基于 agent 技术的并行构件组装及性能优化方法研究 *

彭云峰，刘家磊，郭　磊

(安阳师范学院 软件学院, 河南 安阳 455000)

摘　要：为更好地组装并行构件程序和进行性能优化工作，设计和使用了不同的软件 agent。构件连接 agent 负责构件接口的粘合和数据重分布。构件执行 agent 和资源管理 agent 相互协作，把构件部署在满足要求的计算节点上。定义了 4 种不同的构件自适应策略。不同的构件自适应 agent，构件执行 agent 和资源管理 agent 相互合作，针对平台计算资源的不同情况，完成构件的自适应过程，提高了构件的性能。资源管理 agent，负载探测 agent 和构件执行 agent 相互合作，完成负载均衡工作，提高了整个计算平台的性能和吞吐量。在异构计算机集群上的相关实验证明了所提出的基于 agent 技术的并行构件组装及性能优化方法的有效性。相比传统的性能优化方法，基于 agent 技术的方法使用灵活，并且具有性能上的优势。

关键词：并行构件；agent 技术；性能优化；负载均衡

中图分类号：TP311　　doi: 10.19734/j.issn.1001-3695.2020.07.0250

Research of parallel component composition and performance optimization based on agent technology

Peng Yunfeng, Liu Jialei, Guo Lei

(*School of Software Engineering*, *Anyang Normal University*, *Anyang Henan* 455000, *China*)

Abstract: In order to better assemble and optimize the performance of parallel component programs, this paper designed and used different software agents. The component connection agents are responsible for the component interface bonding and data redistribution. The component execution agents and resource management agents cooperate with each other to deploy the components on the computing nodes that meet the requirements. This paper defined four adaptive strategies. Different component adaptive agents, component execution agents and resource management agents cooperate with each other to complete the adaptive process of component and improve the performance of component. Resource management agent, load detection agent and component execution agent cooperate with each other to complete the load balancing work and improve the performance and throughput of the whole computing platform. Experiments on heterogeneous computer clusters demonstrate the effectiveness of the proposed parallel component assembly and performance optimization methods based on agent technology. Compared with the traditional performance optimization methods, the method based on agent technology is flexible and has performance advantages.

Key words: parallel component; agent technology; performance optimization; load balance

0　引言

并行构件程序的基本组成单元是并行构件。并行构件内部封装了并行计算的程序代码，一般要被部署在异构的计算平台上执行。在运行框架的支持下，这些构件之间通过共享内存、进程间通信或网络通信交互。构件自身可能具有不同的并行度。它们可以以多进程的方式并行执行，也可以采用多线程的方式执行，甚至以两者混合的方式执行。构件执行模式的不同导致出现了 SCMD 和 MCMD 两种并行构件编程模型。并行构件程序往往混合使用这两种编程模型。构件的实现代码一般是 C、C++、Java、Fortran 等编程语言和 MPI[1]、OpenMP[2]等并行化指导语句。在构件程序组建时，往往遇到不同语言编写的构件需要相互调用、交互的情况。Babel[3]提供了高性能计算常用的程序设计语言的互操作功能。以 Babel 为中介，可以实现不同语言编写的并行构件的互操作。但是，如果两个相互调用的构件接口定义不同，具有不同的接口名、输入参数或返回值，就无法通过 Babel 进行连接。传统的并行软件性能优化往往采用性能预测、自适应和负载均衡的方法。最常见的是根据运行资源的变化，改变某个构件的并行度，或者把高负载计算节点上的构件迁移到低负载的节点上[4~7]。已有的并行构件程序性能优化方法仅仅是将传统并行软件性能优化方法固化在并行构件运行框架内[8]。并行构件运行框架负责构件的连接和数据交互。每一种并行构件框架定义了自己所能提供的构件交互模式。这种由框架定义的构件交互和在其基础上的性能优化机制往往具有一定的局限性，不能根据构件程序的特点和运行时的情况进行变化，不能从构件自身的角度出发，灵活地进行并行构件程序性能优化。

agent 技术在分布式系统中被广泛应用[9][10]。agent 一般可看做一个计算实体，可以由软件和硬件构成。在分布式系统中，agent 一般具有自主性、交互性、反映性和主动性[11]。agent 能感知外部环境和自身内部状态的变化，并随之作出相应的反映。agent 之间能够相互交互，协同工作。agent 能够主动地把信息提交给特定的目标。并行构件程序大多具有分布式的特点。在并行构件程序连接和运行时使用 agent 技术具有天然的优势。本文提出，在构件程序的组建和运行阶段使用多种不同的 agent，来辅助构件的连接和运行，提高构件

收稿日期：2020-07-30；**修回日期：**2020-09-01　　**基金项目：**河南省重点研发与推广专项(19219239212，202102210152)；河南省高等教育教学改革研究与实践项目(2019SJGLX386)

作者简介：彭云峰(1982-)，男，河南安阳人，讲师，博士，主要研究方向为并行计算、软件工程(peng_ayit@163.com)；刘家磊(1982-)，男，河南开封人，讲师，博士，主要研究方向为云服务可用性、云计算；郭磊(1976-)，男，河南安阳人，副教授，硕士，主要研究方向为软件工程.

的运行性能。在构件程序组建阶段，可以使用构件连接 agent 对接口不匹配的并行构件进行粘合，同时支持 M×N 数据重分布[12]。在构件进行初次调度运行时，可以通过资源管理 agent 查找满足构件运行要求的计算节点。构件执行 agent 负责构件到计算资源的部署。在构件的运行阶段，采用构件自适应 agent，来针对运行平台的资源情况进行构件的自适应，提高构件的性能。在进行负载均衡时，需要负载探测 agent 收集各个节点上的负载情况。构件执行 agent 完成负载迁移的过程。这些 agent 都将由 agent 管理系统按需生成并分配。每一种类型的 agent 已经按照功能被定义为一个 C++的类。在需要生成 agent 时，agent 管理系统将生成特定 agent 类的一个实例。

1　接口粘合和数据重分布

在构件程序组建者连接构件时，会指定哪些构件需要通过哪些接口两两连接以及每个构件初始运行时的并行度。对于需要进行组合连接的两个构件，首先需要 agent 管理系统为两个构件各分配一个构件连接 agent。构件连接 agent 生成后，它能够主动探测两个构件的接口情况，根据接口的名称和参数，判断能否用特定的粘合代码进行粘合。Babel 的多语言互操作机制是把相互连接的两个构件接口都转换成 SIDL[3] 形式，从而实现接口的匹配连接。但是如果两个接口的接口名、输入参数或返回结果类型不一样，即使都转换成了 SIDL 形式，它们也无法匹配。

如图 1 所示，在 Babel 的基础之上，本文提供了接口名、输入参数以及返回结果类型不同的接口的粘合机制。这种粘合的前提是构件程序的组建者在连接构件时，就已经确定这两个构件从功能上来说具有相互的调用关系，在接口匹配后，一定能够连接运行。在接口粘合的过程中，发起调用的构件的一个特殊的 agent(本文称之为构件连接 agent)，负责调用 Babel，将发起调用的构件接口转换成 SIDL 形式。被调构件的构件连接 agent 调用 Babel，将被调构件接口转换成 SIDL 形式。如果发现两个接口不匹配，则采用以下粘合方式：被调构件的构件连接 agent 把被调构件的 SIDL 形式的接口名改为发起调用者的 SIDL 接口名形式。这样调用的发起者就能够识别出与之匹配的被调构件。对于输入参数，在将它们传递给被调构件时，需要发起调用的构件的构件连接 agent 使用粘合代码，将其转换成被调构件接口中输入参数的个数和类型，以便于被调构件使用。如果两个构件的参数个数不相等，那么任何一方多出来的参数将被赋值为 NULL。在被调构件运行代码的最后，同样需要被调构件的构件连接 agent 通过粘合代码将它的返回值类型改为发起调用的构件接口的返回值类型，以正确地返回发起调用的构件。

同时，在构件程序组建时，不同的构件连接 agent 相互协作，还能够实现 M╳N 数据重分布。在构件程序组建时，如果已知相互连接的两个构件分别运行在 M 个和 N 个进程上，且 M 不等于 N，就需要在两个构件之间进行数据的重分布操作。具体方法是将前一个构件的 M 个进程的运行结果收集起来，再分发给后一个构件的 N 个进程。数据的收集工作由前一个构件的构件连接 agent 实现，分发工作由后一个构件的构件连接 agent 实现。图 1 给出了的构件连接 agent 的工作过程，假设发起调用的构件 A 以 C+MPI 编程方式实现。在构件 A 的代码执行过程中，发起了对构件 B 的调用，该调用的接口名为 comp，输入参数为布尔类型的 isready 和字符类型的 optobject。此时，A 的构件连接 agent 将向 B 的构件连接 agent 发起查询，查询 B 提供的接口的情况。被调构件 B 使用 FORTRAN+MPI 编程实现。B 提供的接口名、参数、返回值类型和 A 都不相同。于是，A 的构件连接 agent 将通知 B 的构件连接 agent，启动粘合过程，最终把两个构件以接口匹配的方式连接起来，并且在连接过程中加入了数据的收集和分发操作，实现了 M×N 数据重分布。在构件程序组建阶段，将完成两个构件的接口粘合和数据重分布代码的生成。这些代码在在运行阶段构件相互调用时将会自动地执行。所以在粘合代码和数据重分布代码生成后，就可以由 agent 管理系统销毁两个构件连接 agent，释放它们所占用的资源。

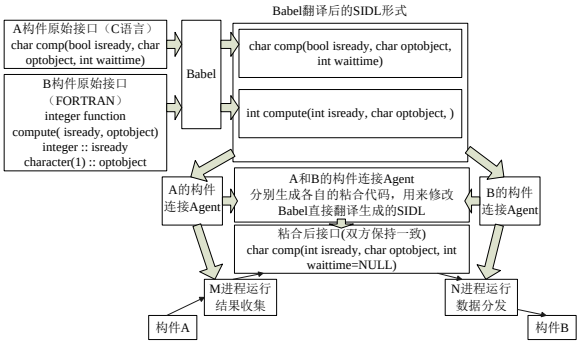

图 1　接口粘合和数据重分布

Fig. 1　Interface bonding and data redistribution

2　构件的部署

一旦一个构件程序被组建完毕，它的每个构件将被 agent 管理系统分配一个构件执行 agent。然后构件程序将被部署到计算平台上运行。一个构件程序在执行时，它的组成单元，即组成程序的各个构件之间可能有着数据依赖等依赖关系。只有那些被依赖构件已经执行完毕，依赖数据已得到，依赖关系已解决的构件才满足被执行的条件。构件程序的组建者需要给出构件间的依赖关系，然后在构件执行 agent 的生成过程中，把当前构件所依赖的其他构件的信息写入构件执行 agent 内的一个依赖关系表内。一个构件运行结束时，它的构件执行 agent 将向其他构件的构件执行 agent 发送一个消息，声明自己已经运行结束。这个消息将在各个构件执行 agent 之间进行传递，每个收到消息的 agent 检查这个已结束构件是否被自己管理的构件所依赖。如果依赖关系存在，收到消息的 agent 将把已结束构件的信息从自己的依赖关系表内删除。当一个构件的所有依赖关系都被解决，它的构件执行 agent 将开始为它寻找合适的计算资源进行部署运行。

经典的 CCA 并行构件运行方式是在每个计算节点上都保留所有构件的一个备份[13]。如果需要指定部署构件 A 到计算节点 B 上运行，需要构件程序组建者把这个部署信息写在 rc 启动文件内。这样构件 A 在启动运行时只会启动节点 B 上的实例代码，其他节点上的 A 的备份不被启动。这种方法使构件的部署工作变得比较简单。但是，当需要部署运行的构件程序比较多时，为所有的构件程序在所有节点上都保留一个备份将占用大量的系统资源。为更好地进行构件的部署，本文选取计算平台中一个性能较好的多核(16 核，内存 32G)节点作为根节点。所有的构件程序在首次运行前，都被加载到这个根节点上，在根节点上完成构件的连接工作(包括前面说明的接口粘合和数据重分布代码的生成)。构件连接完成后，将为每个构件分配一个构件执行 agent。一个构件的所有依赖关系都被解决后，它的构件执行 agent 将开始这个构件的部署运行。

构件执行 agent 部署运行一个构件时，将根据构件组建者预先定义的资源需求为它分配计算资源。计算资源主要包括计算节点的 CPU 核数、内存大小和网络带宽。为更好地管理计算节点上的计算资源，需利用 agent 管理系统为每个计算节点生成一个资源管理 agent。资源管理 agent 能够自主地探测计算节点的各种资源情况。在为构件分配资源时，构件

执行 agent 向离自己最近的节点发送资源管理 agent 能够识别的消息，消息内容是当前要部署的构件的资源需求信息。资源管理 agent 收到后，如果自身管理的节点可以满足构件的要求，则把自己的信息写入这个消息，然后传递回构件执行 agent。如果该资源管理 agent 发现，该资源需求需要本节点联合其他计算节点才能满足，它就会把自己的信息写入消息，然后把消息传递给下一个节点的资源管理 agent。如果收到消息的资源管理 agent 发现自己所在节点无法满足构件需求，它将把该消息直接传递给下一个节点的 agent。这个消息传递过程一直持续到已经登记过信息的节点资源总和满足构件的资源需求才结束。最后一个登记自己信息的节点将把该消息传递回待部署构件的构件执行 agent。构件执行 agent 将构件部署到已登记的满足要求的计算节点上开始运行。和已有的计算资源管理方法相比，这种 agent 之间传递消息，寻找合适的计算资源的方法减轻了被调度构件所在节点的负担，加快了查找满足条件的计算资源的速度。构件部署完毕后，构件执行 agent 和被它管理的构件是一一对应的关系。例如，如果构件 A 以 MPI 多进程的方式并行运行，那么每个 MPI 进程中都会运行着 A 的一个实例。每个 A 的实例都会有一个相应的 A 的构件执行 agent 的实例，分布在每个 MPI 进程所在的计算节点上。

由于各个构件的构件执行 agent 是独立运行的，这时可能会出现多个构件被部署到同一个节点上运行的情况。为防止某些节点负担过重，减少构件间对资源的争夺，需要构件程序组建者给构件程序指定一个调度优先级。构件程序的调度优先级分为低，中，高 3 种。构件的调度优先级初始化为它所属的构件程序的调度优先级，保存在构件执行 agent 中。每个节点的资源管理 agent 会维护一个待执行构件任务队列。队列中存放了已经部署到本节点的构件。当本节点有了空闲的 CPU 处理器资源时，将从这个队列中取出一个任务执行。如果一个计算节点任务队列中的任务超过了节点的 CPU 处理器核心数的 3 倍，就不再往该节点上部署新的构件。队列中的构件任务按照优先级排序。优先级相同的构件，选择那些预期运行时间较短的先进行调度，以提高系统的吞吐量。构件的输入数据规模和对应的历史运行时间被保存在一个构件信息库中。在构件程序组建后，根据实际输入数据的规模，在信息库中找到对应的历史运行时间，作为预期执行时间，存放到构件执行 agent 中。如果没有对应的历史数据，将由构件组建者根据经验给出一个运行时间的预估值。当一个构件进入待执行队列时，它的构件执行 agent 将启动一个计时器，若低优先级构件 12 小时后还未得到计算资源并执行，其优先级将变为中。若中优先级构件 12 小时后还未得到计算资源并执行，其优先级将变为高。这种动态地改变优先级的方式防止了某些低优先级构件长期得不到执行。图 2 给出了构件的部署过程示意图。

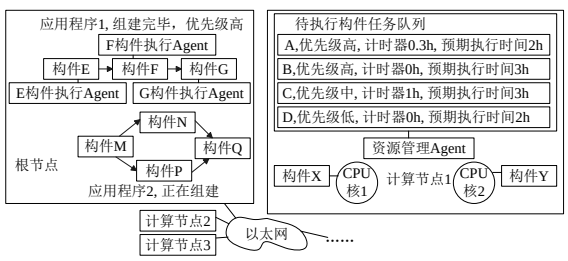

图 2　构件的部署

Fig. 2　Component deployment

3　并行构件的自适应

构件的自适应策略包括动态地改变构件的并行度、改变数据划分、构件迁移和改变实现。本文相应地定义了 4 种自适应 agent。构件程序组建者在选择构件组建程序时，要指定一个构件是否需要结合某种自适应策略。如果需要，agent 管理系统将为该构件生成针对不同策略的自适应 agent，在构件部署时和构件部署到同一个节点上。一个自适应 agent 内包含两部分内容，分别是触发自适应的事件和 agent 对事件的响应。

一般来说，触发构件并行度改变的事件是构件运行时间较长，可用资源(主要是 CPU 核数)较多的情况。假设构件被初始分配到了 m 个 CPU 处理器核心上运行。并行度自适应 agent 将对构件的运行时间计时。同时，它会向相邻的一个计算节点的资源管理 agent 发送负载查询信息，消息中包含已部署构件的节点信息。该资源管理 agent 检查自己的负载情况，如果高于某个负载阈值，则把该查询发送给下一个节点。如果资源管理 agent 发现自己所在节点的负载低于阈值，则把自己的信息写入查询消息，继续传递。这个过程持续到该消息中包含的低于阈值的节点的 CPU 处理器核心数总和满足并行度自适应的要求为止。为加快并行度自适应的过程，每个新的节点把自己的信息写入查询消息的同时，会根据消息中已部署构件的节点信息，选择一个离自己较近的节点，向该节点上的构件执行 agent 发送消息，要求它发送构件的拷贝给自己。该构件执行 agent 收到消息后，立即响应请求，为新加入的节点发送一个构件的拷贝，同时也发送一份构件执行 agent 和并行度自适应 agent 的拷贝。并行度自适应 agent 一旦发现构件的运行时间超过某个时间阈值，同时可用的 CPU 处理器核心数满足并行度自适应的要求，它将作出自适应的决策，暂停构件的执行，重新对未执行的数据进行分配，将其分配到 m+n 个 CPU 处理器核心上并行执行。这里负载阈值以及 n 的值都是通过构件程序组建者在构件运行前设定，并写入并行度自适应 agent 中。如果构件使用 MPI 多进程的方式并行运行，改变构件的并行度将通过 MPI _Comm _Spawn 操作实现。如果构件使用 OpenMP 或类似的共享内存方式并行运行，并行度的改变将通过 fork 操作实现。图 3 给出了并行度自适应 agent 的工作过程。

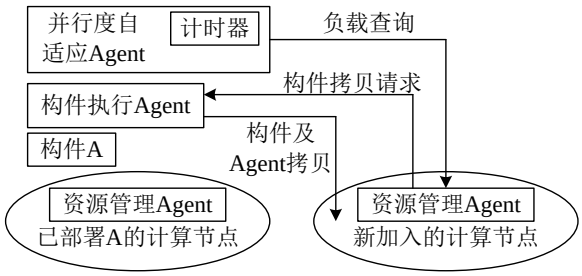

图 3　并行度自适应

Fig. 3　Adaptive parallelism

改变数据划分自适应策略的触发往往是由于并行执行的各计算节点处理速度不一致造成的。那些处理速度快，负载任务较轻的计算节点在运行时往往比其他节点较早地完成自己分配到的数据块。这时，其他节点还远远没有完成任务。数据划分自适应 agent 在每个计算节点完成自己分配到的任务后，将检测已完成的数据量和总体数据量的比值。如果这个比值小于 1/10，改变数据划分的自适应策略将被触发。数据划分自适应 agent 将暂停构件任务的并行执行，收集还未完成的数据任务，重新进行划分和执行。如果某次检测发现已完成的数据量和总体数据量的比值大于 1/10，本次构件程序运行的数据划分自适应功能将被关闭，本次构件程序运行的剩余部分不再进行数据量的检测。

构件迁移往往是由于构件运行的计算节点的某种故障失效造成的。在这种情况下，如果没有提前保存中间结果，构

件需要被部署到其他计算节点上重新执行。如果整个计算平台性能很不稳定，经常出现计算节点失效的情况，就需要在构件调度之前，由 agent 管理系统为其分配一个构件迁移自适应 agent。在构件迁移自适应 agent 中具有周期性地保存中间结果到外部文件的代码。同时，要选择较为稳定的节点作为备份节点，构件迁移自适应 agent 调用构件执行 agent 向备份节点发送一份构件的拷贝，包括构件执行 agent 和构件迁移自适应 agent 自身的拷贝。在运行过程中，如果计算节点因故障失效，可以取出中间结果，启动备份节点上的构件实例，继续完成构件任务。

在构件运行时改变实现的前提是已经具有功能相同，实现不同的构件版本。比如同样的数学运算功能，可能既有适合于普通服务器运行的通用版本，也有能够在 GPU 上加速处理的特殊版本。在运行过程中，改变实现自适应 agent 将主动探测各个节点的硬件情况。如果发现有包含 GPU 且负载较轻的节点，将暂停构件的执行，保存中间结果，然后调用构件执行 agent，将特殊版本的构件部署到 GPU 节点上继续运行。

在不同节点上的自适应 agent 和资源管理 agent 的控制下，完成 4 种不同的并行构件自适应过程。这种分布式的自适应方法和已有的集中调度的方法相比，减轻了作业控制节点的负担。同时，这种不同节点并行执行，共同完成自适应过程的方法，往往比已有的其他方法有较优的性能表现。

4　负载均衡

当需要开启整个系统的负载均衡机制时，构件程序的组建者(计算平台的用户)要定义 2 个参数，包括负载探测的周期和负载的上限。构件所在节点的负载情况需要考查 top 命令返回的 load average 与 cpu 核数的比值。负载的上限可以自定义，但一般以 0.7 作为阈值。每个节点上的资源管理 agent 周期性地探测它所在节点的负载情况。在探测时如果发现本计算节点的负载大于上限，则通知本节点上所有构件的构件执行 agent，暂停构件的执行，把中间结果保存在构件执行 agent 中。资源管理 agent 请求 agent 管理系统为其生成一个负载探测 agent。负载探测 agent 将在计算平台中自主移动，收集各个节点的负载信息，然后返回发起负载探测的源节点，更新源节点上保存的负载信息。这种使用 agent 收集平台上不同节点负载信息的方法，极大地减轻了源节点的负担，同时具有较高的性能表现。本文一般考虑计算节点的 3 个资源利用指标，即 CPU 利用率、内存利用率和网络带宽。假设平台有 n 个节点，负载探测 agent 将返回每个节点的这 3 个指标数据。表 1 是某次运行负载探测 agent 返回的信息。

表 1　负载探测 agent 返回的信息

Tab. 1　Information collected by load detection agents

ID	IP	CPU 利用率	内存利用率	网络带宽
1	192.168.0.3	30%	10%	21M
2	192.168.0.7	80%	70%	17M
3	192.168.0.66	50%	80%	23M
4	192.168.0.5	10%	20%	20M
5	192.168.0.21	90%	30%	19M

从表 1 可以知道，本次负载探测共经过了 5 个节点，表中记录了这 5 个节点的 ID、IP 地址和 3 个指标数据。由于这 3 个指标数据是在某一特定时间段内，基于不同的硬件收集的，直接用这 3 个指标来评价各个节点的负载情况不太准确。假设这 3 个指标对节点的负载影响具有相同的权重，接下来需要对这 3 个指标进行归一化处理。本文采取如下方法：

假设某次负载探测返回了 n 个节点的信息，$L_i$cpu 代表第 i 个节点原始的 CPU 利用率，$P_i$cpu 代表第 i 个节点归一后的 CPU 利用率。Xcpu 代表这 n 个节点中 CPU 利用率的最大值。Mcpu 代表这 n 个节点中 CPU 利用率的最小值。对 CPU 利用率进行归一化的公式如下：

$$P_i cpu = (L_i cpu - Mcpu)/(Xcpu - Mcpu) \tag{1}$$

$L_i$mem 代表第 i 个节点原始的内存利用率，$P_i$mem 代表第 i 个节点归一后的内存利用率。Xmem 代表这 n 个节点中内存利用率的最大值。Mmem 代表这 n 个节点中内存利用率的最小值。对内存利用率进行归一化的公式如下：

$$P_i mem = (L_i mem - Mmem)/(Xmem - Mmem) \tag{2}$$

$L_i$net 代表第 i 个节点原始的网络带宽，$P_i$net 代表第 i 个节点归一后的网络带宽。Xnet 代表这 n 个节点中网络带宽的最大值。Mnet 代表这 n 个节点中网络带宽的最小值。对网络带宽进行归一化的公式如下：

$$P_i net = (L_i net - Mnet)/(Xnet - Mnet) \tag{3}$$

按照这 3 个公式，用归一化后的资源负载指标代替原始的数据，可以得到表 2。

表 2　负载信息归一化

Tab. 2　Load balance normalization

ID	IP	归一化 CPU 利用率	归一化内存利用率	归一化网络带宽
1	192.168.0.3	0.25	0	0.67
2	192.168.0.7	0.875	0.86	0
3	192.168.0.66	0.5	1	1
4	192.168.0.5	0	0.14	0.5
5	192.168.0.21	1	0.29	0.33

这时，规定第 i 个节点的负载综合指标 L_i 为其归一化后的 3 个资源负载指标的和。即：

$$L_i = P_i cpu + P_i mem + P_i net \tag{4}$$

L_i 的值越大，代表第 i 个节点的负载越重。利用表 2 可以得到 L_1 到 L_5 的值，即本文评价这 5 个节点负载情况的最终指标数据。如表 3 所示，负载最重的是节点 3，综合负载指标为 2.5。负担最轻的是节点 4，综合负载指标为 0.19。此时，可以在构件执行 agent 的控制下，把源节点上的构件任务迁移到负载最小的节点即节点 4 上。

表 3　负载综合指标

Tab. 3　Load balance overall metric

Li	L1	L2	L3	L4	L5
负载综合指标值	0.92	1.735	2.5	0.19	1.62

在以上过程种，负载探测 agent 收集负载信息，资源管理 agent 负责负载数据的计算和比较，构件执行 agent 负责具体的构件任务迁移工作。为进一步优化整个过程，负载探测 agent 在向源节点返回负载数据后，就被 agent 管理系统删除，以节约系统资源。

5　实验

本文为检验提出的基于 agent 的并行构件程序组装和性能优化方法，进行了相关的实验。实验平台是一个异构的计算机集群，内部包含 32 台 SMP 服务器(CPU intel J3060，内存 8G)，2 台带有 GPU(GPU TESLA K80，内存 24G)的服务器，1 台 8 核的多核服务器(CPU intel i7-9700，内存 16G)和 1 台 16 核的多核服务器(CPU intel E5-2682V4，内存 32G)。各服务器安装的都是 Linux 操作系统(Fedora 32 Server)。各服务器之间通过以太网连接。

为模拟一个计算机集群日常的工作情况，本文选取用 cca-tools[14]并行构件开发工具包开发的 10 个不同的构件应用程序作为被测程序。表 4 是这 10 个程序的基本信息。这 10 个程序将在 16 核的多核服务器上进行连接组装，然后被构件执行 agent 部署到计算机集群上运行。

表 4　被测程序基本信息

Tab. 4　Applications tested

程序名称	主要功能	构件数量	编程语言	初始优先级
CCA_SPM	核磁共振图像处理	3	C+MPI+OpenMP+Python	高
Medical Control	医疗信息系统	11	Java+OpenMPI	低
CCA_Boast	油藏数值模拟	12	FORTRAN+MPI	低
Par_Protein	蛋白质分子预测	7	Python	中
MM5_Component	天气预报	8	C+MPI+OpenMP	中
Face Right	人脸识别	6	Python+MPI	高
Financial manager	金融信息处理	9	C++ and MPI	低
R_sensing	遥感图像处理	7	C++ and MPI	低
XMD_Component	分子动力学模拟	5	C+MPI	高
XMD_GPU	分子动力学模拟	5	C+MPI+OpenCL	中

本文首先进行的是构件程序组装实验。CCA_SPM 构件程序由 3 个构件组成。其中，第一个构件 Preprocessing 采用 C+OpenMP 的方式实现(初始分配 1 个进程)，用来对输入图像进行预处理。第二个构件 Model_estimate 采用 C+MPI+OpenMP 方式实现(初始分配 8 个进程)，用来进行模型估计。第三个构件 View 使用 Python 实现(初始分配 1 个进程)，用来显示图像处理的结果。在构件连接过程中，生成的 3 个构件连接 agent 通过探测构件的接口情况并相互交换信息，发现在 Preprocessing 和 Model_estimate 之间要进行 M╳N 数据重分布，在 Model_estimate 和 View 之间要进行接口粘合和 M╳N 数据重分布。在 Preprocessing 和 Model_estimate 进行连接时，Model_estimate 构件的构件连接 agent 生成数据重分布代码，使用 MPI_Scatter 用来把从 Preprocessing 接收到的数据从 1 个进程分发到 8 个进程上。在 Model_estimate 和 View 进行连接时，Model_estimate 发起调用的接口是 int Results_view(int MPicture[128][128][1][36])。MPicture 参数是经过处理以后的图像数组。View 提供服务的接口为 int View(int MRIP[128,128,1,36], int is_overlap)，MRIP 参数是输入的图像数组，is_overlap 参数取 1 时结果图像覆盖原图像，取 0 时不覆盖，默认不覆盖。按照本文提出的接口粘合方法，这两个接口将被转换为 SIDL 接口 int Results_view(int MRIP[128,128,1,36], int is_overlap=NULL)，使两个构件能够通过接口连接起来。构件连接 agent 为 Model_estimate 构件生成数据重分布代码，使用 MPI_Gather 把 8 个进程上的图像数据处理结果收集到 1 个进程上。CCA_SPM 构件程序的组装过程使用了本文提出的构件连接 agent，在构件组装过程中生成接口粘合和数据重分布代码，完成了接口的匹配连接和数据重分布。

接下来本文进行了构件的自适应机制的相关实验。为检验本文提出的构件自适应机制的效果，本文在 MM5_Component 构件程序中选取一个构件 Background 加入了本文提出的并行度自适应机制。作为对比，本文手工实现了一个具有 Concerto[15]的自适应功能的 Background 构件。Concerto 的自适应机制的特点是构件运行的并行度并不由构件程序组建者指定，而是在运行前进行一次资源探测，根据可用资源的状况，确定一个构件运行的并行度，此后在构件运行过程中一直以此并行度执行。而本文提出的并行度自适应机制是由构件程序组建者先指定一个构件的运行并行度。然后在运行过程中再根据实际可用资源的情况进行动态的并行度变化。很明显，如果构件程序组建者能够指定较为合适的并行度，比如和 Concerto 经过资源探测确定的并行度相等或接近，那么本文提出的方法能够在运行中反映资源的变化，取得比 Concerto 方法更好的性能。

图 4 是对 Background 构件进行测试的结果。在图 4 的实验中，本文在构件部署时，只启动特定个数的 SMP 节点，分别为 2，4，8 和 16 台 SMP 服务器。然后在构件运行过程中，再启动其他的 SMP 节点。Concerto 版本的构件初始探测得到的并行度和本文提出的优化机制中构件程序组建者指定的初始并行度都是 2，4，8 和 16。本文提出的优化机制在运行过程中由于能够随节点的增加，动态增加并行度，所以取得了更好的性能。

本文在 MM5_Component 构件程序中选取构件 VERinter 加入了本文提出的改变数据划分自适应机制。已有的并行构件程序性能优化方法较少涉及动态地改变数据划分，所以本文采用和未加入任何性能优化机制的原始版本的 VERinter 进行对比。同时，为了更好地模拟实际计算机集群上的运行情况，本文在运行 MM5_Component 构件程序，对 VERinter 进行性能测量的同时，在计算平台上同时部署运行其他 9 个构件程序。由于整个集群上的负载并不均衡，有部分节点加载构件较多，性能下降，造成并行执行 VERinter 构件的不同节点，负载任务较轻的计算节点在运行时比其他节点较早地完成自己分配到的数据块，改变数据划分自适应机制被触发。未执行的数据被重新划分，有效地利用了高速节点的处理能力，提高了构件程序的性能。图 5 是对 VERinter 构件进行实验的结果。从图 5 中可以看出，本文提出的方法能够提高构件的性能，并且在输入数据的数据量比较大时，这种数据重新划分相对于原始版本带来的性能提升更加明显。

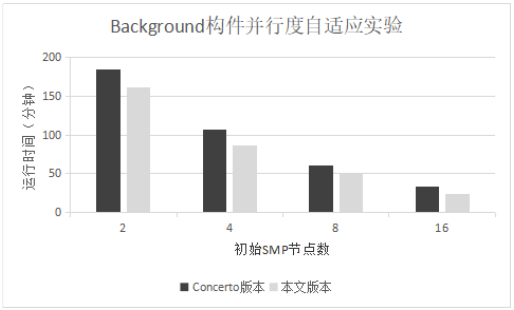

图 4　并行度自适应测试

Fig. 4　Adaptive parallelism test

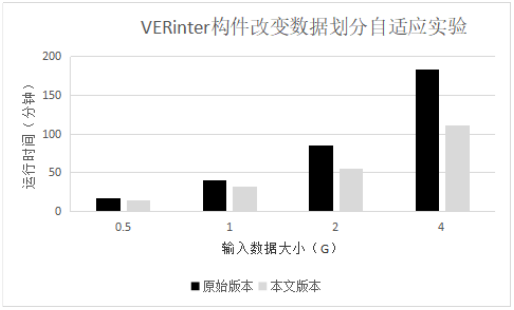

图 5　数据重新划分测试

Fig. 5　Data repartition test

本文在 MM5_Component 构件程序中的 8 个构件中都加入了本文提出的构件迁移自适应机制，然后单独运行 MM5_Component 构件程序，在运行过程中手工关闭单个的计算节点。构件迁移自适应 agent 在单个节点失效后，能够启动备份节点，恢复中间结果，完成构件程序的处理任务。

为对本文提出的改变实现自适应机制进行测试，本文选取 MM5_Component 构件程序中的 Topographic 构件准备了两个不同的版本。除了 C+MPI 实现的普通版本外，还实现了一个基于 GPU 的 C+OpenCL 版本。作为对比，本文手工实

现了一个具有 ICENI[16] 的性能预测和自适应功能的 Topographic 构件。ICENI 在构件运行前预测它的不同版本在当前可用资源上的性能，然后选择性能较好的版本运行。一旦选定版本，在运行过程中不再进行版本的改变。本文提出的改变实现自适应机制，直接指定基于 GPU 的 C+OpenCL 版本为高速版本，不需要进行性能预测。只要系统中具有负载较轻的 GPU 节点，就调用该版本构件到 GPU 上运行，省去了性能预测的开销，比 ICENI 版本具有更好的性能表现。同时，在现实的计算机集群平台上，具有特殊加速功能的硬件在构件被初次部署时经常会出现不可用的情况。在本实验中，为了模拟这种情况，本文先把构件程序 XMD_GPU 调度到唯一的 GPU 计算节点上运行。然后启动 Topographic 构件的部署和运行。对于 ICENI 版本的程序，Topographic 构件仍会被部署到 GPU 计算节点上。但是它需要等待 XMD_GPU 执行完毕才能开始自己的工作任务。而如果使用本文提出的改变实现自适应机制，在首次部署 Topographic 时，由于 GPU 计算节点被 XMD_GPU 占用，Topographic 会选择 C+MPI 实现的普通版本在其他节点上运行。同时它将周期性地探测 GPU 计算节点的可用情况。在 GPU 节点空闲时，暂停普通版本的 Topographic 构件执行，提取中间结果，选择 C+OpenCL 版本的 Topographic 构件调度到 GPU 上运行。图 6 显示了在构件首次部署时 GPU 节点不可用情况下，ICENI 版本和带有本文提出的改变实现自适应机制的 Topographic 构件在不同输入数据规模下的性能情况。显然，在实际的计算机集群上，本文提出的改变实现自适应机制具有比 ICENI 版本更好的性能。

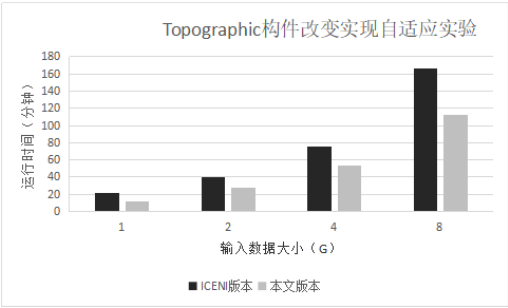

图 6　改变构件实现测试

Fig. 6　Component realization change test

为检验本文提出的负载均衡方法的有效性，本文对使用传统的基于集中式控制的负载均衡机制的系统和开启了本文提出的负载均衡机制的运行平台进行了对比。对集中式的负载均衡系统，负载探测、负载策略生成和实际的负载迁移工作都在根节点的控制下完成，根节点的任务较重，负载均衡工作的执行效率相对较低。对使用本文提出的负载均衡机制的系统，本文定义负载探测的周期为 20 分钟，负载的上限为 0.7。在这两种作为对比的运行环境下，本文分别组装提交了前面选取的 10 个构件程序中的 2 个、4 个、8 个和 10 个，相应的构件任务数目分别为 14 个、33 个、63 个和 73 个。图 7 给出了负载均衡对比实验的结果。从图 7 可以看出，本文提出的负载均衡机制能够合理地均衡各个计算节点上的负载，减少了所有构件任务总的运行时间，提高了系统的吞吐量。并且，负载均衡对系统性能的提升效果在构件任务的个数较多，整个系统的负载较重时更为明显。本文提出的负载均衡机制，主要通过各个计算节点上分布的资源管理 agent，待迁移构件的构件执行 agent，以及自主行动的负载探测 agent 相互配合，共同完成整个负载均衡工作。相对于集中式控制的负载均衡机制，具有更高的性能表现。同时，本文提出的负载均衡机制也有一定的性能代价。负载的探测和构件任务的迁移需要一定的时间和计算资源开销。在整个系统加载的构件程序个数为 8 个和 10 个，也就是构件任务数为 63 个和 73 个时，开启整个系统的负载均衡功能将给系统整体性能带来较大的提升。这时维持负载均衡的代价相对来说就比较小了，负载均衡所带来的收益相对较大。

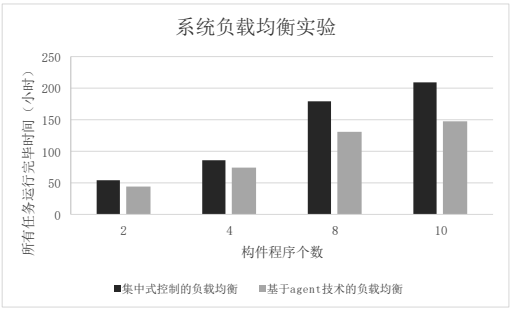

图 7　负载均衡测试

Fig. 7　Load balance test

6　结束语

本文在对已有的并行构件性能优化技术进行分析的基础上，结合 agent 技术的特点和优势，提出基于 agent 技术的并行构件程序组装和性能优化方法。预先把具有特定功能的 agent 定义为一些 C++类，在需要使用时通过 agent 管理系统生成特定类的实例，可以生成具有不同功能的 agent。构件连接 agent 通过调用 Babel 工具对用不同语言编写的构件接口进行翻译，同时生成相应的粘合代码对接口名，参数或返回值类型不匹配的接口进行粘合。构件连接 agent 还能够根据需要生成支持 M×N 数据重分布的粘合代码。构件执行 agent 管理构件间的依赖关系。它通过和资源管理 agent 交互，收集满足构件运行要求的计算资源的信息，将构件部署到特定的计算资源上。构件程序组建者指定构件程序的初始调度优先级。构件执行 agent 管理构件的优先级。计算节点的资源管理 agent 维护一个待执行构件任务队列。等待执行的构件任务按优先级和预期执行时间依序执行。为提高并行构件程序的性能，本文提出在自适应 agent、构件执行 agent 和资源管理 agent 的相互协作下，能够完成动态地改变构件的并行度、改变数据划分、构件迁移和改变实现 4 种不同的构件自适应过程。针对异构集群平台可能出现的负载不均衡情况，本文提出通过资源管理 agent、负载探测 agent 和构件执行 agent 的共同合作，能够把高负载的计算节点上的构件任务迁移到低负载的节点上执行，提高整个系统的吞吐量和性能。通过对异构的计算机集群上的 10 个并行构件程序进行组装和运行实验，证明了本文提出的方法的有效性。相比传统的性能优化方法，基于 agent 技术的方法使用灵活，并且具有性能上的优势。

参考文献：

[1] Lawrence Livermore National Laboratory (LLNL) . Message Passing Interface (MPI) [EB/OL]. (2020) [2020-07-27]. https://computing. llnl. gov/tutorials/mpi/.

[2] OpenMP Architecture Review Board. Home-OpenMP [EB/OL]. (2020) [2020-07-27]. https://www. openmp. org/.

[3] Lawrence Livermore National Laboratory (LLNL) . Babel homepage [EB/OL]. (2020) [2020-07-27]. https://computing. llnl. gov/projects/babel-high-performance-language-interoperability/#page=home.

[4] Ono K, Uchida T. High-Performance parallel simulation of airflow for complex terrain surface [EB/OL]. (2020) [2020-07-27]. https://www. hindawi. com/journals/mse/2019/5231839/.

[5] Ho M T, Zhu Lianhua, Wu Lei, *et al.* A multi-level parallel solver for

rarefied gas flows in porous media [J]. Computer Physics Communications, 2019, 234 (1): 14-25.

[6] Egorova M. S. , Dyachkov S. A. , Parshikov A. N. , *et al.* Parallel SPH modeling using dynamic domain decomposition and load balancing displacement of Voronoi subdomains [J]. Computer Physics Communications, 2019, 234 (1): 112-125.

[7] Baiges J, Martinez-Frutos J, Herrero-Perez D, *et al.* Large-scale stochastic topology optimization using adaptive mesh refinement and coarsening through a two-level parallelization scheme [J]. Computer Methods in Applied Mechanics and Engineering, 2019, 343 (1): 186-206.

[8] Peng Yunfeng, Liu Hai. Extending OpenMP for the Optimization of Parallel Component Applications [J]. IEEE Access, 202, 8 (1): 95435-95441.

[9] 黄诗琦. 基于 agent 技术的电力物资供应保障系统的结构设计 [J]. 电子世界, 2019, 24 (1): 208-209. (Huang Shiqi. Structure design of power material supply and guarantee system based on agent Technology [J]. Electronics World, 2019, 24 (1): 208-209.)

[10] 霍启正. 基于M-agent技术的新能源接入下多区域联合调度研究 [J]. 通信电源技术, 2019, 36 (5): 1-4. (Huo Qizheng. Research on multi region joint scheduling under new energy access based on M-agent Technology [J]. Telecom Power Technology, 2019, 36 (5): 1-4.)

[11] 霍启敬. 基于 agent 技术的多区域联合调度协调消纳风电研究 [D]. 北京: 华北电力大学, 2019. (Huo Qijing. Based on agent technology multi-area joint dispatching coordination and consumption wind power research [D]. Beijing: North China Electric Power University, 2019.)

[12] Carvalho Silva J, Oliveira Dantas A B, Heron de Carvalho Junior F. A Scientific workflow management system for orchestration of parallel components in a cloud of large-scale parallel processing services [J]. Science of Computer Programming, 2019, 173 (1): 95-127.

[13] Mu Lifeng, Kwong C. K. A multi-objective optimization model of component selection in enterprise information system integration [J]. Computers & Industrial Engineering, 2018, 115 (1): 278-289.

[14] 彭云峰. 核磁共振图像处理并行构件框架设计与实现 [J]. 电子世界, 2018, 4 (1): 142-143. (Peng Yunfeng. Design and implementation of parallel component framework for NMR image processing [J]. Electronics World, 2018, 4 (1): 142-143.)

[15] 2020 CASA Team. Concerto: Parallel Adaptive Components-CASA Team [EB/OL]. (2020) [2020-07-27]. https://www-casa. irisa. fr/concerto/.

[16] Imperial College London. Imperial College e-Science Networked Infrastructure (ICENI) [EB/OL]. (2020) [2020-07-27]. https://www. imperial. ac. uk/london-e-science/projects/archive/.