高性能计算节点中的同步操作加速引擎设计

陈 飞，曹 政，王 凯，胡农达，安学军

(中国科学院计算技术研究所 北京 海淀区 100190)

【摘要】随着GPU等加速部件在超级计算领域的广泛应用，超级计算机单个节点的硬件并行度比单核时代高几倍甚至几十倍。在该环境下，并行应用于单个芯片、计算节点内和计算节点间的通信密度较单核时代急剧增加，通信瓶颈问题愈发突出。为应对高并行度带来的通信瓶颈问题，提出一种同步引擎的硬件设计，该同步引擎可有效地支持和加速计算节点内多任务间频繁小数据量传输(细粒度同步)以及计算节点内和节点间的Barrier、All-reduce集合操作，进而加速并行应用的性能。测试结果表明，在16进程规模下的集合操作测试中，同步引擎相比传统的软件实现有约4倍的加速，在三角矩阵分解(LU分解)测试程序中可以获得约20%的性能提升。

关 键 词 集合操作； 通信系统； 计算节点； 细粒度同步； 高性能计算机； 混合编程； 消息传递

中图分类号 TP303 **文献标识码** A doi:10.3969/j.issn.1001-0548.2012.01.018

Design of Synchronization Accelerator in HPC Computing Node

CHEN Fei, CAO Zheng, WANG Kai, HU Nong-da, and AN Xue-jun

(Institute of Computing Technology, Chinese Academy of Science Haidian Beijing 100190)

Abstract With the widely use of acceleration devices, hardware parallelism of single hybrid programming computer (HPC) node has increased many. As a result, both on-chip communication and inter-node communication become more and more frequently. Apparently, communication is becoming the bottleneck of system performance. This paper proposes a design of hardware module called synchronization accelerator to accelerate synchronization communication patterns. These patterns include fine-grain synchronization, barrier, and all-reduce. At the scale of 16 processes, synchronization accelerator can achieve about 4 times speedup than software-based collective operations. Also, the performance of benchmark LU can achieve 20% improvement with the use of synchronization accelerator.

Key words collective operation; communication system; computing node; fine-grain synchronization; high performance computer; hybrid programming; message passing

1 背景介绍及分析

随着众核处理器、GPU及FPGA等计算加速部件的广泛使用，单个计算节点的硬件并行可以轻易达到数百个进程/线程并发执行的程度。不断提高的并行和计算密度，对系统的通信能力提出了更高的要求。而对于数据长度短，且规模敏感的同步/集合操作，其性能面临着更严峻的挑战。

面对上述挑战，本文通过对同步/集合操作的特点进行分析，设计并实现了一种细粒度同步及Barrier、All-reduce集合操作的加速引擎。

1.1 细粒度同步

常见的编程模型有消息通信模型和共享存储编程模型，前者使用消息传输数据，如典型的MPI编程；后者使用Load/Store指令传输数据，如OpenMP编程。在紧耦合的计算节点内，由于多个计算单元之间通信的延时相对节点间延时要低，因此MPI程序对延时的容忍特性在该环境下不能很好地得到发挥。在一个计算节点内，分别使用MPI和Load/Store进行通信的多个测试程序，得到的性能对比结果如图1所示。从对比结果可以看到，在部分测试程序中，使用Load/Store比使用MPI进行通信有更高的执行效率，加速比最高超过4倍。该现象说明：该类程序具有频繁小数据传输的通信特征，使用MPI进行通信，需要为传输的数据附加额外的信息头部以指示数据长度、路由等信息，但会造成实际传输数据量增加。

收稿日期： 2011 – 07 – 15；修回日期： 2011 – 11 – 15

基金项目：国家863计划(2009AA01A129)；国家自然科学基金(61100014)

作者简介：陈飞(1981 –)，男，博士，主要从事计算机体系结构和高性能互连网络方面的研究.

并且，额外信息的发送和接收涉及数据拷贝、完成事件检测、包匹配等过程，因此在传输小数据时传输效率很低。而使用Load/Store传输数据所需要的额外信息较少，操作涉及的步骤相对MPI要少，因此在传输小数据量时有较高的传输效率。相关研究也表明，在相对紧耦合的硬件环境下，使用共享存储的编程模型，可以在特定的应用中取得比消息传输更好的性能。因此，在节点内支持共享存储编程非常必要。

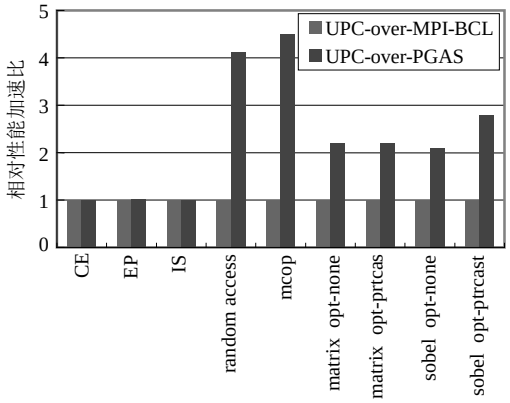

图1　消息通信和共享存储通信的性能对比

为支持共享存储编程，除需提高硬件通信的带宽延时性能外，还需改进多个进程或者线程之间的同步机制，以实现同步操作的加速。

通常，粗粒度同步被广泛使用。为了保证并行应用中多个任务间的RAW(read-after-write)次序，需要在对同一个数据进行操作的Store指令后面、Load指令的前面使用一个Barrier调用，以实现多任务之间的同步。由于编程者在编写共享存储时不关心数据在哪些进程或者线程之间流动，因此Barrier操作是粗粒度的全局同步操作。相关研究表明，粗粒度同步操作有可能导致多个进程或者线程之间的假依赖关系，从而造成性能损失。

细粒度同步[1]被提出以解决频繁小数据传输以及多任务之间的同步问题，使得在多任务发出多个Load/Store操作时，只对同一个地址的Load/Store进行它们之间的同步，而不需要粗粒度的全局同步。

1.2　集合操作

集合操作是一种在两个及两个以上的进程/线程间进行显式数据迁移和控制调用，如Barrier、Reduce、Multicast、All-to-All等操作。该类操作的特点是执行时间随着参与任务个数的增多而增长，最终与性能、算法和网络拓扑有着直接联系，并且操作性能对并行应用的性能有着关键的影响。

常见的集合操作算法[2]包括中央计数器算法、树形算法、Butterfly算法和Tournament算法等。其中Butterfly和Tournament算法在类似机群系统中有较好的可扩展性，但在多核甚至众核环境下，该类算法仍会导致系统性能的急剧下降。

1.3　验证系统

本文提出一种同步引擎设计，对上述的细粒度同步和多核环境下的集合操作进行硬件加速。

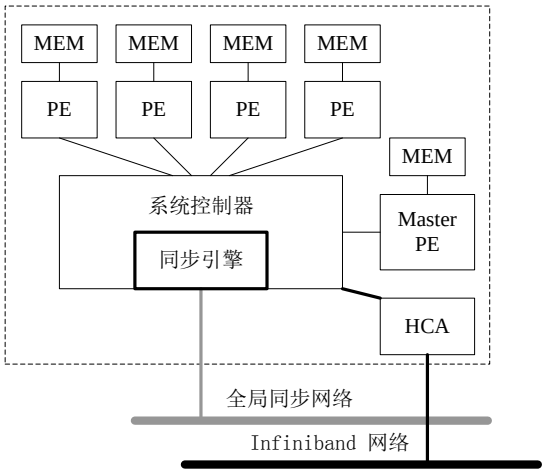

图2　超龙一号节点结构示意图

本文工作基于中国科学院计算技术研究所研制的超龙一号超级计算机展开，超龙一号计算节点的结构如图2所示。计算节点有5个处理单元(PE)，其中一个是主PE，其他4个是从PE。主PE采用×86通用处理器，而从PE采用低功耗的龙芯MIPS处理器。一个系统控制器提供节点内互连以及节点间互连。节点间采用两套网络互连，一套采用商用的Infiniband网络组成高带宽、高可扩展性的单播网络；另一套采用定制的全局同步网络，加速系统范围内的Barrier和All-reduce集合操作。

2　相关工作

HEP[3]、Tera[4]和 MIT Alewife [5]等系统均使用细粒度同步硬件支持。通过在每个字(32 位或者 64 位)存储空间前面增加 1 位或者多位的状态位，配合定制的 Load/Store 指令，通过执行读写操作时对状态位进行判断和修改，实现细粒度同步。这些方法依赖于定制的处理器和存储器，因此具有成本高、难以维护的缺点，不适应当前超级计算以商业化部件为主流的发展趋势。

近年来，由于单芯片硬件并行度的增加，一些文献在片上系统如 SMT 平台、CMP 平台[6]以及 Many-core 平台[7]上研究细粒度同步支持，这些工作没有使用定制的存储芯片，但仍然需要定制处理器和定制指令以支持细粒度同步。这些工作的缺点是

仅仅支持片上系统的细粒度同步，不能扩展至多处理器平台。另外，研究均仅局限于模拟器，未有真实系统予以验证。

基于硬件进行集合操作加速的研究，大多集中在节点间的加速，不对节点内和节点间加以区分，如 CM-5[8]和 IBM Blue Gene[9]。近年来，由于单个处理器芯片的核数增多，节点内硬件并行度增加，一些研究发现在设计集合操作算法时，区分节点内和节点间可以更好地利用节点内的通信局部性，进而加速整个集合操作的执行。文献[10]提出在 Infiniband 网络中区分节点内和节点间的广播操作算法，文献[11]提出在 QsNet 网络中区分节点内和节点间的 All-gather 操作算法。虽然两种算法在节点间使用硬件加速集合操作，但是在节点内仍然使用软件的方法实现集合操作的计算和数据迁移。

本文提出的同步引擎在不使用定制处理器指令集以及存储器的前提下，能很好地支持细粒度同步操作和 Barrier、All-reduce 集合操作。虽然在硬件上区分节点内和节点间集合操作，但是软件接口并不区分节点内和节点间，使得软件人员编程更加容易。同时引擎提供的集合操作软件接口基于共享存储模型，而不是消息传递，因此能更有效地支持混合编程。

3 实现原理

3.1 共享同步存储结构

本文提出的同步引擎基于 Tag-Value 的存储结构，该存储结构被命名为共享同步存储结构，如图 3 所示。Value 域用于存储普通数据，宽度为 32 位；L-bit 用于实现细粒度的锁操作，宽度为 1 位；P-bit 用于实现保证 read-after-write 次序的存储–读取操作，宽度为 1 位；counter 用于实现集合操作，也被用于实现多种模式的 Read-After-Write 次序的 Load/Store 操作，宽度为 8 位。

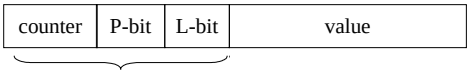

图3 共享同步存储的数据结构

同步引擎使用系统控制器的片内 SRAM 以及主 PE 上的部分主存作为存储空间，并使用控制逻辑实现基于物理地址映射的 hash 表。通过该表将节点内主存(存储 value 值)映射为图 3 所示的 Tag-Value 数据结构，通过附加的 Tag 域以实现细粒度同步和集合操作的硬件同步支持。由于仅在 hash 的存储空间内实现 Tag-Value 数据结构，因此不需要更改通用存储空间的结构，也不需要定制处理器所管理的存储器。

3.2 同步引擎支持的基本操作

基于共享同步存储数据结构包含同步引擎支持维护 read-after-write 次序的读操作(ssm_get)和写操作(ssm_put)、细粒度的加锁(ssm_lock)和解锁(ssm_unlock)操作，以及集合操作 Barrier 和 All-Reduce。它们的示意性软件接口的含义如下：

ssm_put(address,value)：把value值存储到hash表的与address对应的数据结构中，并更改Tag-Value结构中的P位和counter位的状态，表示该数据可用。

value=ssm_get(address,size)：从 hash 表的与address对应的数据结构中尝试读取数值。在读取数据之前探测对应的P位和counter位的状态，如果该数据已经被标识为可用，则向软件返回数据结构中的数据，并更改P位和counter位的状态；否则该命令自动在同步引擎中排队，等候下一次调度。

ret=ssm_lock(address)：对 hash 表中对应于address的数据结构加锁。在加锁前测试对应Tag-Value中L位的状态，如果该数据已经被加锁，则该ssm_lock命令自动在同步引擎中排队，等候下一次调度；否则执行加锁操作，更改L位状态，并向软件返回加锁成功的信息。

ssm_unlock(address)：对hash表中对应于address的数据结构解锁，并在解锁完毕后更改L位的状态。

ret=ssm_barrier(address,size,io)：使用hash表中对应于address的数据项执行Barrier操作，参数sizeo为节点内参与该Barrier的任务数目。参数io表示该操作是节点内Barrier还是节点间的Barrier。同步引擎通过更改Tag-Value数据结构中的counter位判断是否所有任务都已经达到同步点。当达到同步点时通过返回值通知软件。

ret=ssm_reduce(address,value,type,op,size,io)：使用hash表中对应于address的数据项执行Reduce操作，参数value为参与Reduce运算的数据；参数type为数据类型；参数op为Reduce的运算类型；size为节点内参与Reduce运算的任务数；io为该操作是节点内Reduce还是节点间Reduce。同步引擎通过更改Tag-Value数据结构中的counter位判断是否完成计算，并把计算结果存放到Value域中。当完成计算时通过返回值通知软件，由软件主动使用Load指令到hash表中读取计算结果。

ret=ssm_burst_reduce(src,dest,length,type,op,size,io)：为实现批量 Reduce 操作，同步引擎支持该批量 Reduce 操作，在软件接口中每个任务指定需要计算的源地址(src)、目的地址(dest)、数据长度(length)、

数据类型(type)、操作类型(op)、任务数量(size)、是否节点内操作(io)，同步引擎会自动加载数据，并完成它们之间的运算，把计算结果存放到指定的缓冲之后，通过返回值通知软件完成状态。

4　同步引擎的硬件实现

同步引擎的硬件实现包括同步引擎的微体系结构设计及它的工作方法。

4.1　同步引擎的微体系结构

同步引擎被实现在系统控制器内部，节点内通过系统总线与 5 个处理单元互连，对节点间通过系统控制器提供的全局同步网络实现互连。当同步操作为节点内操作时，如节点内 Barrier、节点内 All-reduce，则无需访问全局同步网络；当同步操作是节点间操作时，则首先完成节点内的同步和计算，然后把同步消息和计算结果发送到全局同步网络中完成全局同步和计算，最后在全局网络中接收完成消息和计算结果，通过与节点内操作同样的方式通知软件。同步引擎的微体系结构如图 4 所示，各模块含义如下。

操作队列模块：用于连接多个处理单元的输入缓冲和调度。同一个处理单元上的不同进程或线程的操作被缓存到同一个队列中的不同 FIFO 中，以保证不同进程或线程的操作不会由于排队和调度原因而相互阻塞。

Reduce 模块：用于执行 ssm_reduce 操作，可以完成 64 位整型、双精度浮点的 Reduce 运算，支持“求和”“最大”和“最小”运算。

Burst-Reduce 模块：用于执行 ssm_burst_reduce 操作，可以高效地完成批量的 Reduce/All-reduce 计算，支持 32 位整型、64 位整型和双精度浮点类型，支持“求和”“最大”和“最小”运算。

Barrier 模块：用于执行 ssm_barrier 操作。

处理核 0 和处理核 1 模块：用于执行同步引擎支持的 ssm_put、ssm_get、ssm_lock 和 ssm_unlock 操作。使用两个处理核心模块执行细粒度同步操作是为了增加处理细粒度同步的吞吐率，使之不成为处理的瓶颈。一个调度模块在两个处理核心模块之间调度各个操作的分配，保证一个进程或线程的操作不会在两个处理核心模块中乱序执行。

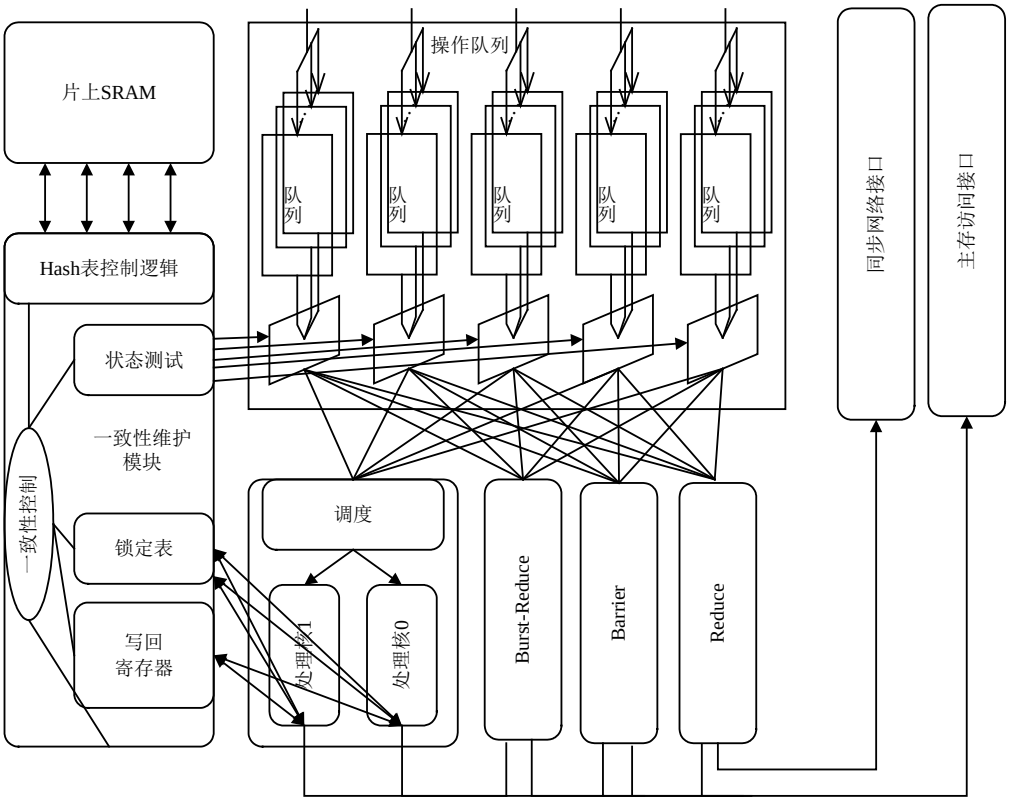

图4　同步引擎微体系结构

一致性维护模块：用于维护同步引擎中多个功能模块的原子操作以及存储一致性。由于多个功能模块可能会并发操作同一个变量，因此功能模块在操作前需要到一致性维护模块上对该变量进行加锁以保证原子性。并且在对变量计算后，自动广播到正在等待该数据的功能模块中。

片上 SRAM 模块：实现一个 4 路组相联的 hash 表存储结构，并且实现多个读写端口，当访问 hash

表中的项没有冲突时，可以允许最多 4 个操作者发起的并发读取或者写入。

4.2 软硬件交互模式

同步引擎将其寄存器通过内存映射的方法直接映射至存储地址空间，软件通过写内存映射的地址空间(memory mapped I/O address space)的寄存器向同步引擎发送各种操作。该写操作是流水发送的，因此相对普通的I/O写操作具有高带宽、低延时的特点。

处理单元接收从同步引擎发送的数据的过程相对复杂，是通过处理单元内存中如图5所示的控制页表实现的。每个处理单元的内存中都有一个控制页表，该控制页表被处理单元上所有的进程或线程共享。图5描述了一个具有*m*个槽位的控制页表。初始化时软件把页表的起始物理地址start_address发送到同步引擎，但软件发送一个需要返回值(除了ssm_burst_reduce的计算结果以外)的同步操作时，在发送的命令参数中都会包含一个偏移信息，如ret_offset 0。当同步引擎执行该命令后，根据ret_offset信息把计算结果以及计算状态写入到控制页表对应槽位中的Value和State域。软件通过查询State状态，判断Value域中的数据是否可用。

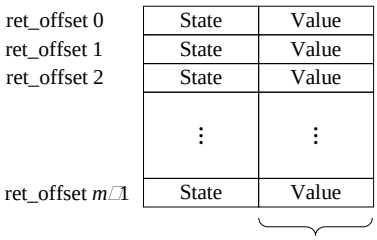

图5 控制页表结构示意图

4.3 Hash表的设计

为了实现内存到Tag-Value结构的共享同步存储映射，同步引擎使用片内的SRAM空间以及主处理单元的部分内存空间存储一张分布式的hash表。该hash表的每一项都包括Tag-Value数据结构，以及用于hash索引的关键字。

该hash表的容量可动态配置。同步引擎首先使用片内的SRAM作为hash表的存储，当片内SRAM溢出时，使用一块在系统启动时就预留的256 MB物理内存存放hash表的数据。当该块预留的物理内存也溢出时，同步引擎向处理器发送中断，由中断处理程序动态分配物理内存。所有的上述存储空间都以链表的方式被统一管理。

文献[7]证明很多应用都具有很好的访存局部性，因此片内SRAM溢出的概率不大，本文的测试也证明了这一点。

5 性能评价

本文基于系统控制器FPGA原型系统对同步引擎进行测试和分析。该原型系统上使用1个AMD Opteron处理器作为主处理单元，4个龙芯3A处理器分别作为4个从处理单元。同步引擎的实现采用Xilinx FPGA，工作频率为200 MHz，系统采用带宽为1.6 GB/s的HyperTransport总线。同步引擎的片内SRAM容量为4 096个Hash表项。

5.1 细粒度同步的带宽测试

1) 使用ssm_put和ssm_get测试伪代码如下：

```
task0                    task1
while (in test)          while(in test)
ssm_put(A);              ssm_get(A);
```

2) 使用Barrier测试伪代码如下：

```
task0                    task1
while (in test){         while(in test){
Write(A);                Barrier;
Barrier; }               Read(A); }
```

使用上述伪码对同步引擎支持的ssm_put和ssm_get进行测试，把它们的数据传输和使用Barrier的方式得到的性能进行对比。测试中使用软件实现Barrier和同步引擎支持的ssm_barrier两种方式。比较传输数据块*A*时的带宽，使用同步引擎支持的细粒度同步在频繁传输小数据时可以获得比Barrier方式更高的带宽，如图6所示。

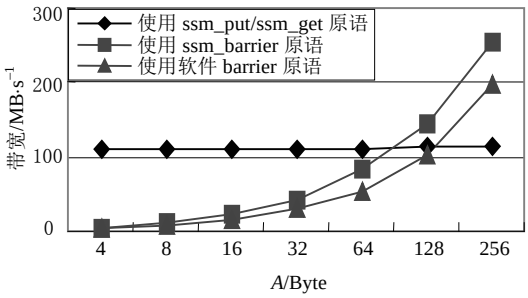

图6 两种同步方式得到带宽对比

5.2 矩阵三角分解测试

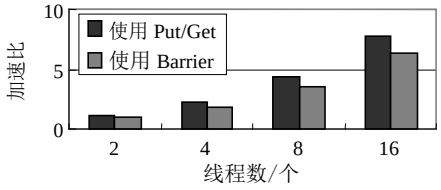

图7 LU分解结果对比图

矩阵三角分解(LU分解)是许多科学应用的核心算法之一，是斯坦福大学公布的Splash 2测试集中重

要的一个测试程序。本文使用ssm_put/ssm_get重写共享存储的LU分解程序，并用1024×1024大小的双精度浮点矩阵进行LU分解测试，测试结果如图7所示。结果表明，使用同步引擎支持的细粒度同步操作有约19%的性能提升，并且在该测试过程中没有片上SRAM的溢出。

5.3　Barrier和Reduce的性能测试

测试中，本文使用Tournament算法实现了软件的Barrier和Reduce，把它们与同步引擎硬件支持的ssm_barrier和ssm_reduce进行了对比。在不同的线程参与下，性能对比如图8所示，图中前缀HW表示硬件实现的性能，前缀SW表示软件实现的性能。可以看出，在16个线程参与下，使用同步引擎实现的Barrier和Reduce操作可以有约4倍的加速比，并且随着进程数的增多，该加速比会继续增大。

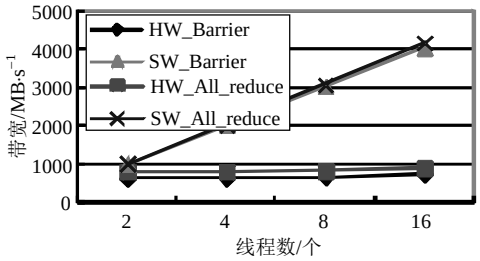

图8　软硬件实现的Barrier和Reduce性能对比

ssm_burst_reduce操作相对ssm_reduce有较大的启动延时，但由于其类似于DMA的工作方式，因此有较高的执行效率，测试结果如图9所示。

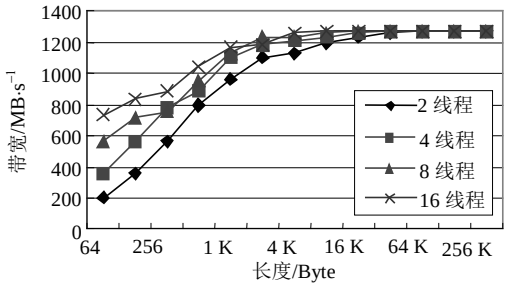

图9　批量数据的Reduce操作性能

从测试结果可以看到，当批量Reduce的数据量增大时，ssm_burst_reduce的带宽逐渐饱和，最终达到约1.2 GB/s。该性能受限于系统控制器的HyperTransport系统总线的1.6 GB/s带宽。去除总线协议本身的开销，同步引擎执行ssm_burst_reduce的有效带宽利用率约为90%。此外，参与运算的进程数目增多时，在较少数据量时即可达到饱和带宽。

6　结　论

本文提出的同步引擎设计，支持细粒度同步及集合操作。该设计在不依赖于特定处理器指令集以及存储器条件，可加速并行应用，并且使用统一的存储结构支持共享存储和消息通信的数据迁移方式，对于研究编程模型有很好的探索意义和实际意义。同时，该设计支持异构的处理器和加速部件，因而能够更好地适应当前的超计算机的设计趋势。测试结果表明，本文设计的同步引擎达到了较好的性能。

参　考　文　献

[1] ARVIND, RISHIYUR S N, KESHAV K P. I-structures: Data structures for parallel computing[J]. ACM Transactions on Programming Languages and Systems, 1989, 11(4): 598-632.

[2] MELLOR M J, SCOTT L M. Algorithms for scalable synchronization on shared-memory multiprocessors[J]. ACM Transactions on Computer Systems (TOCS), 1991, 9(1): 21-65.

[3] SMITH B J. Architecture and applications of the HEP multiprocessor computer system[C]//Proc 4th Real Time Signal Processing. Bellinghan, WA: SPIE, 1981: 241-248.

[4] ALVERSON R, CALLAHAN D, CUMMINGS D, et al. The tera computer system[C]//Proceedings of the 4th International Conference on Supercomputing. New York, NY: ACM, 1990:1-6.

[5] KRANZ D, LIM Beng-hong, AGARWAL A. Low-cost support for fine-grain synchronization in multiprocessors [DB/OL]. [2011-04-28]. http://citeseer.ist.psu. edu/ kranz94lowcost.html.

[6] FIDE S, JENKS S. Architecture optimizations for synchronization and communication on chip multiprocessors [C]//IEEE International Symposium on Parallel and Distributed Processing. Miami: IEEE Press, 2008:1-8.

[7] ZHU W R, SREEDHAR V C, HU Z, et al. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures[C]//Proceedings of the 34th annual International Symposium on Computer Architecture. New York: ACM. 2007:35-45.

[8] HILLIS W D, TUCKER L W. The CM-5 Connection Machine: a scalable supercomputer[J]. Communications of the ACM, 1993, 36(11): 30-40.

[9] GARA A, BLUMRICH M A, Chen D. Overview of the Blue Gene/L system architecture[J]. IBM Journal of Research and Development, 2005, 49(2): 195-212.

[10] MAMIDALA A R, CHAI L, JIN H W, et al. Efficient SMP-aware MPI-level broadcast over Infiniband hardware multicast[C]//Proceeding of the 20th International Parallel and Distributed Processing Symposium. Washington DC: IEEE Press, 2006: 305-312.

[11] YING Q, AFSAHI A. RDMA-based and SMP-aware multi-port all-gather on multi-rail QsNet II SMP clusters [C]//International Conference on Parallel Processing. Xi'an: IEEE Press, 2007: 48-48.

编　辑　张　俊