一个结构网格并行CFD程序的单机性能优化

车永刚[1] 张理论[1] 王勇献[1] 徐传福[1] 刘 巍[1] 王正华[1] 刘化勇[2]
(国防科技大学计算机学院 长沙 410073)[1] (空气动力学国家重点实验室 绵阳 621000)[2]

摘 要 从单机性能优化角度对一个高阶精度结构网格CFD并行程序进行了优化。通过识别关键变量并对其进行常量参数化优化,使编译器能够实现更高级别的针对性优化;根据程序数据结构特点及访问模式,设计了分级数据缓存技术,使程序主要计算代码能够以更优的方式访问主要数据结构,提高了访存空间局部性;进行了各种循环变换,以优化访存性能。在国家超算长沙中心"Tianhe-1A"并行机上的测试结果表明,相对于采用Intel编译器最高优化级别的版本,其对100万网格点二维翼型算例,串行程序性能提高约22.2%～28.9%;对1.12亿网格点三角翼算例,并行程序性能提高约13.9%～20.2%。

关键词 CFD并行计算,单机性能优化,关键变量参数化,分级数据缓存
中图法分类号 TP311 **文献标识码** A

Uniprocessor Performance Tuning of a Structured Grid Based Parallel CFD Application

CHE Yong-gang[1] ZHANG Li-lun[1] WANG Yong-xian[1] XU Chuan-fu[1] LIU Wei[1] WANG Zheng-hua[1] LIU Hua-yong[2]
(School of Computer, National University of Defense Technology, Changsha 410073, China)[1]
(State Key Laboratory of Aerodynamics, Mianyang 621000, China)[2]

Abstract This paper optimized the performance of a high order structure grid based parallel CFD (Computational Fluid Dynamics) application from a view of uniprocessor optimization. Performance critical variables were identified and transformed into constant parameters to enable compiler to apply specific high level optimizations. Multi-level data buffering was applied for the application's main data structures based on their structure and access characteristics, enabling the main computation codes to access these data more efficiently. Some loop transformations were applied to optimize the application's memory access performance. Performance evaluation was carried out on "Tianhe-1A" parallel computer installed at national super computer center in Changsha. Compared to the original code compiled by Intel compiler with the highest optimization level, the optimized code improves the serial performance for about 22.2%～28.9% for an 100 million grid points 2D aerofoil test case, and improves the parallel performance for about 13.9%～20.2% for an 112 million grid points delta aerofoil test case.

Keywords Parallel CFD, Uniprocessor performance tuning, Key variable parameterization, Multi-level data buffering

1 引言

计算流体动力学(Computational Fluid Dynamics,CFD)采用数值方法求解流体运动控制方程(如Euler方程或Navier-Stokes方程)来揭示各种流动规律,涉及流体力学、计算数学、计算机等多个学科。当前,CFD已广泛应用到飞机、火箭、飞船等航空航天器及汽车、列车等地面交通工具设计当中,成为型号设计的重要工具[1]。随着CFD技术的进步,所模拟的流场日益复杂,流动机理研究越来越精细,加上数值模拟精度要求的提高,对CFD软件计算能力的需求也极大地增长,CFD也发展成为典型的高性能数值计算应用领域。

高性能计算是高性能计算机系统、高效算法与高效程序实现相结合的产物。当前高性能计算领域应用程序实际持续性能与机器峰值性能的差距日益扩大,大多数实际应用程序的持续性能通常只达到机器峰值性能的5%～10%[2]。CFD计算领域同样如此。为提高CFD问题的求解效率,需要对CFD程序进行性能优化,以充分发挥机器系统硬件的潜力。并行优化是提高CFD并行程序性能的重要方面[3],但单机性能优化(Uniprocessor Performance Optimization)也不容忽视,并且从某种角度来说其更具挑战性[4]。

到稿日期:2012-10-29 返修日期:2012-12-25 本文受国家重点基础研究发展计划(973)课题(G2009CB723803),国家自然科学基金项目(11272352,61103014,60603055)资助。

车永刚(1973－),男,博士,副研究员,CCF高级会员,主要研究方向为CFD并行计算、程序性能优化,E-mail:ygche@nudt.edu.cn;**张理论**(1975－),男,博士,研究员,主要研究方向为CFD并行计算、并行算法;**王勇献**(1975－),男,博士,副研究员,主要研究方向为CFD并行计算、并行算法;**徐传福**(1980－),男,博士,助理研究员,主要研究方向为CFD并行计算、性能评测;**刘 巍**(1980－),男,博士,助理研究员,主要研究方向为CFD并行计算,并行算法;**王正华**(1962－),男,博士,教授,主要研究方向为CFD并行计算;**刘化勇**(1976－),男,博士,助理研究员,主要研究方向为CFD。

本文对一个高阶精度结构网格 CFD 应用程序进行了单机性能优化。该程序求解三维 Navier-Stokes 方程，集成了多种时间格式，空间格式采用较为复杂的高阶 Steger-Warming 迎风格式，具有很好的应用前景。程序使用 Fortran 90 语言编写，具有良好的模块化结构，并基于 MPI(Message-Passing Interface)进行了高效的并行化。本文通过识别关键变量并对其进行常量参数化，使编译器能做更高级别的针对性优化；通过分级数据缓存技术使程序主要代码可以以更高效的方式访问数据结构，提高了程序访存性能；进行了循环变换优化等；在高性能计算平台上进行了串行与并行性能测试，验证了本文优化技术的有效性。

2 关键变量的常量参数化优化

通过代码分析发现，程序中的两个变量非常关键，分别是原始变量个数 neq 和守恒变量的个数 npv，它们定义在一个模块 mod_fieldvars 中。其定义如下：

```
integer(kind_int) ::neq
integer(kind_int) ::npv
```

这两个变量之所以关键是有两个原因：一方面，这两个变量用于定义各个通量数据第一维的大小；另一方面，这两个变量还控制着很多最内层循环的迭代次数。例如在一些核心子程序中，主要数据定义和访问这些数据的循环嵌套语句的形式如下：

```
! 数据定义
type fld_array
  real(kind_real),pointer::r3d(:,:,:,:)
end type fld_array
type(fld_array),pointer ::pv(:)
real(kind_real)          ::pvi(1:npv),pvj(1:npv),pvk(1:npv)
! 访问这些数据的循环嵌套
! 外层 k、j、i do 循环
    do m=1,npv
  pvi(m)=pv(m)%r3d(il,j,k)
  pvj(m)=pv(m)%r3d(i,jl,k)
  pvk(m)=pv(m)%r3d(i,j,kl)
end do
! end 外层 k、j、i do 循环
```

进一步分析发现，这两个变量在程序中是不变的，在程序初始化过程中给其显式地赋值为一个常数。因此，可修改这两个变量的定义方式，即将其直接定义为常量参数，如下所示：

```
integer(kind_int),parameter::neq=5
integer(kind_int),parameter::npv=5
```

上述修改的好处是，使 neq、npv 成为编译时已知的常量，则编译器可以针对性地做各种优化。在 neq、npv 已定义为常量参数的情况下，核心子程序中各个相关数组第一维的大小是已知的，编译器还可以做针对性的访存性能优化，例如数组 Padding、软件预取等；此外，由于这两个常量较小，编译器可以针对相关最内层循环进行完全循环展开(full loop unrolling)，从而增加指令级并行性(Instruction Level Parallelism, ILP)，并减少循环语句执行的相关开销。通过查看优化前后的串行程序可执行代码(使用 Intel 编译器优化选项-fast)，发现优化后程序的可执行代码(串行版本)比原来增大约 44kB，可见编译器确实做了针对性的优化。

因为大多数子程序均通过模块引用了 neq、npv 这两个变量，上述方法可使大多数子程序得到针对性的编译优化。但是，某些子程序尚无法获得该优化方法的好处。例如，程序中的一个矩阵求逆子程序 matinv，其子程序定义及参数表如下：

```
subroutine matinv(n,a)
integer(kind_int),intent(in)::n
real(kind_real),intent(inout)::a(n,n)
```

其计算代码包括多个嵌套循环，形式如下：

```
do i=1,n
  do j=1,n
    ...
    a(i,j)=a(i,j)-a(i,k)*a(k,j)
    ...
    end do
end do
```

该子程序被调用处的形式如下：

```
call matinv(neq,matA)
```

可见，尽管在整个程序中每次调用该子程序时矩阵的阶均不变，但因为该子程序定义为通用的形式，在编译时编译器不知道 n 为常数，仍然无法做具有针对性的优化。针对这种情况，修改子程序定义及参数表如下：

```
subroutine matinv(a)
use mod_fieldvars,only:neq!从模块 mod_fieldvars 中引用常量参数 neq
    integer(kind_int),parameter::n=neq
    real(kind_real),intent(inout)::a(n,n)
```

这样就将 n 定义为了常数参数。该子程序的计算代码无需修改，而在该子程序调用处，将代码修改如下：

```
    call matinv(matA)
```

经过上述修改，编译器对 matinv 这类子程序也可以进行针对常量参数的专门优化。

3 基于分级数据缓存的存储优化

结构网格 CFD 计算中涉及的主要变量均是定义在网格坐标上的变量，其存储结构与访问方式对程序存储访问性能影响很大，因为现代处理器中普遍采用 Cache 存储层次，要求程序访存具有良好的空间与时间局部性，以提高 Cache 命中率。本文针对的结构网格 CFD 程序中，对主要数据而言，原来的存储方式是首先定义一个 3 维数组结构：

```
type fld_array
  real(kind_real),pointer::r3d(:,:,:,:)
end type fld_array
```

然后，对每种变量，定义一个 1 维指针形式的数组，每个数组元素指向一个上述 3 维数组结构。例如，针对一个结构网格块，守恒变量 q 的增量 dq 定义如下：

```
    type(fld_array),pointer::dq(:)
```

在上述数据结构下，dq(1)% r3d(i,j,k)，dq(2)% r3d(i,j,k)，dq(3)% r3d(i,j,k)，dq(4)% r3d(i,j,k)，dq(5)% r3d(i,j,k)分别存储坐标点(i,j,k)上守恒变量 $\rho,\rho U,\rho V,\rho W,\rho E$ 的增量。而程序中访问这些变量的代码形式如下：

```
! 外层 k、j、i do 循环
```

```
do m=1,npv
…
  …=dq(m)% r3d(i,j,k)…
…
end do
…
do m=1,npv
…
  dq(m)% r3d(i,j,k)=…
…
end do
! end外层 k、j、i do循环
```

可以看到，在上述嵌套循环中，数据访问看起来似乎使用连续的下标，但是最内层循环中相邻两次迭代所访问的数据项在主存中的存储位置距离相距 $i_{max}\times j_{max}\times k_{max}$ 个数组元素（其中 i_{max}、j_{max}、k_{max} 分别是数组 r3d(i,j,k)在三维上的大小）。因此，存储访问的空间局部性差，访存开销大。

为优化数据访问局部性，一种方式是将上述分层数据结构展开为平面的数据结构，例如可将上述变量 dq 修改为如下定义：

```
real(kind_real),pointer::dq_r4d(:,:,:,:)
```

其中，数据项 dq _r4d(1,i,j,k)，dq _r4d(2,i,j,k)，dq _r4d(3,i,j,k)，dq _r4d(4,i,j,k)，dq _r4d(5,i,j,k)分别存储坐标点 (i,j,k) 上守恒变量 $\rho,\rho U,\rho V,\rho W,\rho E$ 的增量。

我们注意到，文献[5]采用这种优化方式来优化程序访存局部性。但是对本文所研究的 CFD 程序来说，这种方式意味着需要对程序数据结构进行全面的修改，代码改动量很大，并且会影响程序对变量使用的灵活性，因为对一些变量进行不同顺序的访问不能通过简单的指针赋值来实现。因此本文采用了另一种折中的方法，即针对占用时间比例比较大的子程序（包括各个求解器子程序和右端项通量计算子程序），将其所访问的主要数据结构以 4 维数组的方式缓存起来，主要计算过程中使用缓存的数据来获得好的访存局部性。根据程序中主要数据使用的方式，使用了分级缓存的技术，其主要思想如下。

（1）一些数据（如网格导数 sxyz）只在程序初始化阶段计算一遍，而在主要求解过程（即时间步循环）中不变。对这样的数据，在程序初始化阶段将其缓存到新的 4 维数组数据结构中，在后续的时间步循环中使用缓存的数据而不让原始数据参与计算。在这种数据缓存方式中，数据缓存操作按照“拥有者缓存”原则进行，每个进程需要一次缓存分配给其计算的多个网格块所对应的数据。因此，用于缓存的数据结构中需要采用两层的方式，一层是指向各块的指针数组，另一层是实际存储各块数据的 4 维数组，并且需要记录每一块的全局块编号（在后面求解过程中，每个进程根据全局块编号与局部块编号的映射关系，确定缓存数据与原数据的对应关系）。例如，用于缓存 sxyz 的数据结构如下所示：

```
type dim4_fld_array
  integer(kind_int) ::nb !全局块编号
  real(kind_real),pointer::r4d(:,:,:,:) !块数据
end type dim4_fld_array
type(dim4_fld_array),pointer::buf_sxyz(:) !指向各块的指针数组
```

这种方式只需一次数据缓存操作，其时间开销相对于大量时间步循环计算来说可忽略不计。

（2）另一些数据（例如守恒变量、原始变量、右端项等）在求解循环的每一个时间步中都会被修改。对这样的数据，在使用它们的核心子程序中定义相应的 4 维临时变量，在进入子程序时将数据缓存在相应的 4 维数组临时变量中，子程序的计算代码使用临时变量中的数据进行计算，在退出子程序时使用临时变量中的值更新原变量的值。在这种数据缓存方式中，每个进程一次只需要缓存单块网格对应的数据，但是子程序每次调用时都要进行相应的临时存储空间分配、数据缓存和临时存储空间释放操作。这里用于缓存的数据结构与上述 4 维数组 dq_r4d 的形式相同。

4 其它优化

除上述优化之外，还进行了其它优化。例如，对部分嵌套循环中的语句进行了重排，以改善存储访问的空间局部性；对一些嵌套循环进行了循环交换，使得对数组数据的访问沿着其下标变化最快的维进行，以增加数据访问空间局部性。这些均是常规的优化，将其归在一类，这里不再详细介绍。

5 性能评测

在国家超算长沙中心（www. nscc-cs. com）“Tianhe-1A”并行计算机系统上进行了性能评测。该系统在 2012 年 6 月国际超级计算机排行榜（top500）上排名 28 位，系统共有 2048 个结点，每结点含 2 个 Intel X5670 6 核处理器（主频 2. 93GHz，每个核含 32kB L1 Instruction Cache 和 32kB L1 Data Cache、256kB L2 Cache，6 个核共享 12MB Last Level Cache（LLC）），每结点主存 48GB 以及 1 块 NVIDIA Tesla 2050 GPU 卡，结点间采用自主高速互连。所用的 MPI 库为 MPICH2 1. 2. 1p1-g16，编译器为 Intel FORTRAN Compiler 11. 1。经测试对比，在确保计算结果正确的情况下，采用编译器优化选项-fast 时的程序性能最高（对该程序的串行版本，-fast比缺省的编译器优化选项-O2 快约 10%），故对优化前后的串行/并行程序均采用-fast 优化选项进行编译，即本文优化代码比较的基准是采用 Intel Fortran 编译器选项-fast 进行高度优化的原程序版本。

5.1 针对串行程序的优化效果

为考察各个优化步骤的效果，对各个优化步骤得到的程序版本进行了串行性能测试。各个版本记录如下，列在后面的每个版本都是在前一版本基础上增加了新的优化：

- ORI：最初的未优化版本
- PARA：进行了关键变量常量参数化优化的版本
- PARA+OM1：对时间步迭代中不变数据结构进行缓存，在时间步迭代中多次使用的版本
- PARA+OM1+OM2：对时间步迭代中变化的数据，在相关子程序中使用临时缓存的版本
- PARA+OM1+OM2+OM3：进一步进行循环变换优化的版本

测试所用算例为绕二维翼型 NACA0012 的流动，单块结构网格共约 100 万个网格点。表 1 显示了不同版本的程序采用 3 种时间推进格式——非定常标量点松弛（ust_sca）、非定

常矩阵点松弛(ust_mat)和定常矩阵点松弛(std_mat)计算时,迭代100步的串行执行时间。

表1 各个版本的程序迭代100步的串行执行时间(单位:秒)

	ust_sca	ust_mat	std_mat
ORI	3351.1962	5760.7157	2624.5200
PARA	2928.6515	4751.6726	2018.4471
PARA+OM1	2817.0688	4535.7998	1922.1325
PARA+OM1+OM2	2643.1453	4349.3393	1898.5447
PARA+OM1+OM2+OM3	2606.1721	4268.6420	1865.8366

在表1数据的基础上,图1显示了各种优化技术对串行程序性能提升的比例。可以看到,关键变量的常量参数化优化获得的性能提升最大,对3种时间推进格式计算均超过12.5%,对定常矩阵点松弛格式的计算甚至达到22.4%,这说明只要为编译器提供程序的更精确信息,编译器就能够对程序进行更深入的针对性优化。其次是分级缓存优化,两级数据缓存加起来的优化效果均超过5.3%,这种优化与程序特定的数据结构及访问模式紧密相关,编译器很难自动进行优化。其它优化的效果在1.1%~1.4%之间,这些优化都是一些常规的循环变换,编译器在使用较高优化级别时,能够自动识别代码中的大部分此类情况并进行优化,因此手工实施这类优化的性能提升效果并不显著。

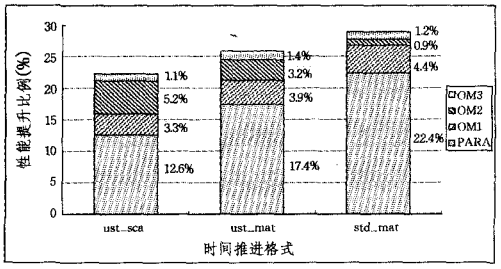

图1 针对串行程序的性能优化效果

为进一步考察性能优化的原因,利用Intel Vtune[6]进行了测试。Vtune基于CPU上专门的硬件性能监视单元(Performance Monitoring Units)进行测试,可获得微体系结构级的性能数据。由于在"Tianhe-1A"计算结点上采用Vtune进行性能测试存在一定不便,在一台具有相近微体系结构的台式机上进行了测试,该台式机的CPU为Intel Core i3 550(与Intel X5670同属Intel Nehalem微体系结构),主频3.2GHz,有2个CPU核,各含32kB L1 Instruction Cache和32kB L1 Data Cache、256kB L2 Cache,两个核共享4MB LLC,系统主存2GB。采用Intel Fortran Compiler 11.1.065编译器来编译程序,优化选项为-O3(Maximize Speed plus Higher Level Optimizations),测试算例同上。表2给出了串行程序原始版本(ORI)和优化后最终版本(OPT)使用定常矩阵点松弛格式计算时的性能数据。由于测试平台CPU与"Tianhe-1A"计算结点CPU不同,这些数据与"Tianhe-1A"上的数据不可能完全相等,但所反映的优化前后数据变化情况基本是一致的。

从表2可看出,优化使程序的总指令数、分支指令数、包含Load和Store的指令数都明显减少,分支预测错误数、数据TLB不命中数、L1和L2数据Cache等访问次数也明显减少,程序访存性能和指令级并行性均得到了明显的改善,这些是程序性能得到提升的主要原因。由于使用分级缓存优化,增加了主存使用量(对本算例,主存使用量增加不到5%),优化也增加了需要访问主存的Load指令数(增加约6.2%),但其开销与前述性能数据优化得到的收益相比小得多。

表2 基于Intel Vtune测得的性能数据

性能事件	ORI(10^6)	OPT(10^6)	OPT/ORI
INST_RETIRED.ANY	926960	764876	82.51%
BR_INST_EXEC.ANY	42238.4	23983.2	56.78%
BR_MISP_EXEC.ANY	266.24	216.48	81.31%
DTLB_MISSES.ANY	11123.2	4840	43.51%
L1D_ALL_REF.ANY	473680	391552	82.66%
L2_DATA_RQSTS.ANY	14332.8	9534.4	66.52%
L2_RQSTS.MISS	2686.4	2540	94.55%
MEM_INST_RETIRED.LOADS	385000	295688	76.80%
MEM_INST_RETIRED.STORES	138592	123000	88.75%
MEM_LOAD_RETIRED.L1D_HIT	371328	285864	76.98%
MEM_LOAD_RETIRED.L2_HIT	9396	5648.8	60.12%
MEM_UNCORE_RETIRED.LOCAL_DRAM	843.8	896.04	106.19%

5.2 针对并行程序的优化效果

为考察对并行程序的优化效果,在"Tianhe-1A"上对优化前后的并行程序进行了性能测试。测试所用算例为某三角翼,网格点数约1.12亿。因为"Tianhe-1A"上同时运行大量用户作业,为避免其它作业与本程序作业使用相同计算结点对测试数据的干扰,所有测试均采用独占结点的方式进行(在提交作业时指定结点数并使用exclusive选项),取迭代100步的执行时间。

图2给出了使用128、256和512个CPU核的情况下,本文优化技术对程序3种时间推进格式的性能提升效果。可以看到,性能提升幅度最低为13.9%,最高可达到20.2%,其中对后两种时间推进格式,性能优化的幅度均超过17.9%。因为本文单机优化主要提高了并行程序中计算代码的执行性能,而并行程序的通信性能并未提高,故对并行程序的性能提升比例比串行程序的低。但总体而言,这些优化对并行程序的加速效果还是比较明显的。

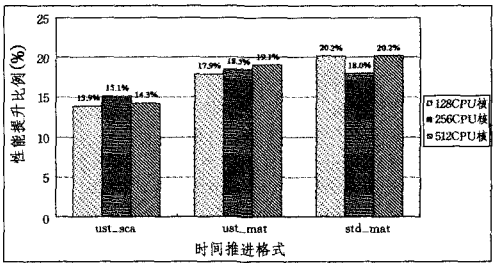

图2 针对并行程序的性能优化效果

并行测试时需要将网格剖分为多块,分配给不同的进程,网格剖分时并不能保证每个进程上的网格点数完全相等,并且一些进程上可能存在多块大小不一的网格,因此并行性能相对于串行性能来说会受到负载不均、数据规模变化使程序访存特性变化、通信开销等因素的影响,本文优化技术在并行计算情况下的性能提升比例关系不一定与串行计算情况下的完全相同。因此图2中,在256 CPU核时,优化对非定常矩阵点松弛格式的性能提升幅度(18.5%)高于对定常矩阵点松弛格式的性能提升幅度(18.0%),尽管串行计算时前者的性能提升幅度低于后者。

结束语 本文从单机性能的角度对一个具有良好应用前景的结构网格CFD应用程序进行了优化。通过关键变量的

常量参数化优化使能编译器做更深入的针对性优化，通过分级数据缓存技术提高程序访存性能，并进行了各种循环变换优化。在国家超算长沙中心"Tianhe-1A"并行计算机上进行了性能测试，结果表明，这些优化可有效提升程序的单机性能以及完整并行程序的性能，针对100万网格点二维翼型NACA0012算例，串行程序计算性能提高约22.2%～28.9%；针对1.12亿网格点三角翼算例，并行程序性能提高约13.9%～20.2%。这些优化保持了原程序的结构特点与灵活性，没有在代码中使用平台相关的编程技术或指令集，具有良好的可移植性。论文还通过微体系结构级的性能测试，分析了这些性能优化技术取得效果的内在原因。本文对于从事高性能计算应用软件开发与优化的科研人员具有一定的参考价值。

参 考 文 献

[1] Cosentino G B. Computational Fluid Dynamics Analysis Success Stories of X-plane Design to Flight Test [R]. NASA/TM-2008-214636

[2] Resch M M, Küster U. HPC Processor Technologies and Their Impact on Simulation[J]. Computational Science and High Performance Computing IV, Notes on Numerical Fluid Mechanics and Multidisciplinary Design, 2011, 115: 17-28

[3] 金君，乔楠，梁德旺. NAPA软件的并行优化[J]. 数值计算与计算机应用，2008，29(1)：65-72

[4] Gropp W D, Kaushik D K, Keyes D E, et al. Latency, Bandwidth, and Concurrent Issue Limitations in High-Performance CFD[C]//Proceedings of the First M. I. T. Conference on Computational Fluid and Solid Mechanics. Cambridge, MA, 2001

[5] 高瑞泽，于剑，阎超. 基于Cache友好方法的数值计算代码优化[J]. 计算机工程，2010，36(5)：7-9

[6] http://www.intel.com/software/products/vtune/ [OL]. 2012-05-30

(上接第106页)

蓝光"计算机系统上用NAS基准测试程序对应用性能进行了实测。"神威蓝光"计算节点配置为：每个节点4核，频率1GHz；内存4GB；InfiniBand QDR网络。表2列出了NAS测试包中消息比重较大的课题FT和LU在不同并行规模下的运行时间和消息内存开销情况。其中，测试课题规模为E规模；运行时间单位为s，内存为单个进程所需的队列和预投递接收缓冲内存开销，单位为MB。RC_SRQ为使用普通RC加SRQ的方式，XRC_S_G是使用XRC加SRQ且运用分组连接的方式，两者均使用8kB长度的接收缓冲区。

表2 性能与内存开销对比(时间单位为s，内存单位为MB)

模式/规模		2048进程		4096进程		8192进程		16384进程	
		时间	内存	时间	内存	时间	内存	时间	内存
FT	RC_SRQ	619.38	18.6	584.91	37.2	318.24	74.4	193.87	148.8
	XRC_S_G	620.21	4.65	585.06	9.3	325.52	9.315	198.68	9.344
LU	RC_SRQ	1111.1	18.6	548.95	37.2	311.62	74.4	216.4	148.8
	XRC_S_G	1111.5	4.65	549.65	9.3	316.40	9.315	220.23	9.344

从表2数据可以看出，分组连接与静态全连接相比会有一定的性能损失，这主要是因为在课题运行过程中需要临时握手、创建QP、建立连接等，这些操作都需要进入核心，带来了用户程序与核心之间的切换开销，但是我们认为，这种微小的性能损失与因空间需求的大幅减少而带来的可扩展性好处相比是值得的。

结束语 内存开销引起的可扩展性问题一直是困扰InfiniBand并行规模扩大的难题，尤其是多核系统的出现使得矛盾更加突出。本文首先介绍了基于InfiniBand消息传输服务层的两种可扩展技术：SRQ和XRC。通过这两种技术的结合，能够将单个进程所需的内存开销减少2个数量级，尤其是XRC技术在多核系统上的效果尤为明显。但是，SRQ和XRC并不能从根本上解决可扩展性问题，节点内存开销仍然会随着并行规模的增长而线性增加。

分组连接技术则是基于上层并行通信模型的可扩展性技术，它在静态全连接和动态连接之间进行了折中，使得其既具有静态全连接的性能优势，又具有动态连接的空间优势。采用分组连接技术之后，消息所带来的内存开销主要取决于组内进程个数，而不再随着并行规模线性增长，较好地解决了可扩展性问题。除此之外，分组连接还可以根据网络拓扑和应用课题的通信特征灵活配置，做到在性能和内存开销上的最佳平衡。

当然，解决可扩展性问题的技术不仅仅只有本文介绍的这些，比如MPI[4,5]有基于InfiniBand UD传输服务实现的版本，UD传输服务支持一个QP与多个QP通信而不需要建立连接，这些同样可以解决可扩展性问题。但是UD是不可靠的，且不支持RDMA操作，消息大小受限于网络MTU(Maximum Transmission Unit)[2]，所以在大规模科学计算中应用较少。本文介绍的3种技术均基于RC传输服务，其最大并行规模在"神威蓝光"计算机系统上实测已经达到了数十万核，较好地解决了InfiniBand消息可扩展性问题。

参 考 文 献

[1] InfiniBand Trade Association. InfiniBand Architecture Specification[OL]. http://www.infinibandta.com

[2] Mellanox Technologies. Connectx Family Programmer's Reference Manual[OL]. http://www.mellanox.com/docs/

[3] Koop M J, Sridhar J K, Panda D K. Scalable MPI Design over InfiniBand using eXtended Reliable Connection[C]//IEEE International Conference on Cluster Computing. 2008: 2-4

[4] Yu Wei-kuan, Gao Qi, Panda D K. Adaptive Connection Management for Scalable MPI over InfiniBand[OL]. http://cse.ohio-state.edu, 2006-02-08

[5] Friedley A, Hoefler T, Leininger M L, et al. Scalable High Performance Message Passing over InfiniBand for Open MPI[OL]. http://cs.indiana.edu, 2007-02-07