实验中，每 120 次迭代做 1 次 checkpoint。RP EP 和 PRC EP 的执行时间和开销如图 11 所示。



(a) 执行时间　　(b) 开销百分比

图 11　PR 和 RPC 的比较

从图 11 可知，无故障时，PR EP 和 PRC EP 的开销都可以忽略不计。当发生 1 个故障时，RPC EP 的性能优于 RP EP。但是性能差异随着 N 增大而快速降低。因此，PRC 的性能优势与恢复过程的计算规模有关：规模越大，优势越明显。

结束语　本文的容错方法并行复算采用了与现有方法如 checkpointing 和 message-logging 不同的容错方式，它使得故障恢复并行完成。我们介绍了如何用并行复算实现容错的程序，还提供了并行复算的一种优化方法，将 checkpointing 与之结合。实验结果表明，不管有没有发生故障，并行复算的开销百分比都低于 checkpointing，并且随着处理器数目增大而减小；RPC 的性能则更优于并行复算，这种性能优势与问题规模有关，问题规模越大，则优势越明显。

参 考 文 献

[1] Engelmann C, Geist A. A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform// CLADE 2003: 1st International Workshop on Challenges of Large Applications in Distributed Environments. Seattle, WA, USA, 2003

[2] IBM Research. Blue gene Project overview. Online at http://www.research.ibm.com/bluegene/, 2002

[3] Elnozahy E N, et al. A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys, 2002, 34(3): 375-408

[4] Chandy K M, Lamport L. Distributed snapshots: Determining global states of distributed systems. ACM Trans. Comput. Systems, 1985, 3(1): 63-75

[5] Plank J S, Li K. Ickp—a consistent checkpointer for multicomputers. IEEE Parallel Distrib. Technol., 1994, 2(2): 62-67

[6] Stellner G. CoCheck: Checkpointing and process migration for MPI// Proc. 0th International Parallel processing Symposium. 1996

[7] Elnozahy E N, Johnson D B, Zwaenepoel W. The performance of consistent checkpointing // Proc. 11th Symposium on Reliable Distributed Systems. 1992

[8] Elnozahy E N, Zwaenepoel W. On the use and implementation of message logging // 24th International Symposium on Fault-Tolerant Computing. 1994

[9] Strom E, Yemini S. Optimistic recovery in distributed systems. ACM Transactions on Computer Systems, 1985, 3(3): 204-226

[10] Louca S, et al. MPI-FT: Portable fault tolerance scheme for MPI. Parallel processing Letters (PPL), 2000, 10(4)

[11] Bouteiller A, et al. MPICH-V project: A Multiprotocol Automatic Fault-tolerant MPI. International Journal of High Performance Computing Applications, 2006, 20(9): 319-333

[12] Plank J S, Li K, Puening M A. Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst, 1998, 9(10): 972-986

[13] Silva L M, Silva J G. An Experimental Study About Diskless Checkpointing// Proceedings of the 24th EUROMICRO Conference. 1998

[14] Plank J S, Kim Y, Dongarra J. Algorithm-based diskless checkpointing for fault tolerant matrix operations// 25th International Symposium on Fault-Tolerant Computing. Pasadena, CA, 1995

[15] Beguelin A, Seligman E, Stephan P. Application level fault tolerance in heterogeneous networks of workstations. Journal of Parallel and Distributed Computing, 1997, 43(2): 147-155

[16] Chakravorty S, Kal L V. A Fault Tolerance protocol with Fast Fault Recovery // IPDPS2007: 21st IEEE International Parallel & Distributed processing Symposium. Long Beach, California USA, 2007

[17] Fagg G, Dongarra J. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World // Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000. Hungary: Springer Verlag, 2000

(上接第 13 页)

[22] Marcus S, Subrahmanian V S. Foundations of multimedia database systems. Journal of ACM, 1996, 43(3): 474-523

[23] Kau S-C, Tseng J C R. MQL: a query language for multimedia database. Multimedia Communications, 1994: 511-516

[24] Li J Z, Ozsu M T, Szafron D, et al. MOQL: a multimedia object query language// Proceedings of the 3rd International Workshop on Multimedia Information Systems. 1997: 19-28

[25] 田增平，党华锐，周傲英. 多媒体对象查询语言及其查询处理. 软件学报，1999, 10(7): 694-701

[26] Cao Z S, Wu Z D, Wang Y Z. UMQL: A unified multimedia query language// Proceedings of the 2007 IEEE International Conference on Signal Image Technology and Internet Based Systems. IEEE Computer Society, 2007: 101-107

[27] Dionisio J D N, Cardenas A F. MQuery: a visual query language for multimedia, timeline, and simulation data. Journal of Visual Language and Computing, 1996, 7(4): 377-401

[28] Schmitt I, Schulz N, Herstel T. WS-QBE: a QBE-like query language for complex multimedia queries// Proceedings of the 11th International Conference on Multimedia Modeling. IEEE Computer Society, 2005: 222-229

[29] 吴宗大，曹忠升，王元珍. 可视化多媒体查询语言的设计与实现. 华中科技大学学报，2008, 7(36): 45-49