基于 MPI 和 OpenMP 的多核高性能计算并行系统

Haoqiang Jin,Dennis Jespersen,Piyush Mehrotra, Rupak Biswas
Lei Huang,Barbara Chapman

摘要

现代微处理器核心数量的迅速增加正在推动电流高性能计算系统到千万亿次和百亿亿次的时代。混合这些系统的性质，跨节点的分布式内存和共享内存与非统一内存访问，这对应用程序开发人员提出了挑战。在本文中，我们研究了一种混合的方法来编写这样的系统，组合两种传统的编程模型，MPI 和 OpenMP。我们目前从多区域行基准和两个完整的应用标准基准使用此方法在多核心系统中，包括 SGI 的 Altix 4700， IBM P575 +和 SGI 的 Altix ICE 8200ex。我们也提出了新的数据局部性扩展使得 OpenMP 更好地匹配多核体系结构的层次存储结构。

由 Elsevier B.V 出版

关键字: 混合编程，多核系统，OpenMP 扩展，数据局部性

1.介绍

多计算核心在商业微处理器芯片正普遍存在，当今芯片制造商实现了高聚合性能，同时避免功率需求增加时钟速度。我们期待核心的数量满足穆尔定律，每个晶体管的计数随着晶体管计数的增加而增加。由系统上的证明 TOP500 列表 [30]。如今，高端计算系统厂商，充分利用规模经济的优势可以将部分芯片打造成为大型 HPC 系统。这些系统的基本构建块由多个多核心芯片单节点共享内存组成。占主导地位的 HPC 架构，目前和可预见的未来，包括聚类这样的节点通过高速网络支持异构存储共享存储器互连参数模型与非统一内存访问（NUMA）在一个单节点和分布式存储节点。

从用户的角度来看，编写这些系统程序的最方便方法是采用忽略混合存储器差异的一种纯消息传递编程模型。对于大多数系统这是一个合理的策略，因为大多数消息传递接口库开发人员利用节点内的共享内存来优化节点内通信。因此，用户为他们的应用程序获得了良好的性能，同时使用一种单一的标准化规划模型。然而，有其他方法来编程这样的系统。这些包括新的语言作为开发的 DARPA 的高生产率计算系统程序，这是试图提供一个非常高水平的编程方法和百亿亿次系统计算。由于这些语言在最近几年才刚刚提出，它们的实现都处于萌芽阶段，很难预测它们的最终效用和性能。

另一种方法是采用分区全局地址空间实现的（PGA）语言，尝试结合最好的有限元分析共享内存编程模型和消息传递模型的应用。编程语言，如共同阵列 fortran，UPC 和利用让用户通过指定数据分布这种编程模式，控制数据的存储位置来提高可编程性，允许他们使用全局名称空间与远程数据来明确规格访问。

虽然这些语言提供了一个渐进的方法来编程 HPC 系统，研究表明，用户必须仔细使用语言功能，以获得等效 MPI 程序的性能。

在本文中，我们专注于采用一种混合的方法来编写基于多核的 HPC 系统的程序，结合标准化编程模型 MPI 的分布式内存系统和 OpenMP 的共享内存系统。虽然这种方法已经被许多用户采用以实现他们的应用程序，我们使用一些标准的基准测试方法来研究性能特点，多区域并行基准测试和使用由美国宇航局的工程师们采用的两个完整的应用程序。我们在几个平台测量这些代码的各种混合配置的性能。包括 SGI 的 Altix 4700（英特尔的 Itanium 芯片），IBM P575 +（IBM Power5 芯片）和 SGI 的 Altix ICE 8200ex（英特尔至强 Nehalem-EP 芯片）。

本文还介绍了一种新的方法，在该方法中，我们扩展 OpenMP 模型与采用新的数据局部扩展使得在现代 HPC 系统中更好地匹配更复杂的内存子系统。这个想法是允许用户使用简单的共享内存模型的 OpenMP 同时提供的”hints“的形式，数据局部性的编译器编译和运行时系统，它们有效地针对底层体系结构的层次存储子系统。

本文的结构如下。在下一节中，我们简要介绍了 MPI 和 OpenMP 模型和本文研究的混合编程方法。第 3 节介绍了多区域 NAS 并行基准描述的这两个应用程序溢出问题。第 4 节开始描述并行系统的研究和使用，讨论的性能结果，并得到的结论，简要总结我们的研究结果。在第 5 节中，我们提出了扩展到 OpenMP 描述的语法和语义的各种指令。第 6 节和第 7 节描述了一些相关的工作和我们的论文的结论。

2.MPI 与 OpenMP 编程模型

MPI 和 OpenMP 是两个主要的编程模型，已在高性能计算领域中被广泛使用了多年。在两个模型的各个方面中已经有许多这样努力研究的论文，从可编程到性能。我们的研究重点是基于混合两种编程模式的方法，这在现代多核并行系统中更重要。首先，我们将简要介绍每个模型，以方便我们研究这种混合编程方法。

2.1 为什么选择 MPI

MPI 是分布式存储系统上并行编程的事实标准。最重要的优点是这种模式有双重的优势：高效的性能和可移植性。好的性能是优化 MPI 的直接结果，程序开发周期中的库和完整用户控件。可移植性来自标准 API 和存在于广泛的机器上的 MPI 库。在一般情况下，MPI 程序可以运行在分布式和共享内存机器上。

MPI 模型的主要难点是它在编程中的离散内存视图，这使得它很难开发，而且往往需要一个很长的开发周期。一个好的 MPI 程序需要仔细思考策略来实现数据和管理产生的通信。其次，由于模型的分布式性质，由于性能原因，数据可能会被复制，从而导致总体内存需求的增加。第三，在大型并行系统上运行非常大的 MPI 程序具有非常大的挑战，由于机器的不稳定性和缺乏故障模型中的公差（尽管在其他编程模型中是这样的）。此外，大型 MPI 工作往往需要大量的资源，如内存和网络。

最后，大多数的 MPI 库有一个重要的运行参数需要“被调用”才能达到最优性能。虽然供应商通常提供一组默认的参数，普通用户掌握它的语义和这些参数

在特定目标系统上的最佳使用是不容易。有一个隐藏的成本，由于底层库的实现往往被忽视 作为一个例子，表 1 列出了时间测量的 cart3d 应用在英特尔 Xeon e5472（四核）两种型号内存节点的节点。两参数，mpi_bufs_per_host 和 mpi_bufs_per_proc，控制多少内部缓冲空间分配的通信 MPI 库。结果表明，从不同的 MPI 缓冲区大小的性能（超过 50%）的影响很大。两失败的情况下，较小的内存（8 GB）的节点是由于内存耗尽，在这些节点，但从不同的原因：在 32 个进程中，应用程序本身需要更多的内存；在 256 个进程中，内部 MPI 缓冲区的内存使用过多。（这可能会增加作为一个二次函数的进程数，正如我们在第 4 节发现的）。

Table 1
CART3D timing (seconds) on a Xeon cluster.

Number of processes	8 GB node	16 GB node
32	*failed*	297.32
64	298.35	162.88
128	124.16	83.97
256	*failed*	42.62
MPI_BUFS_PER_HOST	48	256
MPI_BUFS_PER_PROC	32	128

2.2 为什么选择 openMP

OpenMP 是基于编译器指令和一组支持库调用的基础上实现的库，是一种便携式的并行程序共享内存系统。OpenMP 2.5 规范（及其早期版本）侧重于循环级并行性。随着 OpenMP 3 规范中任务的引入，语言变得更加动态。OpenMP 协议几乎可以在所有主要编译器厂商中移植。OpenMP 的著名优势是它采用应用程序的全局视图内存，允许并行应用程序相对快速开发的地址空间。基于指令的方法使它成为可能为了便于维护，编写顺序一致的代码。OpenMP 提供了对并行增量方法，在并行代码开发周期中优于 MPI。

2.3 MPI + OpenMP 方法

由于集群的共享内存节点成为占主导地位的并行体系结构，这时很自然的考虑混合 MPI 方法。混合模型符合分层机制模型，其中 MPI 用于在分布式存储节点通信，OpenMP 用于在节点中细粒度的并行化。结合两者的优点，混合模型采取 MPI 过程需要在一个节点中，能保持与 MPI 的交叉节点的性能和数量减少，因此，相关的消耗（例如消息缓冲区）。我们稍后在 SEC 讨论，混合的方法可以帮助减少对资源的需求（如内存和网络），这可能是在运行非常大的工作中十分重要的。对于某些类别的应用程序，易于开发的多级并行混合模型可能会降低应用程序开发的工作量。在下面的章节中，我们将研究上述优点和潜在的缺点，在一些案例中研究混合方法。

3.混合编程案例研究

本节简要介绍使用混合 MPI +openMP 实现的应用程序的特性,两个假设的应用和两个真实世界的应用。下一节将侧重于这些应用程序在几个并行系统中的演示。

3.1 多区域 NAS 并行基准

NAS 并行基准多区版本是来自假设应用基准。包括在常规 NPBs。他们模仿许多现实世界的多块代码，其中包含可利用的并行性多层次。由于物理结构的问题，它是使用 MPI 粗粒度并行。用于区域内细粒度并行的 OpenMP。有三个基准问题的定义 npb-mz，在表 2 中总结，与问题大小不等的类的（最小）到 F 级（最大）。两 sp-mz 和 lu-mz 有对于一个给定的问题类和负载平衡在这种情

况下，固定大小的区域是简单而直接。另一方面,在 bt-mz 给定类的区的大小可以通过多达 20 个因素的变化，提高了负载平衡问题。由于固定区域 lu-mz 号，这个基准利用分区并行性是有限的。

Table 2
Three benchmark problems in NPB-MZ.

Benchmark	Number of zones	Zonal size
BT-MZ	Increases with problem size	Varies within a class
SP-MZ	Increases with problem size	Fixed within a class
LU-MZ	Fixed (=16)	Fixed within a class

混合 MPI + OpenMP 实现 npb-mz 使用 MPI 的区域间并行和装箱算法(不同地区间的负载平衡算法)，有没有进一步的区域分解为每个区域。用于区域内环路的 OpenMP 并行。有关实现的详细信息，请参见【15】。由于有限的区域数（= 16）在 lu-mz，这限制了 MPI 进程数，我们没有在这项研究中使用基准。

3.2 OVERFLOW

OVERFLOW 是一个通用的纳维-斯托克斯解算器计算流体动力学（CFD）问题【20】该代码是专注于高保眞粘性模拟围绕现实的航空航天配置。有用的几何区域是分解成一个或多个曲线，但逻辑笛卡尔网格。任意重叠的网格是允许的。这个代码使用有限差分空间，隐式时间步长，明确的信息交换重叠边界。对于并行化，网格分组和分配到 MPI 过程。对于负载平衡的目的，每个网格可能是毛皮分解成更小的域。在解决方案过程中这些较小的域作为逻辑独立网格。溢出的混合分解涉及 OpenMP 并行 MPI 并行以下。所有 MPI 行列有相同数量的 OpenMP 线程。OpenMP 共享内存并行是在相当细粒度域内。

在 OVERFLOW 的数值方案涉及的流量求解步骤在每个网格，然后由一个明确的交换界——元素重叠信息。对于一个给定的情况下，当 MPI 进程的数目增加，更多的网格分裂通常发生是为了获得良好的负载平衡。这个策略有几个缺点。首先，更多的网格分裂的结果在一个数值算法有一个明确的味道（无限的网格分割域大小可能缩小到只有少数每个网格点。这减少了含蓄会导致收敛速度慢（甚至不收敛）。慢展——收敛意味着更多的迭代（从而不再时钟时间）才能达到理想的解决方案。这是图 1 所示的翼型试验用例，示出的效果的收敛速度分裂成一个单一的网格为 120 区。没有网格分裂，计算的 L2 剩余单调减少迭代收益。对于分裂的情况下，残余似乎是不收敛的。这无疑是一个非常不切实际的例子，但对于现实的情况下，POS—在收敛行为由于网格分裂可能改变可以增加不确定性的程度可能已经不需要什么是一个困难的解决过程。

其次，更多的网格分裂导致网点数量的增加，因为“'ghost”或“'halo”点用影响分割边界处的数据插值。表 3 显示了不同数量的网格点的总数从 23 个区域和 3600 万个网格点的数据集产生的域组。对于 256 组，总网目尺寸为增长 30%。由于流码解的时间本质上与网格点的总数成正比，在网格点折痕部分抵消提高并行效率。

混合方法的结果是在少数网格分裂，它提供了双重好处 :有更少的总网格点，和收敛行为不太可能受到不利影响。这是由于这样的事实，这两个属性依赖于 MPI 行列总数，而不是处理器核心总数。混合与 MPI 并行化的影响隐式方法的收敛速度也被 Kaushik 和他的同事们[17]指出。

3.3. Akie

Akie 是透平机械中的应用，是用于转子叶片的三维流动不稳定性研究[11]。多块配置包括求解偏微分方程周围的每个刀片和插值解决方案—吐温叶片。并行可以利用在刀片间和刀片内的水平。对于性能原因，代码使用 32 位数值算法。我们开始与 Akie 顺序版本开发混合 MPI+OpenMP 版本（P1h）。因为这个版本并没有达到预期的可扩展性（特别是在 Altix 4700 以下的证券交易委员会的讨论，我们开发了一个纯 MPI 版本（P2D）以及。这两个版本的基础上一个层次的两级并行化 AP—方法的差异仅在第二级并行完成，如图 2 所示。第一级并行应用域分解的叶片尺寸。每个 MPI 进程（P0 PN）一组作品在每次迭代结束时，叶片和交换边界数据。第一级的并行性仅限于叶片可在给定的问题。为了利用额外的并行性，将需要切片的第二级并行化。

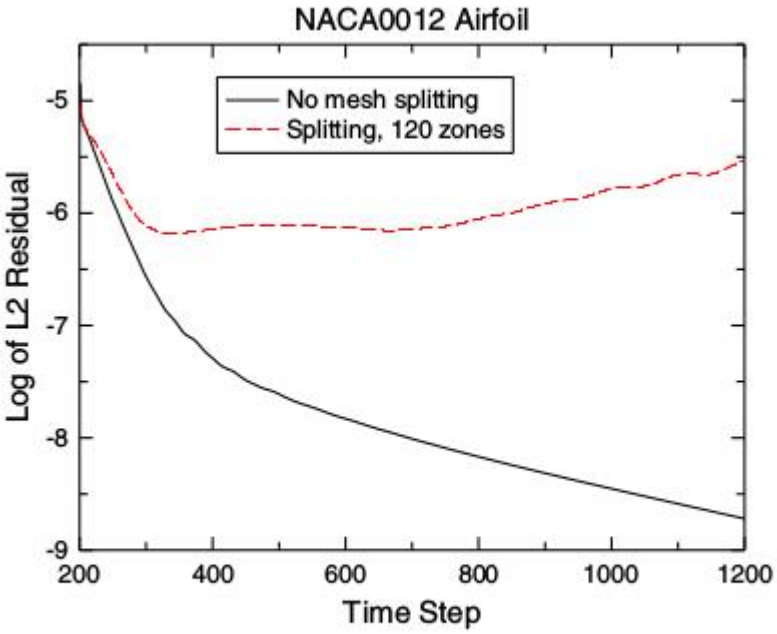

Fig. 1. Numerical convergence rate as a result of mesh splitting.

Table 3
OVERFLOW mesh size for different numbers of domain groups.

Number of groups	Total number of mesh points (M)
16	37
32	38
64	41
128	43
256	47

第二级并行，混合动力版本（P1h）使用 OpenMP 并行区域循环（主要应用于叶片内三维空间的 J 维数）。线程同步是用来避免在不同地区的执行条件。相比之下，纯 MPI 版本（P2D）将进一步分解（对 J 维）在一个 MPI 的叶片分配给每个子域的过程。因此，每个第一级 MPI 组由子级 MPI 进程工作在子域并在组内执行附加通信。与没有变化的算法相比，OpenMP 的方法，也就是说，这两个版本保持相同的数值精度。由于这些原因，这是一个很好的例子来比较 MPI 和混合方法。我们应该指出，无论是混合动力和 MPI 版本实现—与来自同一个代码库。在纯 MPI 版本实际执行（P2D）、二维布局的 MPI 通信组用于匹配的数据分解。在两个层次上的 MPI 过程中，与没有代码的动态进程创建使用相比。

4.性能的研究

我们研究的混合 MPI + OpenMP 的应用程序的性能分析在位于美国宇航局高级超级计算机（NAS）部门中进行了三多核并行系统分析，NASA 艾姆斯研究中心。我们首先简要地并行分析，然后比较混合应用程序的性能。在整个论文的其余部分,我们使用术语“核”和“'cpu”互换。

4.1 并行系统

表 4 总结了这项研究所使用的三个并行系统的主要特点。第一个并行系统，SGI Altix 4700，是哥伦比亚超星系团的一部分，由 512 个计算刀片。每个刀片主机两个双核英特尔 Itanium2 处理器共享相同的本地存储器板通过总线。该系统具有高速缓存相干非统一内存访问（ccNUMA）之间的通信架构的互连 numalink4 支持叶片，系统中的 2048 个核心的全局可寻址空间。第二系统，IBM P575 +，是一簇 IBM POWER5 +节点与 IBM 的高性能开关（HPS）。每个节点包含 8 个双核的 IBM POWER5 +处理器，并提供相对平坦的访问节点的内存从每个 16 个节点中的核心作为一个圆的结果内存页面放置策略。每个 Power5 处理器能同时多线程（SMT）。第三并行系统，SGI 的 Altix ICE 8200ex，是新扩建的昴星团[24]部分。昴星团包括基于节点三种：四核英特尔至强处理器 e5472（四核），四核 x5570（Nehalem-EP），和六个核心 x5670（Westmere）。所有的节点都在一个立方体的拓扑结构 InfiniBand 互联。在这项研究中，我们只使用基于 Nehalem 的节点。这些节点使用两个四核英特尔 x5570（Nehalem-EP）处理器。每个 Nehalem 处理器包含四核心与超线程（HT，类似 SMT）的能力，使每个核心的两个线程。一个片上内存控制器直接连接到本地的 DDR3 内存，使内存的吞吐量相比大幅到上一代英特尔处理器。然而，访问内存在 Nehalem 的节点不均匀。SGI 的 Altix 系统都有第一次接触内存布局策略为默认。

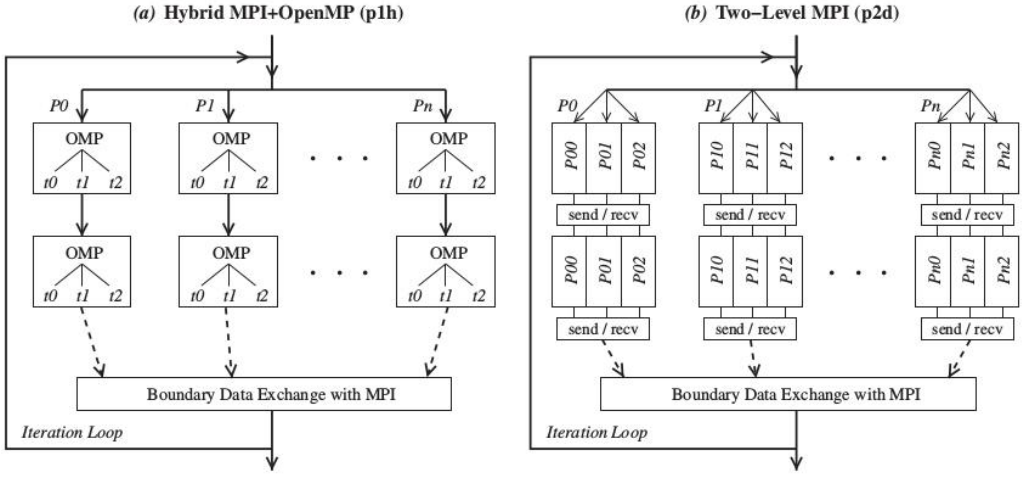

Fig. 2. Schematic comparison of the AKIE (a) hybrid and (b) MPI codes.

Table 4
Summary of the parallel systems.

System name	Columbia22	Schirra	Pleiades (Nehalem)
System type	SGI Altix 4700	IBM p575+	SGI Altix ICE 8200EX
Processor type	Intel Itanium2	IBM Power5+	Intel Xeon
Model	9040		X5570
Series (# Cores)	Montecito (2)	(2)	Nehalem-EP (4)
Processor speed	1.6 GHz	1.9 GHz	2.93 GHz
HT or SMT (Threads/Core)	–	2	2
L2 Cache (cores sharing)	256 KB (1)	1.92 MB (2)	256 KB (1)
L3 Cache (cores sharing)	9 MB (1)	36 MB (2)	8 MB (4)
Memory interface	FSB	On-Chip	On-Chip
FSB speed	667 MHz	–	–
No. of hosts	1	40	1280
Cores/host	2048	16	8
Total cores	2048	640	10240
Memory/host	4096 GB	32 GB	24 GB
Interconnect	NUMALink4	IBM HPS	InfiniBand
Topology	FatTree	FatTree	Hypercube
Operating system	SLES 10.2	AIX 5.3	SLES 10.2
Compiler	Intel-10.1	IBM XLF-10.1	Intel-11.1
MPI library	SGI MPT-1.9	IBM POE-4.3	SGI MPT-1.25

在软件方面，无论是 MPI 还是 OpenMP 都可在 SGI 的 Altix 4700 通过 SGI MPT 库和 COM—编译器支持。这两个集群系统中，MPI，由 SGI MPT 或 IBM 坡库的支持下，可供通信—节点之间的通信，以及 MPI 与 OpenMP 可以在每个节点使用（见表 4）。对于我们所有的实验研究中，我们充分填充了每个节点的核心资源 例如，运行在 SGI Altix ICE 系统中的应用 32 MPI 进程和 4 OpenMP 线程每 MPI 过程中，我们分配的工作和分配 2 个节点的 MPI 过程的 16 个节点每个节点上每个节点总共有 8 个线程。为了确保一致的性能结果，我们应用的过程/线程绑定工具可在每个并行系统上绑定进程和线程到处理器内核。

4.2 多区域 NAS 并行基准

在 3.1 节中提到的，由于区域 lu-mz 数量有限，限制了 MPI 进程数， 我们只讨论在这段 bt-mz 和 sp-mz 性能。在进入演示细节之前，我们第一次测试不同进程和线程组合的基准应用程序的内存使用情况。

测量内存的使用 sp-mz C 类的 Altix ICE 是图 3 所示：每个过程为 MPI 支持功能在左边的面板，每核心在右面板功能 OpenMP 线程（B）。进程总数在每个实验中使用的 ING 核心等于 MPI 进程的数量的 MPI 线程的数量。测量结果从高水位标记内存使用在每次运行结束时的校正用于在节点进程间使用共享内存段。为了比较，数据所使用的实际内存数组中的应用也包括在图中。

一个 Altix ICE 节点内（最多 8 核心，图 3（a）），内存使用情况大致反映了应用数据的需要。当 MPI 进程的数量增加，每个进程的数据大小减少，但实际内存的使用仅减少到次点（128 芯），然后增加几乎二次。内存的增加主要是由于增加的 MPI buf—分配在底层的 MPI 库节点之间的通信提供了集体（在这种情况下，SGI MPT），依赖于使用的节点数。在混合模式，对于给定数量的内核，我们看到在内存使用上大幅减少，特别是在大的核心计数（见图 3（b））。例如 256 个核心，256 的 1 运行（类似纯 MPI）使用 11.9 GB（= 256 MB 的 46.3 MB）的内存，而 32 的 8（混合）运行只使用 5.5 GB（MEM = 21.5 MB）的 MEM—论。这种大的内存节省可能在未来多核系统是非常重要，每个核心内存少。这个数字还显示了内存使用的突然下降超过 64 个节点（512 个核心）。这对应于 MPT，SGI 的 MPI 库，切换到一个最佳的通信方案，分配消息缓冲区空间少并可能在某些应用程序通信造成性能下降（如 cart3d 案例描述在第 2 节）。虽然使用不同的数据数组大小的内存配置，bt-mz（不显示）非常相

似这 sp-mz。D 类的内存使用趋势（这里也没有显示）与 C 类非常相似。

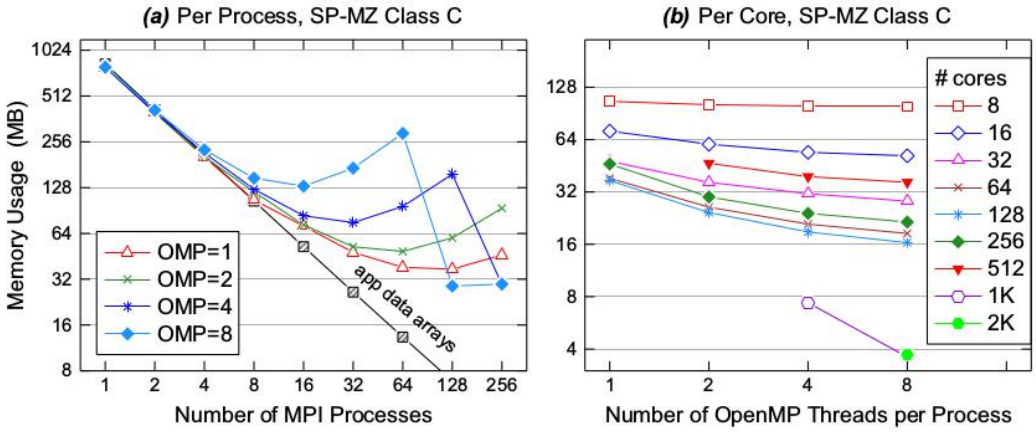

Fig. 3. SP-MZ Class C memory usage on the Altix ICE-Nehalem.

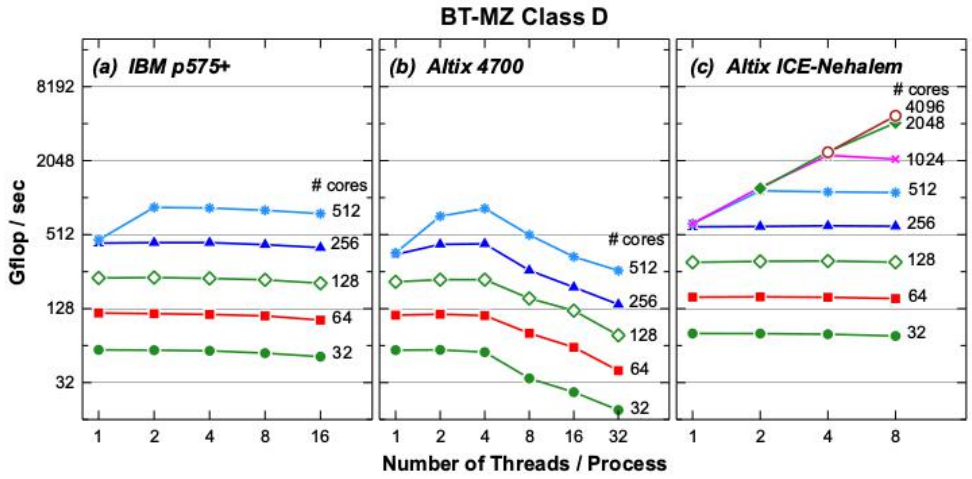

Fig. 4. BT-MZ performance (Gflop/s) on three parallel systems.

4 和 5 显示性能（亿次/秒，高的更好）的 bt-mz 和 sp-mz D 类三个并行系统。每个曲线对应于给定数量的内核使用，同时改变 OpenMP 线程的数量。对于给定的核心数，从左到右，MPI 进程的数量减少，而每个 MPI 进程的 OpenMP 线程的数量从 1 到 16 的折痕在 IBM P575 +，1 到 32 的 Altix 4700，和 1 至 8 的 Altix ICE Nehalem。D 类数据集共有 1024 个区。因此，MPI 只能使用多达 1024 个过程和超越，混合的方法是必要的。对于 bt-mz，MPI 尺度好

最多 256 个进程的所有三个系统，但除此之外，没有进一步的改善是由于负载不平衡的结果，不均匀的大小区域的基准（见图 4）。然而，进一步的业绩提升可以利用 OpenMP 线程提高大芯数负载平衡。对于一个给定的核心数，bt-mz 显示关系不同进程的线程组合只要 MPI 进程负载均衡相对平坦的性能（高达 256 过程）对 IBM P575 +和 Altix ICE Nehalem。然而，在 Altix 4700 的性能下降的很快超过 4 的 OpenMP 线程时叶片是涉及 NUMA 内存。如 4.1 节所讨论的 Altix 4700 刀片包含 4 位的内核通过总线共享相同的本地存储器板。访问内存在另一个刀片更多 4 多芯必须经过 SGI numalink，具有较长的潜伏期。如图 4 所示的性能（B）是某长内存延迟对系统的远程内存访问相关。

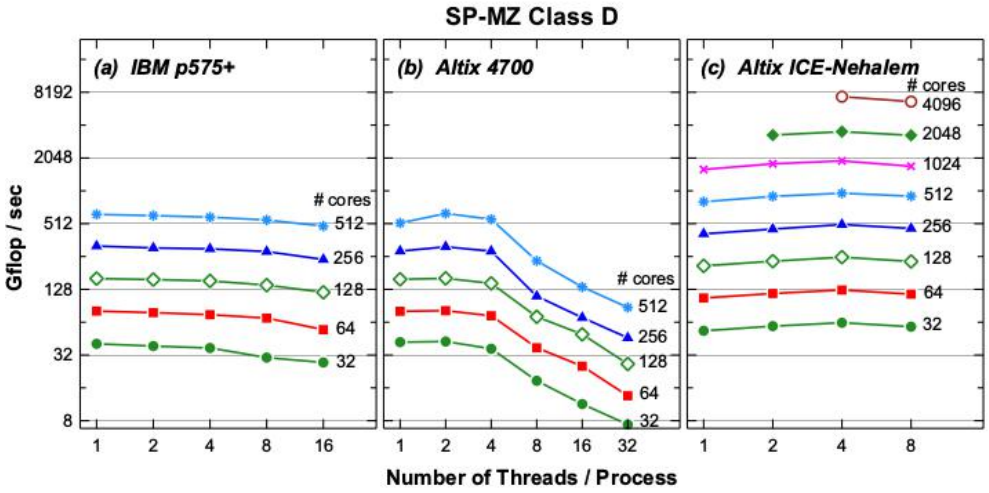

Fig. 5. SP-MZ performance (Gflop/s) on three parallel systems.

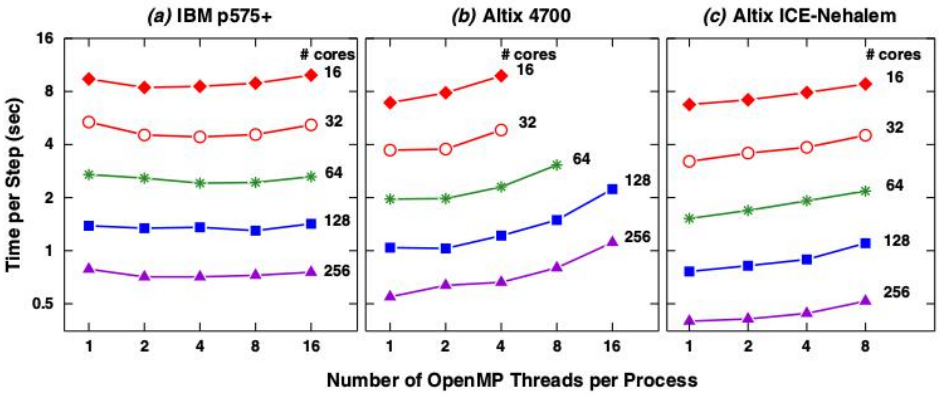

Fig. 6. OVERFLOW2 performance on three parallel systems. Numbers in the graphs indicate the number of cores used.

sp-mz 尺度非常好，高达 1024 的 MPI 进程的 Altix ICE Nehalem（见图 5（c））。然而，有表现使用不同数量的 OpenMP 线程对于给定的核心数的差异。获得最佳性能时每个进程的 OpenMP 线程数为 4。这种现象需要一些解释。首先，使用更多的 OpenMP 线程减少了相应数量的 MPI 进程。其结果是，消息总数减少消息大小的增加，从而导致更好的通信性能，并提高了整体性能。其次，通过对 sp-mz OpenMP 线程分配到 MPI 进程区相同的 MPI 进程的工作了在一起，一个时区一次。使用更多的 OpenMP 线程有效地增加总的缓存大小相同的数据和提高性能（假设一个 OpenMP 线程分配到一个核心）。然而，由于数量 OpenMP 线程增加，与 OpenMP 构建和远程存储器访问套接字的开销相关在节点增加。从这些因素的组合，我们看到一个相对性能提高到 4 线程后我们看到 8 线程的性能下降。

对于一个给定的核心数 IBM P575 +，我们看到 sp-mz 持久性能下降，OpenMP 的数量线程的增加。我们认为这是结果，事实上一个节点的内存页 P575 +安置循环（不是第一次接触）但 sp-mz 的 OpenMP 并行化呈第一触摸记忆位置。这样的不匹配在不同的 OpenMP 线程的数据访问导致较长的延迟。因此，我们看不到数据的有效增长缓存作为观察的 Altix ICE Nehalem。相反，

在 sp-mz Altix 4700 的性能是相对平坦的达 4OpenMP 线程，然后迅速下降，这表明系统严重的 NUMA 效果。

4.3 OVERFLOW

在我们的实验中，我们采用 OVERFLOW 2.0aa 在现实的机翼/机身/发动机短舱/塔结构与 23 区 3590 万目。图 6 显示了混合码的性能在每秒钟的时间步长（较低更好）MEA—分别在 IBM P575 +，Altix 4700，和 Altix ICE Nehalem 采用不同工艺螺纹组合。OpenMP 线程每 MPI 进程数从 1 变化到 8 的 Altix ICE 和另两系统达 16。数据点对于纯 MPI 版本在所有三个系统是非常接近的混合版本与 1 OpenMP 线程绘制在图中。在 IBM p575 +多个 OpenMP 线程混合动力版本优于纯 MPI 版本。这个使用 4 个 OpenMP 线程观察到最好的结果。从第 3.2 节回忆起，随着 MPI 进程的增加，溢出更多的网格分裂，导致更多的网格点。由于 OpenMP 线程的数目增加为固定总数量的核心，使用更少的 MPI 进程和更快的执行时间可能会预计，因为总数网格点减小。在同一时间，与 OpenMP 并行化的开销增加，其中有一个负性能影响。从两个效应的平衡点取决于应用程序和系统特性。观察到的好处从混合动力版本的 IBM 系统不能对两 Altix 系统。

相反，我们观察随着 OpenMP 线程数量的增加，时间单调增加。这在运行时增加主要是造成通过增加访问远程内存在 ccNUMA 架构 OpenMP，而黯然失色任何减少在运行时从少量的工作。为了提高性能在 NUMA 系统 OpenMP 是我的钥匙证明数据局部性，减少远程存储器访问。相反，在 IBM p575 +无明显的 NUMA 效果，这可以归因于系统的轮转存储页面放置策略。

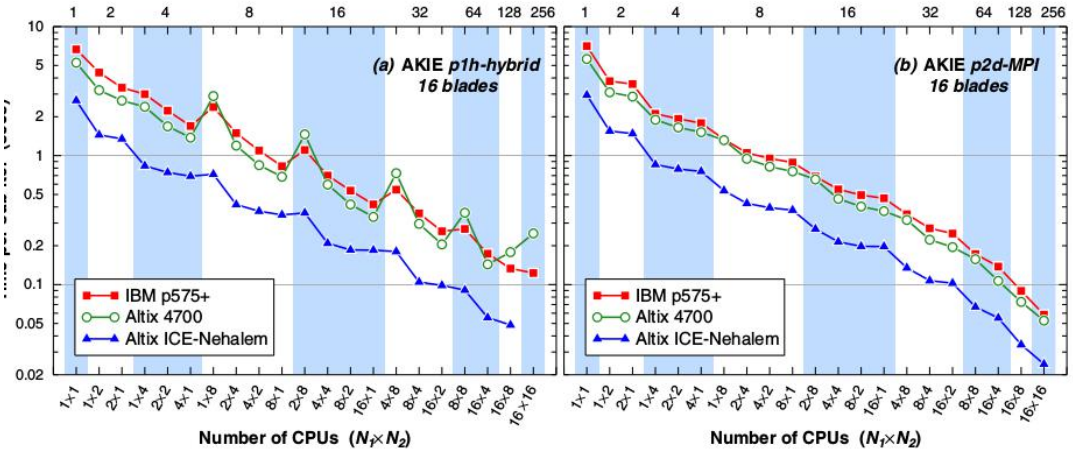

Fig. 7. Timings of the AKIE (a) hybrid and (b) MPI codes on three parallel systems.

4.4. Akie

我们使用了一个 16 叶片的情况下比较混合和 MPI 的 Akie 代码在三版本的性能系统。每个进程线程组合运行的每次迭代时间如图 7 所示（较低为佳）。在 n 1 N 2 符号，N 1 表示在第一级的 MPI 进程的数目，和 n 2 表示 OpenMP 线程的数目（混合版本）或 MPI 进程（MPI 版本）在第一级 MPI 进程的第二个级别。总使用的核心数，N 1 N 2，从 1 增加到 256。N 1 的值是有限的叶片的数目，这是在目前的情况。n 2 的值主要是由一个给定的问题的分区尺寸约束。对于混合的情况下，OpenMP 线程 N 2 是数量也有限的 SMP 节点的大小（16 在 IBM P575 + 8 的 Altix ICE，2048 Altix 4700）。例如，我们不可能跑 16 16 混合动力版的 Altix ICE，但没有对于 MPI 版本这样的限制，因为 N 2 MPI 进程可以跨越多个节点，如果需要。总体而言，我们观察到类似—在 IBM P575 +和 Altix 4700 的性能。基于 SGI 的 Altix ICE 产生 2–性能高 3 倍的 Nehalem 比其他两个系统。首先，这部分是由于更快的处理器速度在系统中。二、使用 32 位在应用数值算法有较大的性能提升（从 64 位数值算法）在 Nehalem 处理器比 POWER5+和 Itanium2 处理器使用在其他两个系统。

对于一个给定数量的核心计数，最好的性能来实现最小的可能 N 2 混合和 MPI 版本。随着 N 2 的增加，运行时也增加。混合动力版本显示更大的跳跃在 P8 OpenMP 线程，尤其是在 SGI 的 Altix 4700 系统，而 MPI 版本表现更平稳的趋势。得到一个更定量的 COM—型坯，如图 8 所示图与各种 N 1 N 2 组合的混合动力版本的 MPI 版本的时间比。大于 1 的值表示混合版本的更好性能。如图所示，在 n = 1 = 2 混合版本执行约 10%，比 2D MPI 版本，表明在混合版本的开销少。更好的混合—所坚持的 Altix ICE Nehalem 系统多达 4 个 OpenMP 线程（在一个插座的核心数量），对比其他两系统。这是由于改进的内存延迟在 Nehalem 系统。然而，在 8 OpenMP 线程的混合性能下降约 80%的 MPI 性能的 Altix ICE 和大约 42%的 Altix 4700，说明 NUMA 内存对这些系统的性能影响较大，OpenMP。IBM p575 内存节点相对平坦，介于两者之间。在 16 OpenMP 线程，性能下降甚至更大的.它指出了更好的 OpenMP 性能的改善内存延迟的重要性，因此，更好的混合性能。

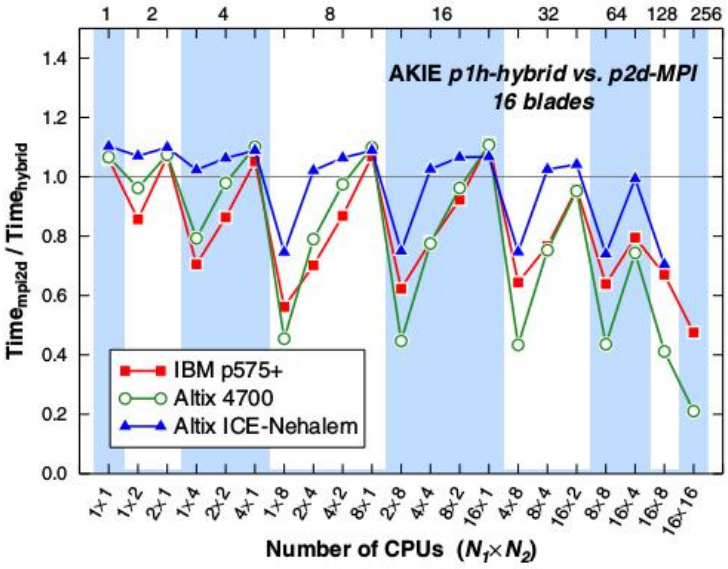

Fig. 8. Timing comparison of the AKIE hybrid and MPI codes.

4.5 混合方法综述

我们观察到了一些好处，从混合 MPI + OpenMP 的方法相比，眞正的 MPI 方法中的应用。混合方法可以减少内存占用和 MPI 调用和缓冲区相关的开销报价，提高负载平衡，并保持稳定的数值收敛。另一个尚未讨论的好处多的是减少这些应用表明利用多并行程序开发周期的潜力—TI 级并行，如由多区应用说明。我们还观察到一些限制因素与杂种方法。这些主要与 OpenMP 模型的限制，包括对内存的性能灵敏度延迟和没有语言的机制来加强数据亲和减轻 NUMA 效果。OpenMP 线程的数目是由硬件 SMP 节点上可用的内核数量限制。在下一节中，我们将讨论改进模型中数据局部性的 OpenMP 编程模型。

最后，MPI 进程和 OpenMP 线程之间的相互作用没有很好地定义在任一模型中，这可能有性能的其他后果。正如以前的研究指出的，例如，[26]，正确绑定进程和线程 CPU 可以在某些情况下提高性能。虽然没有介绍在这里，我们发现，性能影响较大的 NUMA 系统，如 SGI 的 Altix 4700，并在同时存在多线程（SMT）。

5.局部扩展 OpenMP

正如我们在前面的章节中观察到 ,提高混合性能的关键之一是提高 OpenMP 性能.根本的困难是,目前的 OpenMP 模型仍然是基于一个平面内存假设与现代体系结构不匹配,往往表现出更为复杂和复杂的存储子系统。管理数据布局和维护数据和任务的亲和力是实现可扩展性能的重要因素在这样的系统。在本节中，我们将局部性引入 OpenMP 作为语言扩展而不牺牲 OpenMP 模型的简单性。我们描述了建议的语法指定的位置和数据布局，允许用户控制任务数据的亲和力。更详细的讨论，这项工作可以在[12]。本节假定读者熟悉 OpenMP；为进一步详情请参阅[22]。

5.1 定位介绍

关键是引入到 OpenMP 作为一个虚拟层来表达的地方和任务数据的如何定位在程序中。定位的概念类似于在 X10 和现场的概念。我们定义位置 OpenMP 如下。

位置包括虚拟任务执行引擎及其相关存储器。在运行时，位置可以映射到硬件资源,如共享内存节点缓存相干 NUMA 节点,或一个分布式共享内存（DSM）结。分配给一个位置的任务可以同时执行通常的 OpenMP 规则。我们期望的任务在某个位置上运行可能比其他位置上的存储器更快地访问本地存储器。

我们引入一个参数 nlocs 作为预定义的变量对 OpenMP 程序的位置数。它可以被定义在运行时通过环境变量 omp_num_locs 或在编译时通过编译器。每个位置是在唯一确定的范围内。

5.1.1 位置要求

Location（m [: n]）要求可以用于并行和任务分配的隐式和显式指令 OpenMP 任务到一组位置。m 和 n 是定义位置范围的整数。单个值 m 表示单个位置。为了支持某种程度的容错的情况下，m 或 n 的值是出了可用的位置约束元，实际分配的位置是确定的（m 或 n 模 nlocs）。

线程（或隐式任务）从一个并行区域到位置的映射是在（默认的）块分发方式下完成的。例如，如果我们有 16 个线程和 4 个位置，如下面的例子所示：

#pragma omp parallel location (0:3) num_threads (16)

线程 0 - 3 将被分配到位置 0，线程 4 - 7 到位置 1 的映射方案是可能的，但我们只有讨论这个建议中最简单的一个。

5.1.2 位置继承规则

位置的继承规则的并行区域和任务没有“'location”条款是分层次的，即它从父线程继承。在程序执行开始时，默认位置是关联所有位置。因此当没有与顶层并行区域相关联的位置时，将从并行区域中映射隐式任务在所有位置的块分布方式。当遇到没有指定位置的嵌套并行区域时嵌套的并行区域将在其父节点的同一位置执行。这种设计选择允许自然任务定位亲和性，因此，任务数据的亲和力，以实现良好的数据局部性的嵌套并行的存在。

如果任务已分配给某个位置，那么如果没有其他位置，所有子任务将在同一位置运行 子句被指定。相反，如果为其子类指定了一个位置子句，则将执行子任务指定位置。

5.1.3 映射位置到硬件

逻辑位置到下硬件的映射不是在语言级别指定的，而是留给运行时实现或外部部署工具。运行时实现可以分配位置到计算节点的查询结果的基础上的机器拓扑和绑定线程的位置，如果需要。或者，外部工具（也许叫 omprun）将允许用户描述机床结构信息和配置如何映射位置到底层硬件。它将与 OS /作业调度程序交互以分配所需的资源 OpenMP 程序。

5.2 数据布局

当前 OpenMP 规范没有指定如何和在目标硬件上分配数据的位置。它通常依赖于底层操作系统的默认内存管理策略。最常见的是第一次接触策略，即 CPU 触摸数据页，将在包含 CPU 的节点上分配内存页。根据定位的概念，我们在 OpenMP 中介绍一种表达数据布局的方法。目标是让用户控制位置和时间共享数据。数据布局，目前，是静态的程序的生命周期期间。也就是说，我们不是考虑数据分布或迁移，此时却可以延长方案支持动态性之后。对于没有用户指定的数据布局，系统会使用默认的内存管理策略，共享内存编程数据分布最初包含在 SGI MIPSPro 编译器。

5.2.1 分配指令

我们使用一个数据分发语法来表达数据布局，作为一个指令在 OpenMP 如下。

#pragma omp distribute (dist-type-list: variable-list) [location (m:n)]

dist-type-list 是一个逗号分隔的在相应的阵列尺寸分布类型列表变量。变量列表中的变量应该具有与分布类型。可能分布类型包括“BLOCK”一块分布位置的列表中给出的地点条款，以及“/”非分布式的尺寸。如果位置子句不存在，则表示所有位置将用于分配。其他类型的数据分布也是可能的，但我们不讨论它们在这里简单起见。

分布式数据仍然保持其全局地址，并与程序中的其他共享数据一样访问。分布式数据，用户可以控制和管理共享数据，提高数据的地方 OpenMP 程序。以下示例显示了数组 A 的第一个维度是如何在所有位置上分布的。

```
double A[N][N];
#pragma omp distribute (BLOCK/: A) location (0:NLOCS-1)
```

5.2.2 onloc 要求

为了实现更大的控制任务数据的问题，我们可以映射在 OpenMP 的隐式和显式的任务的位置，无论是一个明确的位置号码或分布式数据的位置。对于后者，我们引入“'onloc”，将任务分配到一个基于数据所在的位置。只有分布变量是有 onloc 要求。变量可以是整个数组（用于并行结构），也可以是数组元素（用于任务构建）。在下面的例子中

```
#pragma omp task onloc (A[i])
foo (A[i]);
```

为执行任务的位置是由一个的位置确定。当 onloc 要求适用于并行区域，它意味着该区域的位置列表来自与分布式变量相关联的列表，如下面的例子所示，

```
#pragma omp task onloc (A[i])
foo (A[i]);
```

为执行任务的位置是由一个位置确定。当 onloc 要求适用于并行区域，它意味着该区域的位置列表来自与分布式变量相关联的列表，如下面的例子所示，

```
#pragma omp parallel onloc (A)
{. . .}
```

映射隐式任务的位置，这种情况下，然后遵循相同的规则所讨论的位置子句。相对于位置的 onloc 方法可以达到更好的控制任务的数据需求。

以下限制适用于位置和 onloc 要求：最多一个 onloc 或 location 可以出现并行或任务指令。如果一个非分布式的变量是用于 onloc 要求，它将被视为如果没有这样的子句被指定，并且实现可能打印警告消息。

5.3 验证和实验

我们目前所提出的定位功能在 OpenUH 编译器[23]发展。在 OpenUH 编译器是一个对Open64 编译器分支作为研究基础设施的 OpenMP 编译器和工具研究。除了编译的运行时调用的 OpenMP 构建，重点是 OpenMP 的运行时基础设施的发展，以支持位置和数据布局管理。我们目前的方法依赖于 libnuma 图书馆[18]Linux 下可用于管理分布式资源。详细描述，请参阅[12]。

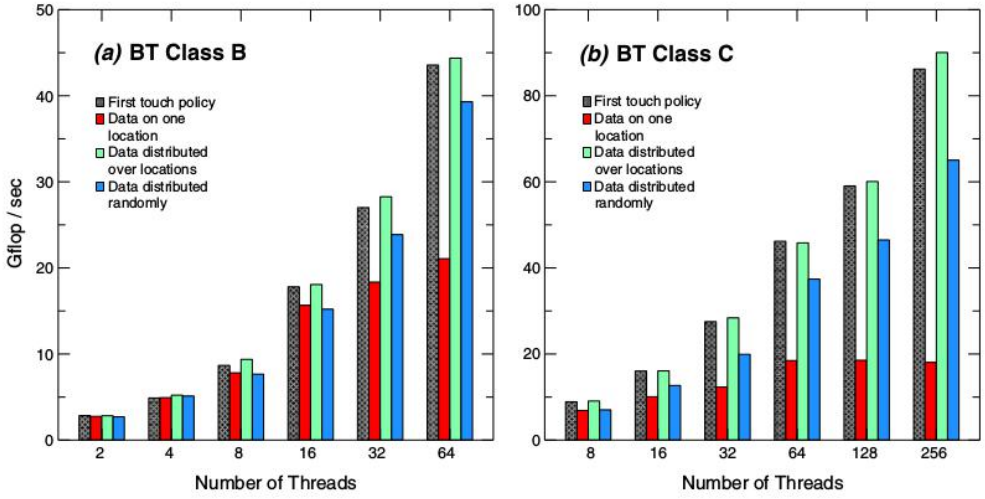

Fig. 9. BT performance (Gflop/s) from different data layouts.

由于我们的验证尚未完成，我们无法证明使用建议的性能提升特征.然而，作为一个实验，我们修改了 BT 的 NPB 基准套件手动来说明重要性，不同数据布局对性能的影响。收集在 SGI 的 Altix 4700 系统的结果如图 9 所示为 B 类和 BT C 类，高亿次/ s 表示更好的性能。每个线程数的四个不同的列显示性能—四种不同的数据布局策略：第一次实验，一个位置上的所有数据，基于数据访问模式的布局随机放置。从这个实验中，我们观察到显着的性能影响，从不同的数据布局 NUMA 系统，特别是对于大的数据集。一般来说，基于数据访问模式的布局（如第三列所示）产生最好的性能，其次是第一次实验（第一列）。单位置和随机布局比其他两个差。虽然分布在位置的数据的情况下，类似的情况下，使用第一次实验验证，我们相信，它可以进一步改善，通过优化数据布局与数据访问模式，这将是我们的编译器实现后可能完成的任务。

虽然工作的目标是通过编译器指令以实现所需的性能引入，一组最小的功能增益，预计一些代码转换可能需要（通过编译器或手工），为了实现高性能。例如，静态的数据布局在目前的建议将不能很好的应用程序—在不同计算阶段改变数据访问模式的阳离子。一个可能的解决方案是介绍—对不同阶段的数据重分布

的概念。然而，数据再分配的好处可能会蒙上阴影。另一种方法将涉及使用不同的数据布局合适的重复数据集。对于每个计算阶段，这种方法的明显缺点是增加内存占用。检查建议的地方扩展的有效性，我们需要更多的实验与实际应用。

6.相关的工作

MPI 和 OpenMP 是近年来研究的两种并行编程模型。有许多关于两种编程模型和混合方法的出版物。混合 MPI 和 OpenMP 的一个很好的总结程序可以在由 Rabenseifner 等人教程发现。相关出版物包括 perfor—该 npb-mz 混合代码，在 AMD Opteron 集群指出一些键实现性能研究良好的可扩展性。描述不同编程选择性能影响的分析程序和 npb-mz Cray XT5 平台上。他们提出了混合模型对内存使用的好处。我们目前的工作扩展的研究，以英特尔处理器为基础的集群详细，并提供了底层 MPI 库实现更好的理解效果。

我们以前的工作比较了在三个并行系统混合溢出代码的性能。目前工作提供了一个更全面的分析的基础上的数值精度和收敛的混合方法的性能。我们还利用 Alike 代码介绍更多 apples-to-apples 测试和纯 MPI 方法的比较。

包括在 SGI MIPSPro 编译器共享存储编程数据分布的概念。benkner 又作了一次尝试 bircsak 扩展 OpenMP ccNUMA 机器。这些努力基于从高性能 Fortran 数据分布的概念（HPF）。我们的扩展 OpenMP 是建立在定位的概念，它遵循类似的概念在现代高性能计算机语言提供了一个虚拟层 NUMA 效果表达。这种设计选择的目的是采用简单的 OpenMP 编程。

7.总结

总之，我们已经提出了研究混合 MPI + OpenMP 编程方法的案例，它适用于两个伪应用基准和两个眞实世界的应用程序，并表现出效益的混合方法的性能三多核并行系统的资源使用。由于目前的高性能计算系统走向 ExaScale 计算，每核心资源，如内存有望成为最小。因此，减少内存使用和开销与 MPI 调用和缓冲导致的混合方法将变得更重要。我们的研究也显示了有用的混合方法，提高负载平衡和数值计算的收敛性。尽管现代的并行编程语言如 HPCS 语言是成熟的，较传统的 MPI + OpenMP 的方法仍然是可行的集群上的编程方法。

有两个主要的限制，阻碍了更广泛地采用混合编程：没有明确定义的 MPI 和 OpenMP 线程的过程和有限的性能，尤其是高速架构的 OpenMP。那里需要采取很大的努力，在 MPI 社区，以改善与线程的接口在 MPI 3。在本文中我们有描述我们的方法来扩展 OpenMP 位置的概念，以提高在 ccNUMA 系统程序性能的所得。我们还提出了语法的语言扩展，以表达任务数据的亲和力。未来工作，这将是有益的应用程序的性能灵敏度进行更详细的研究内存延迟（如通过硬件性能计数器）和 MPI 参数。我们想完善的位置概念设计和通过研究测试用例和基准的性能，验证扩展的有效性，例如，调查不同的线程到位置映射的性能。另一个感兴趣的领域是扩展位置概念相比于其他 NUMA SMPS 平台。

致谢

我感谢 Jahed Djomehri 提供 cart3d 结果，Johnny Chang 和 Robert Hood 提供了他们有价值的讨论和评论，在 NAS 系统进行这些性能的措施，NAS 上人员的支持，和匿名审稿人对他们宝贵的意见手稿。由休士顿大学国家科学基金会的支持

参考文献

[1] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele, Project Fortress: A Multicore Language for Multicore Processors. Linux
Magazine, 2007.
[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, M. Yarrow, The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020, NASA Ames
Research Center, Moffett Field, CA, 1995.
[3] S. Benkner, T. Brandes, Exploiting data locality on scalable shared memory machines with data parallel programs, in: Proceedings of the Euro-Par 2000
Parallel Processing, Munich, Germany, 2000, pp. 647–657.
[4] M.J. Berger, M.J. Aftosmis, D.D. Marshall, S.M. Murman, Performance of a new CFD flow solver using a hybrid programming paradigm, Journal of Parallel
and Distributed Computing 65 (4) (2005) 414–423.
[5] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, C.D. Offner, Extending OpenMP for NUMA machines, in: Proceedings of the 2000
ACM/IEEE Conference on Supercomputing, Dallas, TX, 2000.
[6] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the chapel language, International Journal of High Performance Computing
Applications 21 (3) (2007) 291–312.
[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grotho, A. Kielstra, V. Saraswat, V. Sarkar, C.V. Praun, X10: an object-oriented approach to non-uniform cluster
computing, in: Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, ACM
SIGPLAN, 2005, pp. 519–538.
[8] R. Diaconescu, H. Zima, An approach to data distributions in chapel, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 313–335.
[9] J. Djomehri, D. Jespersen, J. Taft, H. Jin, R. Hood, P. Mehrotra, Performance of CFD applications on NASA supercomputers, in Proceedings of the
International Conference on Parallel Computational Fluid Dynamics, 2009, pp. 240–244.
[10] R.L. Graham, G. Bosilca, MPI forum: preview of the MPI 3 standard. SC09 Birds-of-Feather Session, 2009. Available from: <http://www.open-mpi.org/
papers/sc-2009/MPI_Forum_SC09_BOF-2up.pdf>.

[11] C. Hah, A.J. Wennerstrom, Three-dimensional flow fields inside a transonic compressor with swept blades, ASME Journal of Turbomachinery 113 (1)
(1991) 241–251.
[12] L. Huang, H. Jin, Liqi Yi, B. Chapman, Enabling locality-aware computations in OpenMP, Scientific Programming 18 (3-4) (2010) 169–181 (special issue).
[13] L. Hochstein, V.R. Basili, The ASC-alliance projects: a case study of large-scale parallel scientific code development, Computer 41 (3) (2008) 50–58.
[14] L. Hochstein, F. Shull, L.B. Reid, The role of MPI in development time: a case study, in: Proceedings of the 2008 ACM/IEEE Conference on
Supercomputing, Austin, Texas, 2008.
[15] H. Jin, R.F. Van der Wijngaart, Performance characteristics of the multi-zone NAS parallel benchmarks, Journal of Parallel and Distributed Computing
66 (2006) 674–685.
[16] H. Jin, R. Hood, P. Mehotra, A practical study of UPC with the NAS parallel benchmarks, in: Proceedings of the 3rd Conference on Partitioned Global
Address Space (PGAS) Programming Models, Ashburn, VA, 2009.
[17] D. Kaushik, S. Balay, D. Keyes, B. Smith, Understanding the performance of hybrid MPI/ OpenMP programming model for implicit CFD codes, in:
Proceedings of the 21st International Conference on Parallel Computational Fluid Dynamics, Moffett Field, CA, USA, May 18–22, 2009, pp. 174–177.
[18] A. Kleen, An NUMA API for Linux, SUSE Labs, 2004. Available from: <http://www.halobates.de/numaapi3.pdf>.
[19] G. Krawezik, F. Cappello, Performance comparison of MPI and three OpenMP programming styles on shared memory Multiprocessors, in: Proceedings
of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, 2003, pp. 118–127.
[20] R.H. Nichols, R.W. Tramel, P.G. Buning, Solver and turbulence model upgrades to OVERFLOW 2 for unsteady and high-speed applications, in:
Proceedings of the 24th Applied Aerodynamics Conference, volume AIAA-2006-2824, 2006.
[21] R. Numrich, J. Reid, Co-array fortran for parallel programming, In ACM Fortran Forum 17 (1998) 1–31.
[22] OpenMP Architecture Review Board, OpenMP Application Program Interface 3.0, 2008. Available from:

<http://www.openmp.org/>.
[23] The OpenUH compiler project. Available from:<http://www.cs.uh.edu/openuh>.
[24] Pleiades Hardware. Available from: <http://www.nas.nasa.gov/Resources/Systems/pleiades.html>.
[25] R. Rabenseifner, G. Hager, G. Jost, Tutorial on hybrid MPI and OpenMP parallel programing, in: Supercomputing Conference 2009 (SC09), Portland, OR,
2009.
[26] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programing on clusters of multi-core SMP nodes, in: Proceedings of the 17th Euromicro
International Conference on Parallel, Distributed and Network-Based Processing (PDP2009), Weimar, Germany, 2009, pp. 427–436.
[27] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, R. Biswas, Scientific application-based performance comparison of SGI Altix 4700, IBM Power5+, and
SGI ICE 8200 supercomputers, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008.
[28] H. Shan, H. Jin, K. Fuerlinger, A. Koniges, N. Wright, Analyzing the Performance Effects of Programming Models and Memory Usage on Cray XT5
Platforms, Cray User Group (CUG) meeting, Edingburgh, United Kingdom, 2010.
[29] Silicon Graphics, Inc., MIPSpro (TM) Power Fortran 77 Programmer’s Guide, Document 007-2361, SGI, 1999.
[30] The Top500 List of Supercomputer Sites. Available from: <http://www.top500.org/lists/>.
[31] The UPC Consortium, UPC Language Specification (V1.2), 2005. Available from: <http://www.upc.gwu.edu/documentation.html>.
[32] R.F. Van der Wijgnaart, H. Jin, The NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett
Field, CA, 2003.
[33] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken, Titanium: a high-
performance java dialect, in: Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.

Parallel Computing 37 (2011) 562–575

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

High performance computing using MPI and OpenMP on multi-core parallel systems

Haoqiang Jin [a,*], Dennis Jespersen [a], Piyush Mehrotra [a], Rupak Biswas [a], Lei Huang [b], Barbara Chapman [b]

[a] NAS Division, NASA Ames Research Center, Moffett Field, CA 94035, United States
[b] Department of Computer Sciences, University of Houston, Houston, TX 77004, United States

ARTICLE INFO

Article history:
Available online 2 March 2011

Keywords:
Hybrid MPI + OpenMP programming
Multi-core Systems
OpenMP Extensions
Data Locality

ABSTRACT

The rapidly increasing number of cores in modern microprocessors is pushing the current high performance computing (HPC) systems into the petascale and exascale era. The hybrid nature of these systems – distributed memory across nodes and shared memory with non-uniform memory access within each node – poses a challenge to application developers. In this paper, we study a hybrid approach to programming such systems – a combination of two traditional programming models, MPI and OpenMP. We present the performance of standard benchmarks from the multi-zone NAS Parallel Benchmarks and two full applications using this approach on several multi-core based systems including an SGI Altix 4700, an IBM p575+ and an SGI Altix ICE 8200EX. We also present new data locality extensions to OpenMP to better match the hierarchical memory structure of multi-core architectures.

Published by Elsevier B.V.

1. Introduction

Multiple computing cores are becoming ubiquitous in commodity microprocessor chips, allowing the chip manufacturers to achieve high aggregate performance while avoiding the power demands of increased clock speeds. We expect the core counts per chip to rise as transistor counts increase according to Moore's law. As evidenced by the systems on the TOP500 list [30], the high-end computing system manufacturers of today take advantage of economies of scale by using such commodity parts to build large HPC systems. The basic building block of such systems consists of multiple multi-core chips sharing memory in a single node. The dominant HPC architectures, currently and for the foreseeable future, comprise clusters of such nodes interconnected via a high-speed network supporting a heterogeneous memory model—shared memory with non-uniform memory access (NUMA) within a single node and distributed memory across the nodes.

From the user's perspective, the most convenient approach to programming these systems is to ignore the hybrid memory system and use a pure message passing programming model. This is a reasonable strategy on most systems, since most Message Passing Interface (MPI) library developers take advantage of the shared memory within a node and optimize the intra-node communication. Thus, the users get good performance for their applications while using a single standardized programming model. However there are other approaches to programming such systems. These include new languages such as those developed for DARPA's High Productivity Computing Systems (HPCS) program, Fortress [1], Chapel [6] and X10 [7], which are attempting to provide a very high level approach to programming petascale and exascale systems. Since these languages have just been proposed in the last few years and their implementations are in the nascent stages, it is difficult to predict their ultimate utility and performance.

* Corresponding author.
E-mail address: haoqiang.jin@nasa.gov (H. Jin).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2011.02.002

Another approach is embodied by the partitioned global address space (PGAS) languages that try to combine the best features of the shared memory programming model and the message passing model. Languages such as Co-Array Fortran [21], UPC [31] and Titanium [33] that utilize this model let users control data locality by specifying the distribution of data while enhancing programmability by allowing them to use a global name space with explicit specification of the remote data accesses. While these languages provide an incremental approach to programing HPC systems, studies have shown that the user has to carefully utilize the language features to obtain the performance of equivalent MPI programs [16].

In this paper, we focus on a hybrid approach to programming multi-core based HPC systems, combining standardized programming models – MPI for distributed memory systems and OpenMP for shared memory systems. Although this approach has been utilized by many users to implement their applications, we study the performance characteristics of this approach using some standard benchmarks, the multi-zone NAS Parallel Benchmarks (NPB-MZ), and two full applications used by NASA engineers, OVERFLOW and AKIE. We measure the performance of various hybrid configurations of these codes on several platforms including an SGI Altix 4700 (Intel Itanium2 chip), an IBM p575+ (IBM Power5+ chip) and an SGI Altix ICE 8200EX (Intel Xeon Nehalem-EP chip).

The paper also describes a new approach in which we extend the OpenMP model with new data locality extensions to better match the more complex memory subsystems available on modern HPC systems. The idea is to allow the user to use the simple shared memory model of OpenMP while providing "hints" in the form of data locality pragma to the compiler and the runtime system for them to efficiently target the hierarchical memory subsystems of the underlying architectures.

The paper is organized as follows. In the next section, we briefly describe the MPI and OpenMP models and the hybrid approach that we have studied in this paper. Section 3 presents the descriptions of the multi-zone NAS Parallel Benchmarks and the two applications OVERFLOW and AKIE. Section 4 starts with a description of the parallel systems used in the study, discusses the performance results that we obtained and concludes with a brief summary of our results. In Section 5, we present the proposed extensions to OpenMP describing the syntax and semantics of the various directives. Sections 6 and 7 describe some related work and the conclusions of our paper, respectively.

2. MPI and OpenMP programming models

MPI and OpenMP are the two primary programming models that have been widely used for many years for high performance computing. There have been many efforts studying various aspects of the two models from programmability to performance. Our focus here is the hybrid approach based on the two models, which becomes more important for modern multi-core parallel systems. We will first give a brief summary of each model to motivate our study of this hybrid approach.

2.1. Why still MPI?

MPI is a *de facto* standard for parallel programming on distributed memory systems. The most important advantages of this model are twofold: achievable performance and portability. Performance is a direct result of available optimized MPI libraries and full user control in the program development cycle. Portability arises from the standard API and the existence of MPI libraries on a wide range of machines. In general, an MPI program can run on both distributed and shared memory machines.

The primary difficulty with the MPI model is its discrete memory view in programming, which makes it hard to write and often involves a very long development cycle [13,14]. A good MPI program requires carefully thought out strategies for partitioning the data and managing the resultant communication. Secondly, due to the distributed nature of the model, global data may be duplicated for performance reasons, resulting in an increase in the overall memory requirement. Thirdly, it is very challenging to get very large MPI jobs running on large parallel systems due to machine instabilities and the lack of fault tolerance in the model (although this is true in general for other programming models). Also, large MPI jobs often demand substantial resources such as memory and network.

Lastly, most MPI libraries have a large set of runtime parameters that need to be "tuned" to achieve optimal performance. Although a vendor usually provides a default set of "optimized" parameters, it is not easy for a typical user to grasp the semantics and the optimal usage of these parameters on a specific target system. There is a hidden cost due to the underlying library implementation, which often gets overlooked. As an example, Table 1 lists timings measured for the CART3D

Table 1
CART3D timing (seconds) on a Xeon cluster.

Number of processes	8 GB node	16 GB node
32	*failed*	297.32
64	298.35	162.88
128	124.16	83.97
256	*failed*	42.62
`MPI_BUFS_PER_HOST`	48	256
`MPI_BUFS_PER_PROC`	32	128

application [4] on a cluster of Intel Xeon E5472 (Harpertown) nodes with two sizes of node memory [9]. The two parameters, `MPI_BUFS_PER_HOST` and `MPI_BUFS_PER_PROC`, control how much internal buffer space is allocated for communication in the MPI library. The results show a large impact on performance (more than 50%) from different MPI buffer sizes. The two failed cases on the smaller memory (8 GB) nodes are due to running out of memory in these nodes, but from different causes: at 32 processes, the application itself needs more memory; at 256 processes, memory usage of internal MPI buffers is too large (which may increase as a quadratic function of the number of processes, as we find out in Section 4).

2.2. Why OpenMP?

OpenMP, based on compiler directives and a set of supporting library calls, is a portable approach for parallel programming on shared memory systems. The OpenMP 2.5 specification (and its earlier versions) focuses on loop-level parallelism. With the introduction of *tasks* in the OpenMP 3.0 specification [22], the language has become more dynamic. OpenMP is supported by almost all major compiler vendors. The well-known advantage of OpenMP is its global view of application memory address space that allows relatively fast development of parallel applications. The directive based approach makes it possible to write sequentially consistent codes for easier maintenance. OpenMP offers an incremental approach towards parallelization, which is advantageous over MPI in the parallel code development cycle.

The difficulty with OpenMP, on the other hand, is that it is often hard to get decent performance, especially at large scale. This is due to the fine-grained memory access governed by the memory model that is unaware of the non-uniform memory access characteristics of the underlying shared address space system. It is possible to write SPMD-style codes by using thread ids and managing data explicitly as is done in an MPI code and researchers have demonstrated success using this approach to achieve good performance with OpenMP in real-world large-scale applications [4,19]. However, this defeats some of the advantages in using OpenMP.

2.3. The MPI + OpenMP approach

As clusters of shared memory nodes become the dominant parallel architecture, it is natural to consider the hybrid MPI + OpenMP approach. The hybrid model fits well with the hierarchical machine model, in which MPI is used for communication across distributed memory nodes and OpenMP is used for fine-grained parallelization within a node. Taking advantages of both worlds, the hybrid model in principle can maintain cross-node performance with MPI and reduce the number MPI processes needed within a node and thus, the associated overhead (e.g., message buffers). As we discuss later in Section 4.2, the hybrid approach can help reduce the demand for resources (such as memory and network), which can be very important for running very large jobs. For certain class of applications with easily exploitable multi-level parallelism, the hybrid model can potentially reduce application development effort also. In the following sections, we will examine the aforementioned advantages and potential disadvantages of the hybrid approach in a number of case studies.

3. Hybrid programming case studies

This section briefly describes the characteristics of the applications that were implemented using the hybrid MPI + OpenMP model, two pseudo-application benchmarks and two real-world applications. The next section will focus on the performance of these applications on several parallel systems.

3.1. Multi-zone NAS Parallel Benchmarks

The multi-zone versions of NAS Parallel Benchmarks (NPB-MZ) are derived from the pseudo-application benchmarks included in the regular NPBs [2,32]. They mimic many real-world multi-block codes that contain exploitable parallelism at multiple levels. Due to the structure of the physical problem, it is natural to use MPI for zonal coarse grain parallelism and OpenMP for fine grain parallelism within a zone. There are three benchmark problems defined in NPB-MZ, as summarized in Table 2, with problem size ranging from Class S (the smallest) to Class F (the largest). Both SP-MZ and LU-MZ have fixed-size zones for a given problem class and load balancing in this case is simple and straightforward. On the other hand, the zone sizes for a given class in BT-MZ can vary by as much as a factor of 20, raising load balancing issues. Due to the fixed number of zones in LU-MZ, the exploitable zonal parallelism for this benchmark is limited.

Table 2
Three benchmark problems in NPB-MZ.

Benchmark	Number of zones	Zonal size
BT-MZ	Increases with problem size	Varies within a class
SP-MZ	Increases with problem size	Fixed within a class
LU-MZ	Fixed (=16)	Fixed within a class

The hybrid MPI + OpenMP implementation of NPB-MZ uses MPI for inter-zone parallelization and a bin-packing algorithm for balancing loads among different zones. There is no further domain decomposition for each zone. OpenMP parallelization is used for loops within a zone. For details of the implementations, please refer to [15]. Because of the limited number of zones (=16) in LU-MZ, which limits the number of MPI processes, we have not used this benchmark in this study.

3.2. Overflow

OVERFLOW is a general-purpose Navier–Stokes solver for Computational Fluid Dynamics (CFD) problems [20]. The code is focused on high-fidelity viscous simulations around realistic aerospace configurations. The geometric region of interest is decomposed into one or more curvilinear but logically Cartesian meshes. Arbitrary overlapping of meshes is allowed. The code uses finite differences in space, implicit time stepping, and explicit exchange of information at overlap boundaries. For parallelization, meshes are grouped and assigned to MPI processes. For load balancing purposes, each mesh may be further decomposed into smaller domains. During the solution process these smaller domains act as logically independent meshes. The hybrid decomposition for OVERFLOW involves OpenMP parallelism underneath the MPI parallelism. All MPI ranks have the same number of OpenMP threads. The OpenMP shared-memory parallelism is at a fairly fine-grained level within a domain.

The numerical scheme in OVERFLOW involves a flow solver step on each mesh followed by an explicit exchange of boundary overlap information. For a given case, as the number of MPI processes increases, more mesh splitting generally occurs in order to get good load balancing. There are a few drawbacks to this strategy. First, more mesh splitting results in a numerical algorithm which has more of an explicit flavor (with unlimited mesh splitting the domain sizes could shrink to just a few mesh points each). This reduction of implicitness can result in slower convergence (or even non-convergence). Slower convergence implies that more iterations (and thus longer wallclock time) would be needed to reach the desired solution. This is illustrated in Fig. 1 for an airfoil test case, showing the effect on the convergence rate of splitting a single mesh into 120 zones. Without mesh splitting, the calculated L2 residual monotonically decreases as the iteration proceeds. For the split case, the residual appears to be non-convergent. This is admittedly a highly unrealistic example, but for realistic cases a possible change in convergence behavior due to mesh splitting can add an undesired degree of uncertainty to what may already be a difficult solution process.

Secondly, more mesh splitting causes an increase in the number of mesh points, since "ghost" or "halo" points are used for affecting data interpolation at split boundaries. Table 3 shows the total number of mesh points for different numbers of domain groups produced from a dataset with 23 zones and 36 million mesh points. For 256 groups, the total mesh size is increased by 30%. Since the flow code solution time is essentially proportional to the total number of mesh points, the increase in mesh points partially counteracts the efficiency of increased parallelism.

The hybrid approach results in less mesh splitting, which provides twofold benefit: there are fewer total mesh points, and convergence behavior is less likely to be adversely affected. This is due to the fact that both these properties depend on the total number of MPI ranks and not on the total number of processor cores. The impact of hybrid versus MPI parallelization on the convergence rate of implicit methods has also been pointed out by Kaushik and his colleagues [17].

3.3. Akie

AKIE is a turbine machinery application that is used to study 3D flow instability around rotor blades [11]. The multi-block configuration consists of solving partial differential equations for flow around each blade and interpolating solutions between blades. Parallelism can be exploited at both inter-blade and intra-blade levels. For performance reason, the code uses a 32-bit numerical algorithm. We started with a sequential version of AKIE and developed the hybrid MPI + OpenMP version

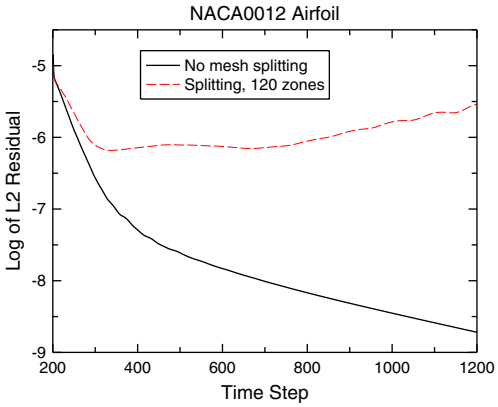

Fig. 1. Numerical convergence rate as a result of mesh splitting.

Table 3
OVERFLOW mesh size for different numbers of domain groups.

Number of groups	Total number of mesh points (M)
16	37
32	38
64	41
128	43
256	47

(*p1h*). Since this version did not achieve the desired scalability (notably on the Altix 4700 as discussed in the following section), we developed a pure MPI version (*p2d*) as well. Both versions are based on a hierarchical two-level parallelization approach with the difference being only in how the second-level parallelization is done, as illustrated in Fig. 2. The first-level parallelization applies domain decomposition to the blade dimension. Each MPI process (*P0*···*Pn*) works on one group of blades and exchanges boundary data at the end of each iteration. Parallelism at the first level is limited to the number of blades available in a given problem. In order to exploit additional parallelism, the second-level parallelization within a blade would be needed.

For the second-level parallelization, the hybrid version (*p1h*) uses OpenMP parallel regions for loops (mostly applied to the *j* dimension of the 3D space) within a blade. Thread synchronization is used to avoid race conditions in different regions. In contrast, the pure MPI version (*p2d*) applies further domain decomposition (to the *j* dimension) within a blade with an MPI process assigned to each sub-domain. Thus, each first-level MPI group consists of sub-level MPI processes working on sub-domains and performing additional communication within the group. There is no change in algorithm compared to the OpenMP approach; that is, both versions maintain the same numerical accuracy. For these reasons, AKIE is a good case for comparing the MPI and hybrid approaches. We should point out that both the hybrid and MPI versions were implemented from the same code base. In the actual implementation of the pure MPI version (*p2d*), a two-dimensional layout of the MPI communication groups is used to match with that of the data decomposition. The MPI processes at the two levels are initiated statically and there is no use of MPI-2's dynamic process creation in the code.

4. Performance studies

Our studies of the performance of the hybrid MPI + OpenMP applications were conducted on three multi-core parallel systems located at the NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center. We first briefly describe the parallel systems, and then compare performance of the hybrid applications. Throughout the rest of the paper, we use the terms "core" and "CPU" interchangeably.

4.1. Parallel systems

Table 4 summarizes the main characteristics of the three parallel systems used for this study. The first system, an SGI Altix 4700, is part of the Columbia supercluster and consists of 512 compute blades. Each blade hosts two dual-core Intel Itanium2 processors that share the same local memory board through a bus. The system has a cache-coherent non-uniform

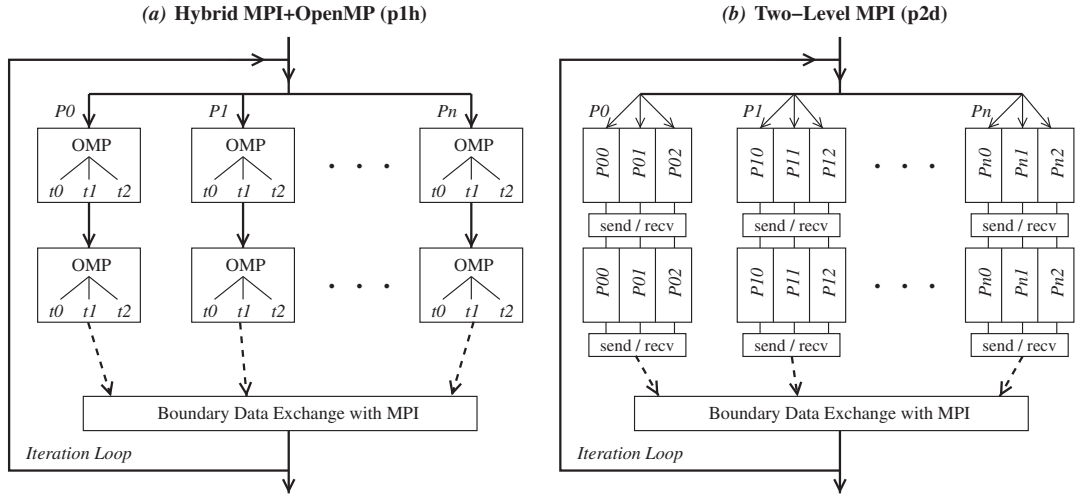

Fig. 2. Schematic comparison of the AKIE (a) hybrid and (b) MPI codes.

Table 4
Summary of the parallel systems.

System name	Columbia22	Schirra	Pleiades (Nehalem)
System type	SGI Altix 4700	IBM p575+	SGI Altix ICE 8200EX
Processor type	Intel Itanium2	IBM Power5+	Intel Xeon
Model	9040		X5570
Series (# Cores)	Montecito (2)	(2)	Nehalem-EP (4)
Processor speed	1.6 GHz	1.9 GHz	2.93 GHz
HT or SMT (Threads/Core)	–	2	2
L2 Cache (cores sharing)	256 KB (1)	1.92 MB (2)	256 KB (1)
L3 Cache (cores sharing)	9 MB (1)	36 MB (2)	8 MB (4)
Memory interface	FSB	On-Chip	On-Chip
FSB speed	667 MHz	–	–
No. of hosts	1	40	1280
Cores/host	2048	16	8
Total cores	2048	640	10240
Memory/host	4096 GB	32 GB	24 GB
Interconnect	NUMALink4	IBM HPS	InfiniBand
Topology	FatTree	FatTree	Hypercube
Operating system	SLES 10.2	AIX 5.3	SLES 10.2
Compiler	Intel-10.1	IBM XLF-10.1	Intel-11.1
MPI library	SGI MPT-1.9	IBM POE-4.3	SGI MPT-1.25

memory access (ccNUMA) architecture supported by the NUMAlink4 interconnect for communication between blades, which enables globally addressable space for 2048 cores in the system. The second system, an IBM p575+, is a cluster of IBM Power5+ nodes connected with the IBM high-performance switch (HPS). Each node contains 8 dual-core IBM Power5+ processors and offers relatively flat access to the node memory from each of the 16 cores in the node as a result of the round-robin memory page placement policy. Each Power5+ processor is capable of simultaneous multithreading (SMT). The third parallel system, an SGI Altix ICE 8200EX, is part of the newly expanded Pleiades cluster [24]. Pleiades comprises nodes based on three types of Intel Xeon processors: quad-core E5472 (Harpertown), quad-core X5570 (Nehalem-EP), and six-core X5670 (Westmere). All the nodes are interconnected with InfiniBand in a hypercube topology. In this study, we only used the Nehalem-based nodes. These nodes use two quad-core Intel X5570 (Nehalem-EP) processors. Each Nehalem processor contains four cores with hyperthreading (HT, similar to SMT) capability, which enables two threads per core. An on-chip memory controller connects directly to local DDR3 memory, which improves memory throughput substantially compared to the previous generations of Intel processors. However, accessing memory within a Nehalem-based node is non-uniform. Both SGI Altix systems have the first-touch memory placement policy as the default.

On the software side, both MPI and OpenMP are available on the SGI Altix 4700 through the SGI MPT library and the compiler support. For the two cluster systems, MPI, supported by the SGI MPT or the IBM POE library, is available for communication between nodes, and either MPI or OpenMP can be used within each node (see Table 4). For all of our experiments in the study, we fully populated the core resources on each node. For instance, to run an application on the SGI Altix ICE system with 32 MPI processes and 4 OpenMP threads per MPI process, we allocate 16 nodes for the job and assign 2 MPI processes on each node with a total of 8 threads per node. To ensure consistent performance results, we applied the process/thread binding tools available on each parallel system to bind processes and threads to processor cores.

4.2. Multi-zone NAS Parallel Benchmarks

As mentioned in Section 3.1, because of the limited number of zones in LU-MZ, which limits the number of MPI processes, we only discuss the performance of BT-MZ and SP-MZ in this section. Before diving into performance details, we first examine the memory usage of the benchmark applications for different process and thread combinations.

The measured memory usage for SP-MZ Class C on the Altix ICE is shown in Fig. 3: per process as a function of MPI processes on the left panel (a) and per core as a function of OpenMP threads on the right panel (b). The total number of processing cores used in each experiment equals to the number of MPI processes times the number of OpenMP threads per MPI process. The results were measured from the high-water-mark memory usage at the end of each run with a correction for the use of shared memory segment among processes on a node. For comparison, the actual memory used by the data arrays in the application is also included in the figure.

Within an Altix ICE node (up to 8 cores, Fig. 3(a)), the memory usage roughly reflects the need of the application data. As the number of MPI processes increases, the data size per process decreases, but the actual memory usage reduces only to a point (around 128 cores), then increases almost quadratically. The increased memory is mainly due to the increased MPI buffers allocated for collective communication between nodes in the underlying MPI library (the SGI MPT in this case) and is dependent on the number of nodes used. In the hybrid mode, for a given number of cores, we see substantial reduction in memory usage, especially in large core counts (see Fig. 3(b)). For instance for 256 cores, the 256×1 run (similar to pure

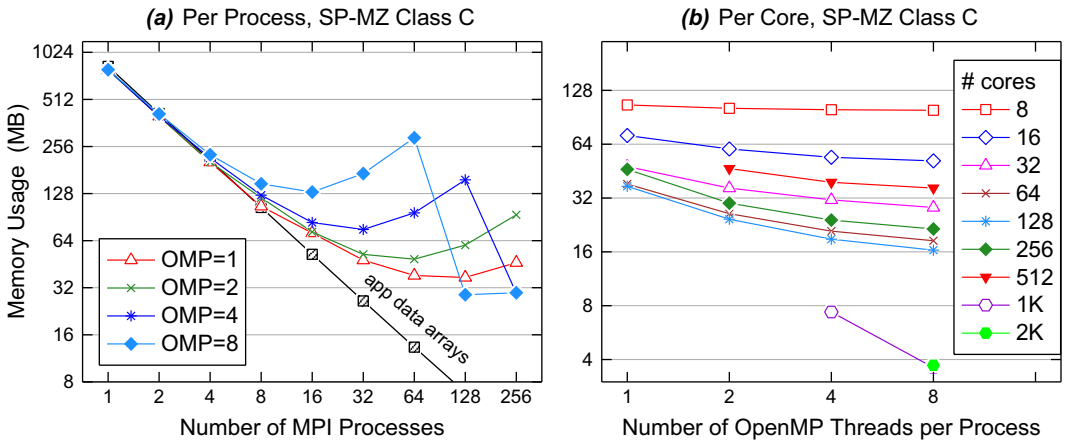

Fig. 3. SP-MZ Class C memory usage on the Altix ICE-Nehalem.

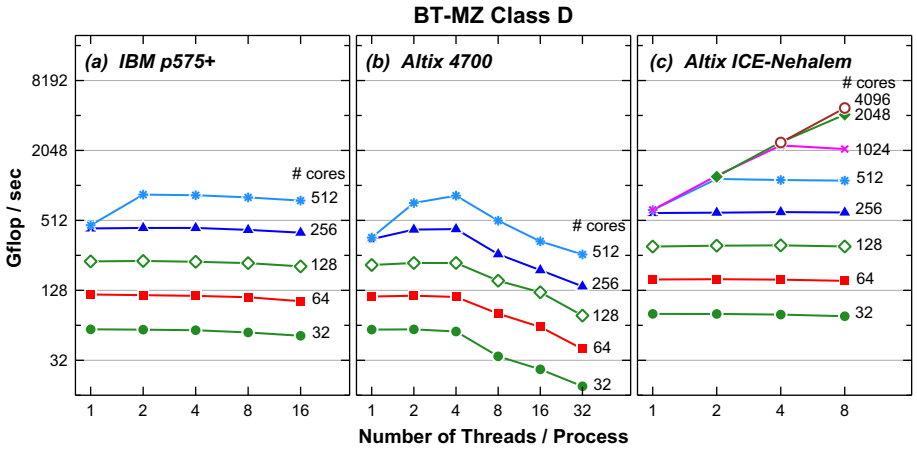

Fig. 4. BT-MZ performance (Gflop/s) on three parallel systems.

MPI) uses 11.9 GB (=256 ∗ 46.3 MB) of memory, while the 32 × 8 (hybrid) run uses only 5.5 GB (=32 ∗ 8 ∗ 21.5 MB) of memory. This large memory saving could be very important for future multi-core systems where less per-core memory is expected. The figure also shows a sudden drop in memory usage beyond 64 nodes (512 cores). This corresponds to the MPT, SGI's MPI library, switching to a less optimal communication scheme, which allocates less space for message buffers and could cause performance degradation for communication in certain applications (such as the CART3D case described in Section 2). Although using a different data array size, the memory profile of BT-MZ (not shown here) is very similar to that of SP-MZ. The trends in memory usage for Class D (also not shown here) are very similar to Class C.

Figs. 4 and 5 show the performance (Gflop/s, higher is better) of BT-MZ and SP-MZ Class D on three parallel systems. Each curve corresponds to a given number of cores used, while varying the number of OpenMP threads. For a given core count, going from left to right, the number of MPI processes decreases while the number of OpenMP threads per MPI process increases from 1 to 16 on the IBM p575+, 1 to 32 on the Altix 4700, and 1 to 8 on the Altix ICE-Nehalem. The Class D data set has a total of 1024 zones. Thus, MPI can only be used up to 1024 processes and beyond that, the hybrid approach is needed. For BT-MZ, MPI scales nicely up to 256 processes on all three systems, but beyond that there is no further improvement due to load imbalance as a result of uneven size zones in the benchmark (see Fig. 4). However, further performance gains can be obtained by using OpenMP threads to improve load balance at large core counts. For a given core count, BT-MZ shows relatively flat performance from different process-thread combinations as long as the load for MPI processes is balanced (up to 256 processes) on both IBM p575+ and Altix ICE-Nehalem. However, on the Altix 4700 the performance degrades quickly beyond 4 OpenMP threads when NUMA memory blades are involved. As discussed in Section 4.1 each of the Altix 4700 blade contains 4 Itanium2 cores sharing the same local memory board through a bus. Accessing memory in another blade for more than 4 cores has to go through the SGI NUMAlink, which has longer latency. The performance as shown in Fig. 4(b) is indicative of long memory latency associated with accessing remote memory on the system.

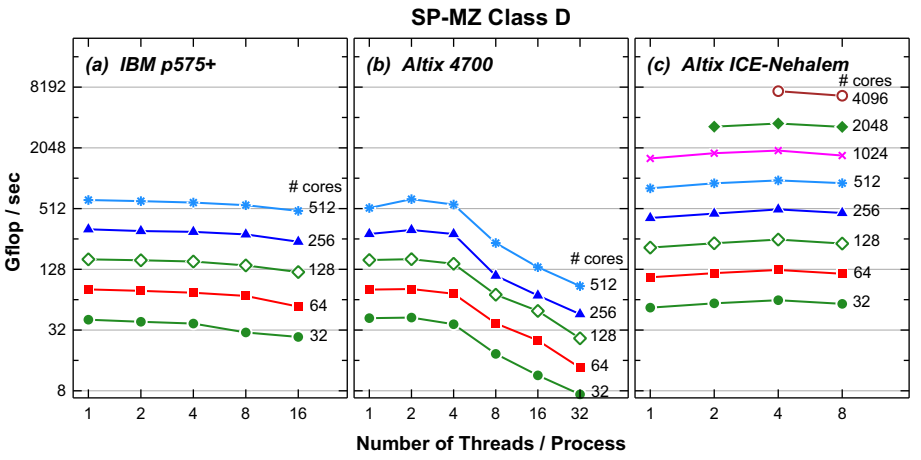

Fig. 5. SP-MZ performance (Gflop/s) on three parallel systems.

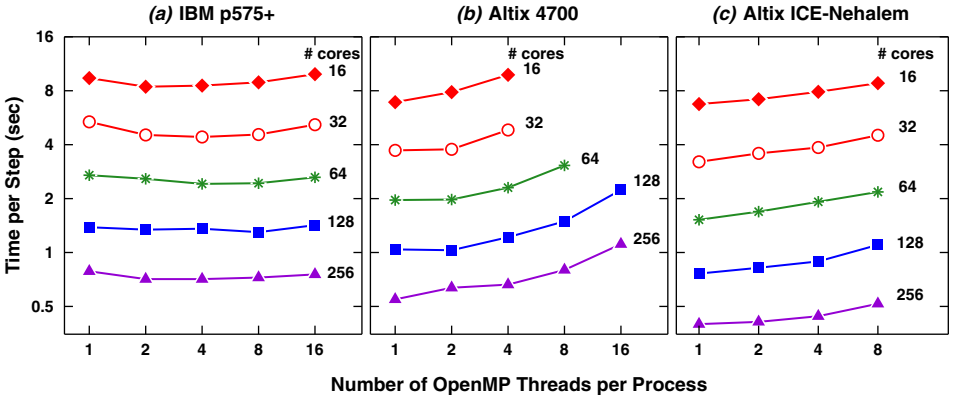

Fig. 6. OVERFLOW2 performance on three parallel systems. Numbers in the graphs indicate the number of cores used.

SP-MZ scales very well up to 1024 MPI processes on the Altix ICE-Nehalem (see Fig. 5(c)). However, there are performance differences from using different numbers of OpenMP threads for a given core count. The best performance is obtained when the number of OpenMP threads per process is 4. This phenomenon needs some explanation. First, using more OpenMP threads reduces the corresponding number of MPI processes. As a result, the total number of messages is reduced while the message sizes are increased, which leads to better communication performance and improves the overall performance. Secondly, the OpenMP threads spawned by the same MPI process work on zones in SP-MZ assigned to the MPI process together, one zone at a time. Using more OpenMP threads effectively increases the total cache size for the same amount of data and improves the performance (assuming one OpenMP thread assigned to one core). However, as the number of OpenMP threads increases, the overhead associated with OpenMP constructs and remote memory accesses across sockets in a node increases. From a combination of these factors, we see a relative performance increase up to 4 threads after which we see a performance drop for 8 threads.

On the IBM p575+ for a given core count, we see persistent performance drop for SP-MZ as the number of OpenMP threads increases. We attribute this result to the fact that on a p575+ node the memory page placement policy is round-robin (not first-touch) but the OpenMP parallelization for SP-MZ assumes a first-touch memory placement. Such a mismatch causes longer latency in data access from different OpenMP threads. As a result we do not see an effective increase in data cache as observed on the Altix ICE-Nehalem. In contrast, the performance of SP-MZ on the Altix 4700 is relatively flat up to 4 OpenMP threads and then drops quickly after that, which indicates the severe NUMA effect on the system.

4.3. Overflow

In our experiments, we used OVERFLOW 2.0aa on a realistic wing/body/nacelle/pylon configuration with 23 zones and 35.9 million mesh points. Fig. 6 shows the performance of the hybrid code in seconds per time step (lower is better) measured on the IBM p575+, the Altix 4700, and the Altix ICE-Nehalem using different process-thread combinations. The number

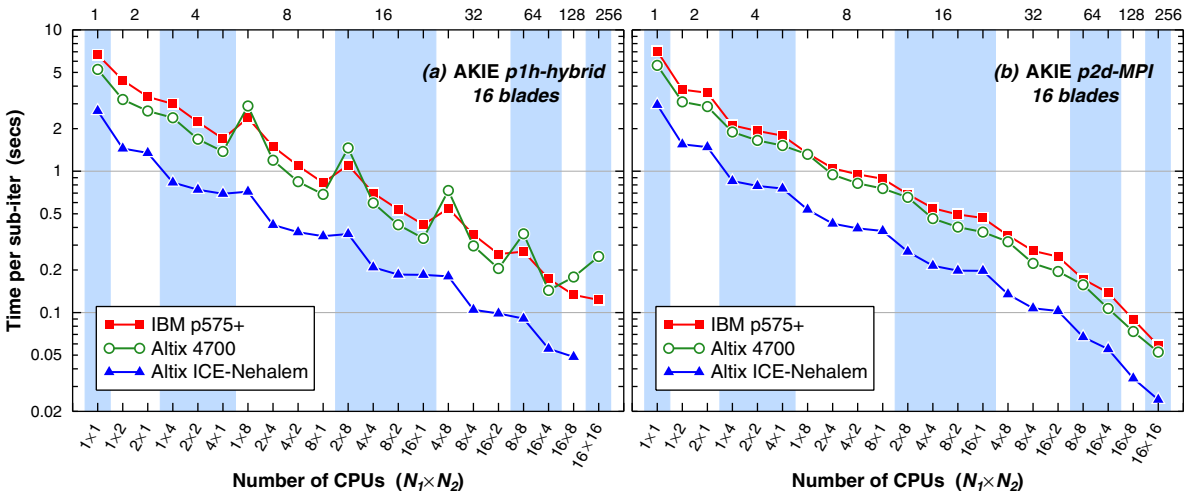

Fig. 7. Timings of the AKIE (a) hybrid and (b) MPI codes on three parallel systems.

of OpenMP threads per MPI process varies from 1 to 8 on the Altix ICE and up to 16 on the other two systems. The data points for the pure MPI version on all three systems are very close to those for the hybrid version with 1 OpenMP thread as plotted in the graph. On the IBM p575+, the hybrid version with multiple OpenMP threads outperformed the pure MPI version. The best results were observed with 4 OpenMP threads. Recall from Section 3.2 that as the number of MPI processes increases, OVERFLOW does more mesh splitting, resulting in more mesh points. As the number of OpenMP threads increases for a fixed total number of cores, fewer MPI processes are used and faster execution time might be expected since the total number of mesh points decreases. At the same time, overhead associated with OpenMP parallelization increases, which has a negative impact on performance. The balance point from the two effects depends on the application and on the system characteristics.

The benefit observed on the IBM system from the hybrid version is not seen on the two Altix systems. Instead, we observe a monotonic increase in time as the number of OpenMP threads increases. This increase in runtime is predominantly caused by the increase in remote memory accesses from OpenMP on the ccNUMA architectures, which overshadows any reduction in runtime from a smaller amount of work. The key to improving OpenMP performance on a NUMA-based system is to improve data locality and reduce remote memory access. In contrast, no obvious NUMA effect was observed on the IBM p575+, which can be attributed to the round-robin memory page placement policy on the system.

4.4. Akie

We used a 16-blade case to compare the performance of hybrid and MPI versions of the AKIE code on the three parallel systems. Per-iteration timings of runs from various process-thread combinations are shown in Fig. 7 (lower is better). In the $N_1 \times N_2$ notation, N_1 indicates the number of MPI processes at the first level, and N_2 indicates the number of OpenMP threads (the hybrid version) or MPI processes (the MPI version) at the second level for each of the first level MPI processes. The total number of cores used, $N_1 \times N_2$, increases from 1 to 256. The value of N_1 is limited by the number of blades, which is 16 in the current case. The value of N_2 is mostly constrained by the partitioned dimension of a given problem. For the hybrid case, the number of OpenMP threads N_2 is also limited by the size of an SMP node (16 on the IBM p575+, 8 on the Altix ICE, 2048 on the Altix 4700). For instance, we could not run the 16×16 hybrid version on the Altix ICE, but there is no such a limitation for the MPI version since N_2 MPI processes can be spread across multiple nodes if needed. Overall, we observe a similar performance for the IBM p575+ and the Altix 4700. The Nehalem-based SGI Altix ICE produces 2–3 times better performance than the other two systems. First, this is partly due to faster processor speed in the system. Second, the use of the 32-bit numerical algorithm in the application has larger performance boost (from the 64-bit numerical algorithm) on the Nehalem processor than on the Power5+ and Itanium2 processors used in the other two systems.

For a given number of core counts, the best performance is achieved with the smallest possible N_2 for both hybrid and MPI versions. As N_2 increases, the runtime also increases. The hybrid version shows a larger jump at $\geqslant 8$ OpenMP threads, especially on the SGI Altix 4700 system, while the MPI version shows a much smoother trend. To get a more quantitative comparison, Fig. 8 plots the ratio of timings of the MPI version with those of the hybrid version for various $N_1 \times N_2$ combinations. A value larger than 1 indicates better performance of the hybrid version. As shown in the figure, at $N_2 = 1$ the hybrid version performs about 10% better than the 2D MPI version, indicating less overhead in the hybrid version. The better hybrid performance persists on the Altix ICE-Nehalem system up to 4 OpenMP threads (number of cores in a socket), in contrast to other two systems. This is attributed to the improved memory latency in the Nehalem system. However, at 8 OpenMP threads the hybrid performance drops to about 80% of the MPI performance on the Altix ICE and about 42% on the Altix

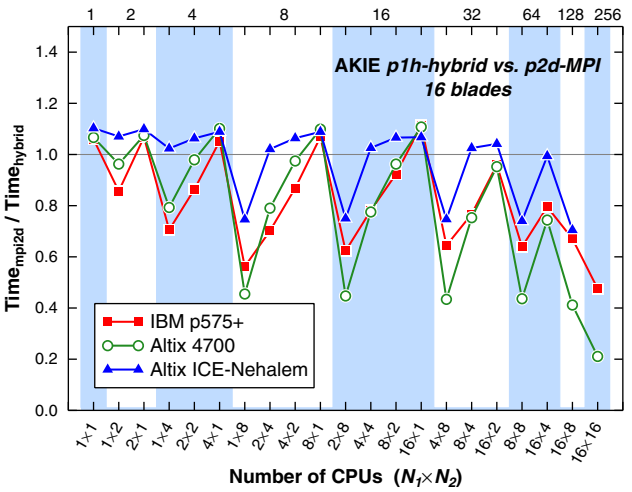

Fig. 8. Timing comparison of the AKIE hybrid and MPI codes.

4700, indicating large impact of NUMA memory on OpenMP performance on both of these systems. The IBM p575+, which has relatively flat memory nodes, is somewhere in between. At 16 OpenMP threads, the performance degradation is even larger. It points out the importance of improved memory latency for better OpenMP performance and, thus, better hybrid performance.

4.5. Summary of the hybrid approach

We observed a number of benefits from the hybrid MPI + OpenMP approach compared to the pure MPI approach for real-world applications. The hybrid approach can reduce the memory footprint and overhead associated with MPI calls and buffers, improve load balance, and maintain numerical convergence and stability. Another benefit that has not been discussed much is the potential to reduce the parallel code development cycle for those applications that demonstrate exploitable multi-level parallelism, as illustrated by the multi-zone applications. We also observed some limiting factors with the hybrid approach. These are mainly associated with limitations of the OpenMP model, including performance sensitivity to memory latency and no language mechanisms to enforce data affinity to mitigate the NUMA effect. The number of OpenMP threads is limited by the number of cores available on a hardware SMP node. In the next section, we will discuss extensions to the OpenMP programming model to improve data locality in the model.

Lastly, the interaction between MPI processes and OpenMP threads is not well defined in either model, which may have other consequences on performance. As pointed out by previous studies e.g., [26], properly binding processes and threads to CPUs (or *pinning*) can improve performance in some cases. Although not presented here, we have found that pinning has larger performance impact on the NUMA-based systems, such as the SGI Altix 4700, and in the presence of simultaneous multithreading (SMT).

5. Extensions to OpenMP for locality

As we observed in the previous sections, one of the keys to improving the hybrid performance is to improve the OpenMP performance. The fundamental difficulty is that the current OpenMP model is still based on a flat memory assumption that does not match with the modern architectures, which often exhibit more hierarchical and complex memory subsystems. Managing the data layout and maintaining the data and task affinity are essential factors to achieving scalable performance on such systems. In this section, we introduce the concept of locality into OpenMP as language extensions without sacrificing the simplicity of the OpenMP model. We describe the proposed syntax to specify location and data layout that allows the user to control task-data affinity. More detailed discussion of this work can be found in [12]. This section presumes that the reader is familiar with OpenMP; for further details please refer to [22].

5.1. Introduction of location

The key concept is the introduction of *location* into OpenMP as a virtual layer for expressing locality and task-data affinity in a program. The location concept is similar to the notion of *Place* in X10 [7] and *Locale* in Chapel [8]. We define location in OpenMP as follows.

Location: *a logical execution environment that contains data and OpenMP implicit and explicit tasks.*

A location includes a virtual task execution engine and its associated memory. At runtime, a location may be mapped onto hardware resources, such as a shared-memory node, a cache-coherent NUMA node, or a distributed shared-memory (DSM) node. Tasks assigned to a location may be executed concurrently following the usual OpenMP rules. We expect that tasks running on a location may have a faster access to local memory than to memory on other locations.

We introduce a parameter `NLOCS` as a pre-defined variable for the number of locations available in an OpenMP program. It can be defined at runtime via the environment variable `OMP_NUM_LOCS` or at the compile time through compiler flags. Each location is uniquely identified with a number `MYLOC` in the range of `[0:NLOCS-1]`.

5.1.1. The `location` clause

The "`location (m[:n])`" clause can be used with the `parallel` and `task` directives to assign implicit and explicit OpenMP tasks to a set of locations. *m* and *n* are integers defining the range of locations. A single value *m* indicates a single location. To support some level of fault tolerance for the case where the value of *m* or *n* is out of the available location boundary, the actually assigned location is determined by (*m* or *n* modulo `NLOCS`).

Mapping of threads (or implicit tasks) from a parallel region to locations is done in a (default) block distribution fashion. For instance, if we have 16 threads and 4 locations as shown in the following example,

```
#pragma omp parallel location (0:3) num_threads (16)
```

threads 0–3 will be assigned to location 0, threads 4–7 to location 1, etc. Other mapping policies are possible, but we only discuss the simplest one in this proposal.

5.1.2. Location inheritance rule

The location inheritance rule for those parallel regions and tasks without the "`location`" clause is hierarchical, that is, it is inherited from the parent. At the beginning of a program execution, the default location association is all locations. Thus, when there is no location associated with a top-level parallel region, implicit tasks from the parallel region will be mapped across all locations in a block distribution fashion. When encountering a nested parallel region without a specified location, the nested parallel region will be executed in the same location as its parent. This design choice allows natural task-location affinity and, thus, task-data affinity to achieve good data locality in the presence of nested parallelism.

If a task has been assigned to a location, all of its child tasks will be running on the same location if no other `location` clause is specified. On the contrary, if a `location` clause is specified for one of its children, the child task will be executed on the specified location.

5.1.3. Mapping locations to hardware

The mapping of logical locations to underneath hardware is not specified at the language level and, rather, is left to the runtime implementation or an external deployment tool. The runtime implementation may assign locations to computing nodes based on the query result of the machine topology and bind threads to locations if required. Alternatively, an external tool (perhaps named `omprun`) will allow users to describe the machine architecture information and configure how to map locations onto the underlying hardware. It will interact with OS/job schedulers to allocate the required resources for the OpenMP programs.

5.2. Data layout

The current OpenMP specification does not specify how and where data gets allocated on the target hardware. It usually relies on the default memory management policy of the underlying operating system. The most common one is the *first touch* policy, that is, whichever CPU touches a data page first will allocate the memory page on the node containing the CPU. With the concept of location, we introduce in OpenMP a way to express data layout. The goal is to allow users to control and manage shared data among locations. The data layout, currently, is static during the lifetime of the program. That is, we are not considering data redistribution or migration at this time but may extend the proposal to support dynamicity later on. For data without a user-specified data layout, the system would use the default memory management policy. The concept of data distribution for shared memory programming was first included in the SGI's MIPSpro compiler [29].

5.2.1. The `distribute` directive

We use a data distribution syntax to express data layout as a directive in OpenMP as follows.

```
#pragma omp distribute (dist-type-list: variable-list) [location (m:n)]
```

"`dist-type-list`" is a comma-separated list of distribution types for the corresponding array dimensions in the `variable-list`. Variables in the `variable-list` should have the same dimensions that match with the number of distribution types listed. Possible distribution types include "`BLOCK`" for a block distribution across a list of locations given

in the `location` clause, and "∗" for non-distributed dimension. If the `location` clause is not present, it means all locations are to be used for the distribution. Other types of data distribution are also possible, but we do not discuss them here for the sake of simplicity.

The distributed data still keeps its global address and is accessed in the same way as other shared data in the program. With distributed data, the user can control and manage shared data to improve data locality in OpenMP programs. The following example shows how the first dimension of array *A* gets distributed across all locations.

```
double A[N][N];
#pragma omp distribute (BLOCK,∗: A) location (0:NLOCS-1).
```

5.2.2. The `onloc` clause

To achieve greater control of task-data affinity, we can map implicit and explicit tasks in OpenMP to locations based on either an explicit location number or the placement of distributed data. For the latter case, we introduce the "`onloc (variable)`" clause to assign a task to a location based on where the data resides. Only distributed variables are meaningful in the `onloc` clause. The variable can be either an entire array (for the `parallel` construct) or an array element (for the `task` construct). In the following example,

```
#pragma omp task onloc (A[i])
  foo (A[i]);
```

the location for the task execution is determined by the location of "`A[i]`". When the `onloc` clause applies to a parallel region, it implies that the list of locations for the region is derived from the list associated with the distributed variable, as shown in the following example,

```
#pragma omp parallel onloc (A)
  {...}
```

Mapping of implicit tasks to locations for this case then follows the same rule as discussed for the `location` clause. Compared to `location`, the `onloc` approach can achieve better control of task-data affinity.

The following restriction applies to the `location` and `onloc` clauses: At most one `onloc` or `location` clause can appear on the `parallel` or `task` directive. If a non-distributed variable is used in the `onloc` clause, it will be treated as if no such a clause is specified and an implementation may print warning messages.

5.3. Implementation and experiments

We are currently developing the proposed location feature in the OpenUH compiler [23]. The OpenUH compiler is a branch of Open64 compiler and is used as a research infrastructure for OpenMP compiler and tools research. Besides the translation of OpenMP constructs into the appropriate runtime calls, the focus is on the development of OpenMP runtime

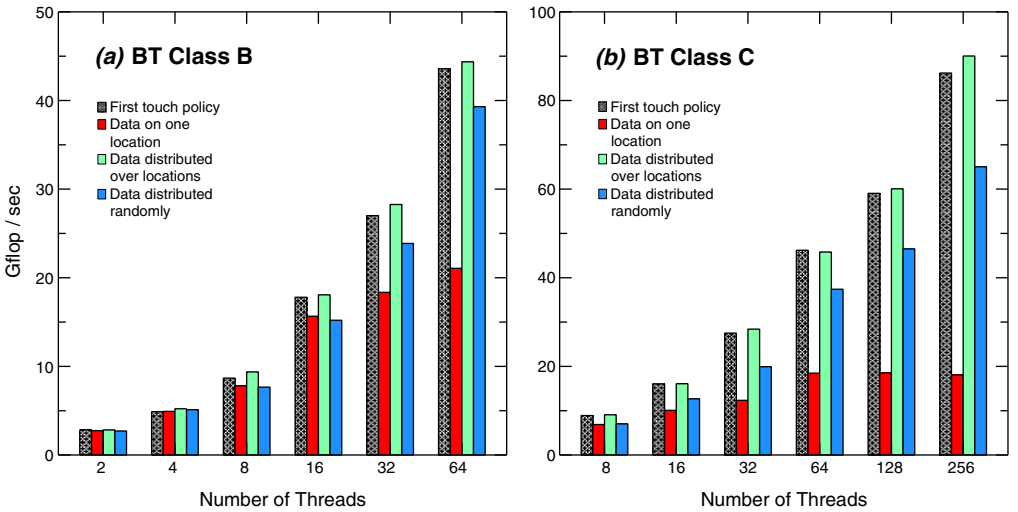

Fig. 9. BT performance (Gflop/s) from different data layouts.

infrastructure to support location and data layout management. Our current approach relies on the `libnuma` library [18] available under Linux for managing distributed resources. For detailed description, please refer to [12].

Since our implementation is not yet complete, we are not able to demonstrate the performance gains using the proposed features. However, as an experiment, we revised the BT benchmark from the NPB suite manually to illustrate the importance of different data layouts on performance. Results collected on the SGI Altix 4700 system are shown in Fig. 9 for BT Class B and Class C, higher Gflop/s indicating better performance. The four different columns for each thread count show the performance of four different data layout policies: first touch, all data on one location, layout based on data access pattern, and random placement. From this experiment, we observe significant performance impact from different data layouts on the NUMA system, especially for larger data sets. In general, the layout based on data access pattern (shown in the third column) produces the best performance, followed by first touch (first column). The single location and random placement performed worse than the other two. Although the case with data distributed over locations is similar to the case using the first touch policy, we believe that it can be improved further by optimizing data layout with the data access pattern, which would be possible after our compiler implementation is completed.

Although the goal of the work is to introduce a minimal set of features via compiler directives to achieve desired performance gain, it is anticipated that some amount of code transformations could be required (either by compiler or by hand) in order to achieve high performance. For instance, the static data layout in the current proposal would not work well for applications that alter data access patterns in different computing phases. One possible solution would be to introduce the concept of *data redistribution* for different phases. However, the benefit of data redistribution could be overshadowed by the associated cost. An alternative approach would involve the use of duplicated datasets with different data layouts suitable for each computing phase. The obvious drawback of this approach is the increased memory footprint. For examining the effectiveness of the proposed locality extensions, we would need more experiments with real applications.

6. Related work

MPI and OpenMP are the two mostly studied parallel programming models over the years. There are many publications on the two programming models and also on the hybrid approach. A good summary of the hybrid MPI + OpenMP parallel programming can be found in a tutorial given by Rabenseifner et al. [25]. The related publication [26] includes the performance study of the NPB-MZ hybrid codes on an AMD Opteron-based cluster and points out some of the keys for achieving good scalability. Shan et al. [28] described an analysis of performance effects from different programming choices for NPB and NPB-MZ on Cray XT5 platforms. They presented the benefit of the hybrid model on memory usage. Our current work extends the study to the Intel processor-based clusters in detail and provides better understanding of the effect from the underlying MPI library implementation.

Our previous work [27] compared the performance of the hybrid OVERFLOW code on three parallel systems. The current work gives a more comprehensive analysis of the hybrid approach based on numerical accuracy and convergence in addition to presenting the performance of the code on the current multi-core architectures. We also utilize the AKIE code to present a more *apples-to-apples* comparison of the hybrid and the pure MPI approaches.

SGI has included the concept of data distribution for shared memory programming in its MIPSpro compiler [29]. Two earlier proposals by Benkner [3] and Bircsak [5] made another attempt to extend OpenMP for ccNUMA machines. These efforts were based on the data distribution concepts from High Performance Fortran (HPF). Our extensions to OpenMP are built on the concept of location, which follows similar concepts in the modern HPCS languages and provides a virtual layer for expressing the NUMA effect. This design choice is intended to preserve the simplicity of the OpenMP programming.

7. Conclusions

In summary, we have presented case studies of the hybrid MPI + OpenMP programming approach applied to two pseudo-application benchmarks and two real-world applications, and demonstrated benefits of the hybrid approach for performance and resource usage on three multi-core based parallel systems. As the current high performance computing systems are pushing toward petascale and exascale, per-core resources, such as memory, are expected to become smaller. Thus, the reduction on the memory usage and overhead associated with MPI calls and buffers resulting from the hybrid approach will become more important. Our studies have also shown the usefulness of the hybrid approach for improving load balance and numerical convergence. Although modern parallel languages such as the HPCS languages are maturing, the more traditional MPI + OpenMP approach is still viable for programming on a cluster of SMPs.

There are two primary limitations that impede a wider adoption of hybrid programming: no well-defined interaction between MPI processes and OpenMP threads, and limited performance of OpenMP, especially on ccNUMA architectures. There is an on-going effort in the MPI community to improve the interface with threads in MPI 3 [10]. In this paper, we have described our approach to extending OpenMP with the concept of location to improve the performance of the resultant programs on ccNUMA systems. We also presented the syntax of language extensions to express task-data affinity. For future work, it would be useful to conduct more detailed studies on the performance sensitivity of applications to memory latency (such as via hardware performance counters) and MPI parameters. We would like to refine the location concept design and verify the effectiveness of the extensions by studying the performance of test cases and benchmarks, for example, to

investigate different thread-to-location mappings on performance. Another area of interest is to extend the location concept to platforms other than NUMA SMPs.

Acknowledgments

The authors thank Jahed Djomehri for providing the CART3D results presented here, Johnny Chang and Robert Hood for their valuable discussion and comments, the NAS HECC staff for their support in conducting these performance measurements on the NAS systems, and anonymous reviewers for their valuable comments on the manuscript. The work was partially supported by the National Science Foundation grant CCF-0702775 with the University of Houston.

References

[1] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele, Project Fortress: A Multicore Language for Multicore Processors. Linux Magazine, 2007.
[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, M. Yarrow, The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.
[3] S. Benkner, T. Brandes, Exploiting data locality on scalable shared memory machines with data parallel programs, in: Proceedings of the Euro-Par 2000 Parallel Processing, Munich, Germany, 2000, pp. 647–657.
[4] M.J. Berger, M.J. Aftosmis, D.D. Marshall, S.M. Murman, Performance of a new CFD flow solver using a hybrid programming paradigm, Journal of Parallel and Distributed Computing 65 (4) (2005) 414–423.
[5] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, C.D. Offner, Extending OpenMP for NUMA machines, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, TX, 2000.
[6] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the chapel language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.
[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grotho, A. Kielstra, V. Saraswat, V. Sarkar, C.V. Praun, X10: an object-oriented approach to non-uniform cluster computing, in: Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, ACM SIGPLAN, 2005, pp. 519–538.
[8] R. Diaconescu, H. Zima, An approach to data distributions in chapel, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 313–335.
[9] J. Djomehri, D. Jespersen, J. Taft, H. Jin, R. Hood, P. Mehrotra, Performance of CFD applications on NASA supercomputers, in Proceedings of the International Conference on Parallel Computational Fluid Dynamics, 2009, pp. 240–244.
[10] R.L. Graham, G. Bosilca, MPI forum: preview of the MPI 3 standard. SC09 Birds-of-Feather Session, 2009. Available from: <http://www.open-mpi.org/papers/sc-2009/MPI_Forum_SC09_BOF-2up.pdf>.
[11] C. Hah, A.J. Wennerstrom, Three-dimensional flow fields inside a transonic compressor with swept blades, ASME Journal of Turbomachinery 113 (1) (1991) 241–251.
[12] L. Huang, H. Jin, Liqi Yi, B. Chapman, Enabling locality-aware computations in OpenMP, Scientific Programming 18 (3-4) (2010) 169–181 (special issue).
[13] L. Hochstein, V.R. Basili, The ASC-alliance projects: a case study of large-scale parallel scientific code development, Computer 41 (3) (2008) 50–58.
[14] L. Hochstein, F. Shull, L.B. Reid, The role of MPI in development time: a case study, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, Texas, 2008.
[15] H. Jin, R.F. Van der Wijgnaart, Performance characteristics of the multi-zone NAS parallel benchmarks, Journal of Parallel and Distributed Computing 66 (2006) 674–685.
[16] H. Jin, R. Hood, P. Mehotra, A practical study of UPC with the NAS parallel benchmarks, in: Proceedings of the 3rd Conference on Partitioned Global Address Space (PGAS) Programming Models, Ashburn, VA, 2009.
[17] D. Kaushik, S. Balay, D. Keyes, B. Smith, Understanding the performance of hybrid MPI/ OpenMP programming model for implicit CFD codes, in: Proceedings of the 21st International Conference on Parallel Computational Fluid Dynamics, Moffett Field, CA, USA, May 18–22, 2009, pp. 174–177.
[18] A. Kleen, An NUMA API for Linux, SUSE Labs, 2004. Available from: <http://www.halobates.de/numaapi3.pdf>.
[19] G. Krawezik, F. Cappello, Performance comparison of MPI and three OpenMP programming styles on shared memory Multiprocessors, in: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, 2003, pp. 118–127.
[20] R.H. Nichols, R.W. Tramel, P.G. Buning, Solver and turbulence model upgrades to OVERFLOW 2 for unsteady and high-speed applications, in: Proceedings of the 24th Applied Aerodynamics Conference, volume AIAA-2006-2824, 2006.
[21] R. Numrich, J. Reid, Co-array fortran for parallel programming, In ACM Fortran Forum 17 (1998) 1–31.
[22] OpenMP Architecture Review Board, OpenMP Application Program Interface 3.0, 2008. Available from: <http://www.openmp.org/>.
[23] The OpenUH compiler project. Available from:<http://www.cs.uh.edu/openuh>.
[24] Pleiades Hardware. Available from: <http://www.nas.nasa.gov/Resources/Systems/pleiades.html>.
[25] R. Rabenseifner, G. Hager, G. Jost, Tutorial on hybrid MPI and OpenMP parallel programing, in: Supercomputing Conference 2009 (SC09), Portland, OR, 2009.
[26] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programing on clusters of multi-core SMP nodes, in: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2009), Weimar, Germany, 2009, pp. 427–436.
[27] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, R. Biswas, Scientific application-based performance comparison of SGI Altix 4700, IBM Power5+, and SGI ICE 8200 supercomputers, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008.
[28] H. Shan, H. Jin, K. Fuerlinger, A. Koniges, N. Wright, Analyzing the Performance Effects of Programming Models and Memory Usage on Cray XT5 Platforms, Cray User Group (CUG) meeting, Edingburgh, United Kingdom, 2010.
[29] Silicon Graphics, Inc., MIPSpro (TM) Power Fortran 77 Programmer's Guide, Document 007-2361, SGI, 1999.
[30] The Top500 List of Supercomputer Sites. Available from: <http://www.top500.org/lists/>.
[31] The UPC Consortium, UPC Language Specification (V1.2), 2005. Available from: <http://www.upc.gwu.edu/documentation.html>.
[32] R.F. Van der Wijgnaart, H. Jin, The NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.
[33] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken, Titanium: a high-performance java dialect, in: Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.