第2卷 第3期 2020年6月
Vol.2 No.3 Jun.2020

ISSN 2096-742X
CN 10-1649/TP

文献DOI：
10.11871/jfdc.issn.
2096-742X.2020.
03.009

文献PID：
21.86101.2/jfdc.
2096-742X.2020.
03.009

页码：101-112

开放科学标识码
（OSID）

基于Charm++的并行FMM实现

丁磊[1,2]，王武[1*]，姜金荣[1]，赵莲[1]

1. 中国科学院计算机网络信息中心，北京 100190
2. 中国科学院大学，北京 100049

摘　要：【**目的**】为了利用Charm++的过分解与运行时迁移特性，提高FMM的并行执行效率，本文在Charm++上完成了FMM的并行实现。【**方法**】通过分析通信、并行任务分解、异步调用转化，采用SDAG实现了基本通信函数，并利用LPT近似策略达到了负载均衡，最终实现了并行FMM。【**结果**】测试结果表明，FMM的Charm++实现的计算精度与MPI实现完全相同，在千核规模上的执行速度优于MPI实现。过分解与负载均衡策略在粒子分布不均的情况下减少了10%的运行时间。【**局限**】目前的实现没有利用Charm++共享内存的结构，仍有优化的空间，负载均衡策略较为简单。【**结论**】本文给出了一个较为通用的MPI风格程序向Charm++转化的策略，并证明了Charm++的过分解与负载均衡策略对FMM有加速效果。

关键词： Charm++；FMM；负载均衡；过分解

Implementation of Parallel FMM Based on Charm++

Ding Lei[1,2], Wang Wu[1*], Jiang Jinrong[1], Zhao Lian[1]

1. Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China
2. University of Chinese Academy of Sciences, Beijing 100049, China

Abstract: [Objective] This paper has implemented a parallel FMM based on Charm++ to take advantage of its over-decomposition and migratability. **[Methods]** It is achieved by analyzing communication, separating parallel tasks, and converting synchronous communication to asynchronous communication. Also, the SDAG was used to implement the basic communication calls and the LPT approximation strategy was adopted for dynamic load balancing. **[Results]** The results show that the implementation of parallel FMM based on Charm++ has the same accuracy as that of MPI implementation, and its execution speed on the thousand-core scale is better than that of MPI implementation. Over-decomposition and load-balancing strategy contribute to the execution time reduction by 10% in the unbalance particle distribution. **[Limitations]** The current implementation does not use the shared memory structure of Charm++ and needs further optimizations. Besides, the load balancing strategy is simple. **[Conclusions]** This paper gives a relatively general method to convert the MPI style programs to Charm++ style ones and proves that over-decomposition and load-balancing strategy can accelerate FMM execution.

Keywords: Charm++；FMM；load balancing；over-decomposition

基金项目：国家重点研发计划“地球系统模式的改进、应用开发和高性能计算”（2016YFB0200800）；中国科学院科研信息化应用工程：高性能应用软件（XXH13506-405）；中国科学院战略性先导科技专项(C)：国产安全可控先进计算系统研制（XDC01040100）

*通讯作者：王武（E-mail：wangwu@sccas.cn）

引言

N 体问题 [1]（N-body problem）是已知多个粒子的初始位置、速度和质量，求解出其在经典力学下的后续运动的问题，属于高性能数值计算的七类典型应用之一 [2]，在宇宙学模拟 [3]、分子动力学研究 [4] 等诸多领域有十分重要的应用。

常用的 N 体求解方法有直接法 [5]（Particle-Particle, PP）、粒子 - 网格法 [5]（Particle-Mesh, PM）、树方法 [6]（Barnes-Hut, BH）、快速多极子方法 [7]（Fast Multipole Method, FMM）等。FMM 采用长程力近似计算方式，基于树结构的多层递归盒子划分策略，使用核心函数的级数展开，将远程力的作用近似为粒子与盒子的作用及盒子间的作用，极大降低了场中粒子相互作用的计算复杂度。由于可以指定划分的层数和展开阶数，FMM 能有效平衡执行时间与计算误差 [8]。与 BH 相比，FMM 基于高阶展开，因此精度可以达到更高，同时可以控制计算量。

近年来，国内外有许多 FMM 相关研究与应用成果。如曹小林 [9] 等基于 JASMIN 框架实现了 FMM 的求解器；Cruz[10] 等基于 PETSc 框架实现了 PetFMM 软件；Winkel[11] 等基于 MPI 实现了 N 体计算软件 PEPC。随着异构系统的不断发展，学者开始针对特定体系结构进行研究。Lashuk[12] 等使用 256 块 GPU 在 NCSA Lincoln cluster 上对 KIFMM 进行加速；Yokota[13] 等在 TSUBAME 2.0 系统上使用 4096 块 GPU 来计算各向同性湍流；王武 [14] 等基于申威众核处理器进行 FMM 优化。此外，还有基于 FMM-PM 方法在 GPU 上进行 N 体模拟实现 [15] 等。

Charm++ 是一种基于消息驱动的编程框架，具有过分解（Over-decomposition）、可迁移（Migratability）、异步消息驱动（Asynchronous Message-Driven Execution）等特点，可以提供运行时的动态负载均衡与容错机制 [16]。近年来，有许多工作利用 Charm++ 的动态负载均衡特性，取得了不错的效果。如中国科学院计算机网络信息中心研发的并行框架 SC_Tangram[17]、基于 Charm++ 的非结构网格 [18] 等。本文中，我们将基于 Charm++ 实现 FMM 程序的并行化，并与 MPI 版本的并行实现进行对比分析，结果表明整体性能有 10% 左右的提升。

1 并行 FMM 介绍

1.1 FMM 的计算流程

FMM 是一种区域划分的计算方法，其核心思路在于对位势函数在远场进行多极展开，再将多极展开转化为近场的局部展开，继而通过位势计算出粒子在引力场所受到的作用力。

FMM 使用球坐标的多极展开来近似多个粒子形成的引力场对外界的作用，用局部展开来近似外界引力场对区域内粒子的作用，包括 P2M、M2M、M2L、L2L、L2P、P2P 六个计算步骤。

下列公式中，α、β、ρ 分别表示球坐标的极角、方位角与径向距离，m_i 为粒子质量。P 为展开阶数，球谐展开中的系数 m、n 与 P 正相关。

（1）P2M (Particle to Multipole Expansion)

P2M 操作只发生在计算区域的最底层，也就是最小的盒子区域内。主要的思想是根据盒子里面的粒子位置，计算出这个盒子的多极展开系数。

P2M 的计算方法如下 [19]：

$$M_n^m = \sum_{i=1}^{k} m_i \cdot \rho_i^n \cdot Y_n^{-m}(\alpha_i, \beta_i)$$

其中，k 为盒子中的粒子数，M_n^m 是多极展开系数，$Y_n^m(\theta,\varphi)$ 是球谐函数，计算公式为：

$$Y_n^m(\theta,\phi) = \sqrt{\frac{(n-|m|)!}{(n+|m|)!}} P_n^{|m|}(\cos\theta) e^{im\phi}$$

式中 $P_n(x)$ 为 Legendre 多项式。

（2）M2M (Multipole Expansion to Multipole Expansion)

M2M 操作的计算方法如下 [19]：

$$M_j^k = \sum_{n=0}^{j} \sum_{m=-n}^{n} \frac{O_{j-n}^{k-m} \cdot i^{|k|-|m|-|k-m|} \cdot A_n^m \cdot A_{j-n}^{k-m} \cdot \rho^n \cdot Y_n^{-m}(\alpha,\beta)}{A_j^k}$$

其中，M_j^k 是转移中心之后的多极展开系数，O_j^k 是下一层盒子的多极展开系数。

M2M 是一个自下向上的过程，不断地收集下层的多极展开系数，计算本层的多极展开系数，再到更高一层进行计算。

（3）M2L (Multipole Expansion to Local Expansion)

P2M 操作与 M2M 只进行多极展开系数的计算，目的是计算出盒子对空间的作用效果。而 M2L 操作是计算相互作用。

M2L 是一个多极展开转化为局部展开的过程，计算方法如下 [19]：

$$L_j^k = \sum_{n=0}^{\infty}\sum_{m=-n}^{n} \frac{O_n^m \cdot i^{|k-m|-|k|-|m|} \cdot A_n^m \cdot A_j^k \cdot Y_{j+n}^{m-k}(\alpha,\beta)}{(-1)^n A_{j+n}^{m-k} \cdot \rho^{j+n+1}}$$

其中，L_j^k 为局部展开系数，O_n^m 为多极展开系数。

在计算 M2L 的过程中，需要找到当前盒子的次相邻盒子，即与当前盒子不相邻，但是父盒子与当前盒子的父盒子相邻的盒子。根据这些盒子的多极展开系数，计算出局部展开系数。

（4）L2L (Local Expansion to Local Expansion)

L2L 的计算公式如下 [19]：

$$L_j^k = \sum_{n=j}^{P}\sum_{m=-n}^{n} \frac{\mathrm{O}_n^m \cdot i^{|m|-|m-k|-|k|} \cdot A_{n-j}^{m-k} \cdot A_j^k \cdot Y_{n-j}^{m-k}(\alpha,\beta) \cdot \rho^{n-j}}{(-1)^{n+j} \cdot A_n^m}$$

其中，L_j^k 为局部展开系数，O_n^m 为局部展开系数。

L2L 是一个与 M2M 相反的过程，完成了相互作用的中心转移。在 M2L 中只考虑了本层的相互作用，那些来自父盒子的作用力是通过 L2L 来计算。这样，就可以计算出来自非邻居盒子的作用。

（5）L2P (Local Expansion to Particle)

L2P 是 P2M 的逆过程。经过 L2L 操作后，已经在最底层盒子中获取来自非邻居盒子的作用，而 L2P 就是把这些局部展开系数转为盒子中粒子所受到的作用力。

L2P 的计算公式如下 [19]：

$$\Phi(P) = \sum_{j=0}^{P}\sum_{k=-j}^{j} L_j^k \cdot Y_j^k(\theta,\phi) \cdot r^j$$

（6）P2P (Particle to Particle)

由于近似计算需要两个区域有一定的距离，无法使用近似的方式求解出最底层盒子来自邻居盒子的作用力。因此，只能依照万有引力公式，求解出邻居盒子中粒子的作用力。

1.2 并行 FMM 算法

FMM 的并行分解即为空间分解。如算法 1 所示，每个并行单元负责一个区域，在区域中按照预定分布初始化全局粒子的位置，通过 partition 操作将粒子划分给处理区域，并依照希尔伯特编码建立树。此后执行粒子到盒子以及盒子到盒子的 P2M 和 M2M 操作，获得多极展开系数。

算法 1 并行 FMM

Algorithm: Parallel FMM

Input: int numBodies, string distribution
Output:
numBodies: number of bodies
distribution: body distribution

```
// initial particles by distribution
bodies = initBodies (numBodies, distribution)
// send particles to regions they should be in
partition (bodies)
// build Octrees in local region
cells = buildTree (bodies)
P2M & M2M (bodies)
// gain message from neighbor to build local essential tree
localEssentialTree (jbodies, jcells)
M2L & P2P (cells, jcells)
L2L & L2P (cells)
```

鉴于 P2P 操作和 M2L 操作需要了解相邻和次相邻盒子的信息，需要进行一次全局通信，将这些信息保存在本地关建树（local essential tree）中。最终执行树的自上而下的遍历，完成 L2L 与 L2P 操作。因此，在并行 FMM 中需要进行两次全局通信，分别发生在 partition 和 local essential tree 处。

2 Charm++ 介绍

Charm++ 是基于 C++ 的面向对象并行编程框架，由 UIUC 并行编程实验室开发，具有过分解、可迁移、异步消息驱动的特点 [20]。过分解指并行计算对象的数量多于实际的处理器数量。可迁移指并行计算对象可以在处理器间移动，完成动态负载均衡。异步消息驱动指每个处理器上的并行对象的执行流由消息驱动，计算对象间的消息传递采用异步方式，不阻塞执行流。

Chare 是 Charm++ 的基本计算单元，其本质是分布在不同的处理器上的 C++ 对象。代理（Proxy）是一种特殊的 C++ 对象，用以表示一个远程的 Chare。Chare 间的通信通过调用代理的特殊成员方法来实现，由 Runtime 负责将本次调用的参数封装成消息，并发送到对应的远程 Chare，该过程通常被称为远程调用。

Charm++ 提供了 Chare array、group、node group 等并行数据结构。其中，Chare array 提供一个统一的代理，每个 array 的成员 Chare 都有一个 thisIndex，可以通过 proxy[someIndex] 来调用编号为 someIndex 的 Chare 上的方法。

PUP (Pack and Unpack) 框架是 Charm++ 提供的序列化框架，负责远程调用和 Chare 迁移时的序列化工作。

PE (Processing Element) 和 Node 是 Charm++ 的抽象概念。其中 PE 是 Chare 的执行实体，其上有许多 Chare 和消息。Node 由多个 PE 组成，是一种共享内存的抽象表达。Node 上的不同 Chare 共享 Node 的地址空间。在实现上，PE 一般对应线程，而 Node 对应于进程。

Charm++ 的各种实体概念如图 1 所示。

Charm++ 的项目由 ci (Charm++ interface) 文件和 C++ 源码构成。其中 ci 文件定义了 Charm++ 中的 Chare、message、module 等实体。预处理系统解析 ci 文件后生成对应头文件和实现文件，使用者需要在 C++ 源码中引入这些文件。

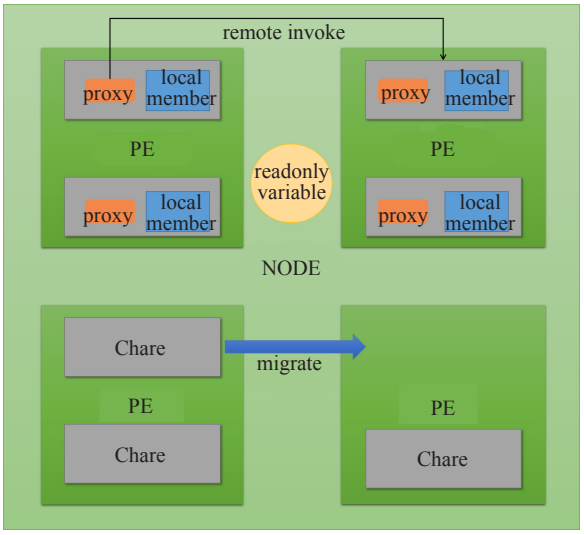

图 1 Charm++ 概念示意图

Fig.1 Charm++ concept diagram

Charm++ 不支持全局可变变量，但支持全局只读变量。只读变量在 ci 文件中使用 readonly 关键字引入，在 mainChare 的构造函数中进行初始化。

每个 PE 上都有一个消息队列。以 PE 的视角看，有一个执行流不断从队列中取出消息。如果消息是种子，将创建新的 Chare。如果消息是远程调用，则找到 PE 上对应的 Chare，将消息解包并调用。在调用过程中可能会产生新的种子和远程调用，由 Runtime 决定新消息对应的 PE。

因此，Chare 间的消息传递是异步的方式。调用者只知道消息在队列里，但不清楚消息是否已被执行，即调用者无法立即获得执行的结果。使用者可以定义一个远程方法供接收方调用，并在这个远程方法中获取通信结果，执行后续操作。这种风格使代码杂乱且易错。为了方便表示执行流的依赖关系，Charm++ 提供了 SDAG[21](structure dagger) 表示方法。

SDAG 的语法采用 <keyword> { } 模式，其中 <keyword> 有 when、serial 等。serial 用于表示一段完整执行的不被抢占的代码。when 表示等待消息，如 when method (arg) {serial {do_something (arg) }} 表示当前执行流等待远程方法 method 被调用，并在该条件触发后执行 do_something 操作。

Charm++ 预处理器会分析 ci 文件中的 SDAG 语法，并生成对应的代码，将 SDAG 语句块拆分成许多 continuation 函数，即多个阶段。以 when 语句为例，Runtime 会在 method 被触发后，调用下一个阶段对应的 continuation。这样使用者无需自行编写等待、判断以及跳转的远程方法。

3 并行实现

3.1 任务分解与预处理

在本文中，我们定义一个名为 master 的 mainChare 作为并行任务的发起者和规约消息的接收者，定义一个名为 slave 的一维 Chare array，作为基本的并行计算单元。

鉴于 Chare 需要在 PE 上执行，而且存在运行时迁移的情况，我们将原始的全局变量保存在 slave 的成员中，并以参数的方式传递进去。

对于计算开始前已知且只读的全局变量，使用 Charm++ 提供的 readonly 变量表示，并在 master 中进行初始化。

3.2 异步通信转化

Charm++ 采用异步通信的方式，这使得通信的发起方无法了解到消息是否被接收。

在 MPI 中，我们可以在一个函数中发起多次通信。对于阻塞通信，操作系统会将函数的实参和局部变量保存在栈中，等到通信结束后再进行调度；对于非阻塞通信，程序中有循环代码对通信结束条件进行判断，控制流始终在该函数中。在 Charm++ 中，用户只能通过消息驱动来触发后续操作，这意味着通信发起时，我们就失去了局部变量和实参的内存分配。为了恢复状态，需要将跨越通信的实参和局部变量保存到 Chare 的成员中。

针对通信函数 XXmethod，我们引入结构体 XXmethod_param 来保存这些信息。考虑到实参的初始化有传值和传引用的方式，我们只在结构体里保存变量地址。对于传值的方式，需要引入 init_XX_by_value 来动态分配内存并保存；对于传引用的方式，引入 init_XX_by_ref，来保存已有变量的值，并提供一个 release 方法用于释放值初始化时分配的空间。

另一方面，某个通信函数可能被调用多次，结束后跳转到不同的执行流。为了保证通信结束后函数的正常驱动，我们需要保存通信结束后的下一步执行的位置。

本文中，我们将需要通信的函数设置为 slave 的成员函数，并根据通信将程序划分为多个 state。在 slave 的成员变量中，使用一个栈来保存通信结束后下一步执行的函数指针以及 state 值。通过 set_continuation 来将函数指针和 state 压栈，do_continuation 来将函数指针和 state 出栈，并调用该函数。在每个 state 开始时，我们取出函数中需要用到的局部变量和实参。在第一个 state 的时候对局部变量进行初始化。实参的初始化发生在函数的调用处。

```
void SomeMethod (SomeType parameter){
    SomeType local_variable;
    do_some_serial_job_1 (parameter, local_variable);
    global_communication (local_variable);
    do_some_serial_job_2 (parameter, local_variable);
}
```

图 2 MPI 风格示例通信函数

Fig.2 Example of communication in MPI style

图 2 是一个 MPI 风格的通信函数示例，在 SomeMethod 函数中，有全局通信 global_communication。其中有实参 parameter，有局部变量 local_variable，这些变量都要跨越通信的。

图 3 是对应的 Charm++ 表达方式。全局通信将 SomeMethod 分成了两个部分。state 是 Chare array 的成员变量，用于标识当前处理的状态。CharmSomeMethod_param 是我们针对通信函数 CharmSomeMethod 引入的结构体，其中保存了该函数执行时需要的实参 parameter、局部变量 local_variable。

在 state 为 0 的时候，我们调用 set_continuation 函数来指定在本次通信后需要执行的函数为 CharmSomeMethod 的下一个 state，并调用 init_XX_by_value 来为局部变量分配内存。实参 parameter 的初始化发生在 CharmSomeMethod 被调用处。此后，我们通过 CharmSomeMethod_param 获得 parameter 和 local_variable，并执行串行操作 do_some_serial_job_1。鉴于通信函数 global communication 需要传入 local_variable 的引用，这里我们对 global_communication_param 使用 init_XX_by_ref 进行初始化，记录 local_variable 的地址。在执行完 global_communication 通信后，执行流到了 CharmSomeMethod 的 state1，再一次从 CharmSomeMethod_param 中获得 parameter 和 local_variable 变量，执行串行操作 do_some_serial_job_2。由于该函数已经执行到最后，可以释放 CharmSomeMethod_param 中的空间，并调用 do_continuation，获取之前保存的下一步执行的函数和 state。global_communication 内部实现与 CharmSomeMethod 类似。

```
void CharmSomeMethod (){
  if (state == 0){
    set_continuation (&mpifmm::slave::CharmSomeMethod, 1);
    CharmSomeMethod_param.init_local_variable_by_value ();
    SomeType& parameter = CharmSomeMethod_param. \
          get_parameter ();
    SomeType& local_variable = CharmSomeMethod_param. \
          get_local_variable ();
    do_some_serial_job_1 (parameter, local_variable);
    global_communication_param.init_by_ref (local_variable)
    global_communication ();
}
else if (state == 1) {
    SomeType& parameter = CharmSomeMethod_param.\
          get_parameter ();
    SomeType& local_variable = CharmSomeMethod_param.\
          get_local_variable ();
    do_some_serial_job_2 (parameter, local_variable);
    CharmSomeMethod_param.release ();
    do_continuation ();
  }
}
```

图 3 Charm++ 风格示例通信函数

Fig.3 Example of communication in Charm++ style

本文只对最基本的通信函数使用 SDAG。这样对于调用层数较深处发生的通信，只需要提供基本通信的实现，无需将所有包含通信的函数都改写成 SDAG 风格，可以较大程度减少代码修改量。至此，除基本通信函数外，所有的 MPI 风格的同步通信逻辑都可以转化为 Charm++ 异步通信的表示，且用户修改部分较少。

基本通信函数的实现将在下一节中介绍。

3.3 基本通信的实现

并行 FMM 的通信发生在 partition 和 local essential tree 操作中，包括基本通信 Alltoallv 与 Allreduce。

Alltoallv 需要每个并行计算单元向其它并行计算单元发送不等长的消息，同时从其它并行计算单元中接收这些不等长的消息。Allreduce 需要对所有并行单元的某个参数进行规约。

```
message MSG_<Typename>
{
    <Typename> data[];
    int size;          // the length of data
    int from;      // the sender ID of this message
}
```

图 4 message 类型的定义

Fig.4 Defination of message

在 Charm++ 中，使用 message 比参数序列化快 [22]，而且用户无需过多了解序列化知识。我们针对不同的类型，定义了对应的 message 类型，如图 4 所示。其中，data 保存信息的本体，size 表示消息的大小，from 表示消息的发送者。

伪代码 1 Alltoallv 的 Charm++ 实现

Pseudocode: Alltoallv
Input: Alltoallparam param, CProxySlave thisProxy **Output:** Param: information about Alltoallv thisProxy: proxy to the Chare array

```
serial {
    sendbuf, sendcount, sdispl = get_data_from (param)
    for i=0 to Chare_size do
        message = init_message (sendbuf, sendcount,sdispl)
        thisProxy[i].send_message (message)
    end for
        recv_cnt = Chare_size
    }
    while (recv_cnt -- ){
        when send_message (msg) serial{
          recvbuf, recvcount, rdispl = get_data_from (param)
          set_variable_from (recvbuf, recvcount, rdispl, msg)
        }
    }
    serial {
        do_continuation ()
    }
```

在实现 Alltoallv 时，我们使用结构体来保存整个通信过程需要用到的变量。在 serial 语句块中，每个 Chare 构造消息并发送，同时触发等待消息的 when 语句块，从消息中取出数据，放到缓冲区中。Alltoallv 的实现如伪代码 1 所示，其中 param 保存了 alltoalv 调用时的参数，thisProxy 指 Chare array 的代理，通过这个代理向自己发送消息。

伪代码 2 Allreduce 在 slave 上的实现

Pseudocode: Allreduce for slave

```
Input: Allreduceparam param
Output:
param: information about Allreduce

serial {
    sendbuf, count= get_data_from (param)
    callback_object = create_callback (sendbuf, count)
    contribute_to_master (masterProxy, callback_object)
}
when recv_message (msg) serial{
    recvbuf = get_data_from (param)
    set_variable_from (recvbuf, msg)
}
serial {
    do_continuation ()
}
```

Allreduce 的实现包括规约和广播。首先在 slave 上获取到所有的规约数据，指定 master 的处理函数来做规约。在规约结束后，master 发布广播消息，触发 slave 的 when 语句，master 和 slave 的实现过程见伪代码 2 与伪代码 3。

伪代码 3 Allreduce 在 master 上的实现

Pseudocode: Allreduce for master

```
Input: CkReductionMsg * msg, CProxySlave slaveProxy
Output:
Param: information about Allreduce
size = get_size_of_message (msg)
databuf = extract_from_message (msg)
slaveProxy.recv_message (databuf)
```

3.4 负载均衡与迁移策略

大规模分布式并行系统上动态负载均衡，可以分成分布式、集中式和层次式三种 [23]。

分布式策略只考虑相邻处理器，如 Cybenko[24] 等给出了扩散算法的一般化形式，处理器从高负载处理器中获取负载，并把负载给低负载处理器。Hui[25] 等提出了基于水动力学的负载均衡，利用水的连通性特性来控制负载均衡。

集中式策略考虑全局负载，将信息集中到一个处理器上进行计算。如全局随机指派 [26]，贪心方法 [26] 等。

层次式负载均衡策略根据拓扑结构建立层次树。如 HBM[27] 根据网络结构将处理器划分成多个组，在层次结构的每一层都执行负载均衡迁移，直到根节点为止。

Charm++ 提供了在运行时移动 Chare 的方法，使得处理器上的负载可以进行动态调节。本文中，我们利用 MigrateMe 函数，使用集中式负载均衡策略，在运行时对 slave 的 Chare 进行迁移。

图 5 是负载均衡的实现时序图。slave 在某个阶段将自己包含的粒子信息发送到 master 上。master 负责对这些信息进行分析，经由负载均衡策略给出

一个 Chare 的重新分布，并依次通知相应的 Chare。这些 Chare 调用 MigrateMe 函数来移动到新的 PE 上，并向 master 汇报迁移的结束。master 收到所有的迁移结束消息后，向 slave 发起一个广播，使其继续执行计算。

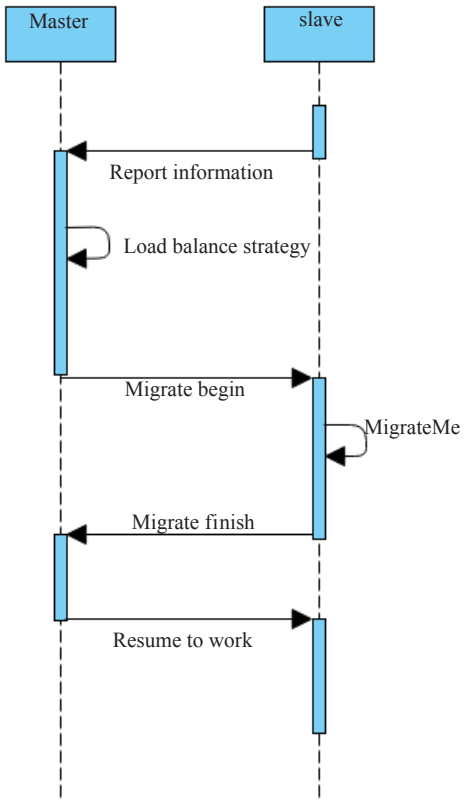

图 5 负载均衡时序图

Fig.5 Sequence Diagram of load balance

本文中，负载均衡的执行时期为 partition 结束时，此时所有的 Chare 都已经拿到自己负责处理的粒子。为了简化负载均衡的实现逻辑，我们假设所有的处理器都是相同性能，仅考虑 PE 的计算负载，并用 PE 上的总粒子数来近似。

该负载均衡问题被称为相同并行机调度（Identical Parallel-Machine Scheduling）问题，属于 NP-hard 问题 [28]。本文中，我们采用最长处理时优先策略（Longest Processing Time，LPT）来求解，简述如下：

(1) 将 Chare 按照粒子数量从大到小排序

(2) 依次将 Chare 指派到当前粒子总数最少的 PE 上。

该策略可以达到 4/3 近似 [28]，即对于该问题的所有情况，近似策略都能给出最优解 4/3 倍以内的可行解。

3.5 实现技术

上述 message、Alltoallv、Allreduce 等需要针对不同类型实现。此外，对于每个通信函数，都需实现一个 XX_param 的结构体，其成员函数需要覆盖到在通信中使用到的变量，且应当区分 by_value 和 by_ref。在通信函数的最开始部分，都需要获得全部的 XX_param 结构体成员。这些代码都是相似的，可以使用脚本语言生成。

本文将需要生成的类型参数写成 json 的格式，通过 Python 脚本解析，基于模板自动生成了 message、基础通信函数、XX_param 结构体的 C++ 代码片段和宏定义。在并行 FMM 源码中只需在对应的地方引入这些宏定义即可，无需大量修改源码。

主要的实现流程包括：

(1) 全局变量局部化；

(2) 基于通信划分 state；

(3) Python 脚本生成 C++ 代码片段；

(4) 调整通信函数内部风格；

(5) 在通信函数中引入 XX_param 等代码。

4 数值结果

以下测试均在中国科学院计算机网络信息中心超级计算机“元”上进行测试。每台刀片计算节点配置 2 颗 Intel E5-2680 V2（Ivy Bridge | 10C | 2.8GHz）处理器，64 GB DDR3 ECC 1866MHz 内存。编译环境为 Intel Composer XE 2013 编译器，Charm++ 版本为 6.6.1，编译时使用 -O3 优化选项和 C++0x 标准。

4.1 正确性检验

为了验证 FMM 的 Charm++ 实现正确性，我们选择了不同分布与参数进行测试，比较了 Charm++ 与 MPI 下每个粒子的受力值，结果完全一致，这表明我们的计算精度和已有的 MPI 实现的精度相同。

4.2 性能检验

以 16384 万个粒子为例，采用均匀的立方体分布作为粒子的初值分布，分别对 MPI 和 Charm++ 实现进行测试，保证 Chare 数量和 PE 数量相等（即不使用过分解），结果如图 6 所示。可以看出，随着处理器规模从 32 核增加到 1024 核，两种实现的速度基本接近，Charm++ 实现相比 MPI 实现在千核规模上有提升。

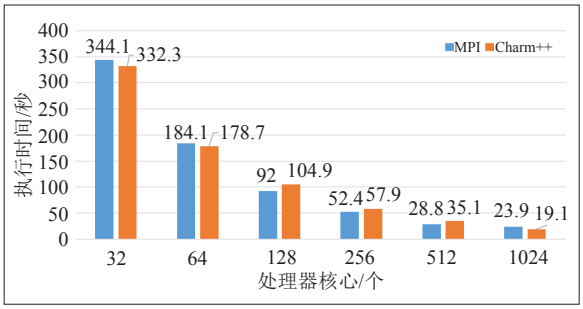

图 6 16384 万粒子的执行时间

Fig.6 Execution time of 163,840,000 particles

在上述测试中，以 32 核规模的执行时间作为基准，计算了 Charm++ 和 MPI 版本的加速比，结果如图 7 所示。可以看出，与理想情况相比，FMM 的 Charm++ 实现在 256 核以下的强扩展性能较好。随着核数增加到千核规模，全局通信会扩大，扩展性有一些下降。与 MPI 实现相比，Charm++ 实现在 1024 核规模的加速效果较好，但在 64 核至 512 核规模的加速效果较为逊色。这一现象是因为千核规模下，通信占比增大，而 MPI 实现使用全局通信，在 partition 阶段的通信时间远大于使用点对点通信的 Charm++ 实现。

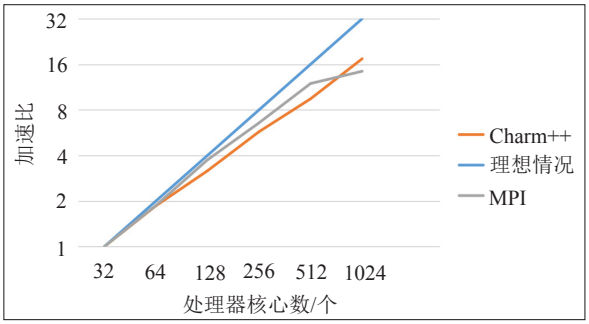

图 7 并行 FMM 的加速比

Fig.7 Speedup ratio of different implementations

为了验证过分解对负载不均衡情况的优化效果，本文针对 65536 万粒子的球壳状不均匀分布，固定处理器数量为 64，调整每个处理器上的 Chare 数量，从 64 增加到 512，进行了测试。我们统计了每个处理器上的总粒子数 N_i，并计算了所有处理器上的极差（$N_{max}-N_{min}$）和标准差（$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(N_i-\overline{N})^2}$）。

表 1 过分解测试结果

Table 1 Result of over-decomposition

Chare 总数量	时间/秒	标准差	极差	均值
64	640.53	527032	2755781	10240000
128	591.73	85038	432258	10240000
256	576.40	112005	563038	10240000
512	585.27	164129	593829	10240000

测试结果如表 1 所示，在不使用过分解的情况下，可以看出粒子分布的标准差和极差较大。随着 Chare 数量增加，LPT 负载均衡策略降低了标准差和极差的数量级，也减少了计算时间。最优情况在使用 256 个 Chare 时取得，与不采用过分解的情况相比，减少了 10% 的计算时间。上述结果表明，过分解和负载均衡策略对粒子分布不均匀下的 FMM 计算有一定优化效果，可以平衡不同处理器上的计算负载。

5 总结与展望

本文基于 Charm++ 并行编程框架，实现了并行

FMM 程序。首先分析了 FMM 的并行实现关键和主要通信，介绍了 Charm++ 并行编程环境。在并行实现的过程中，本文讨论了 MPI 编程模型和 Charm++ 编程模型的异同，引入了用异步模型表示同步计算的一般性方法，给出了一个通用的 MPI 并行程序向 Charm++ 转化的流程，并针对基本通信函数给出了 SDAG 的实现方案。最终，本文在“元”超级计算机上对 FMM 的 Charm++ 实现进行了检验与测试。结果表明，基于 Charm++ 实现的 FMM 具有较好的扩展性，针对负载不均衡的情况，过分解和 LPT 负载均衡策略可以减少 10% 的执行时间。

文本中实现中只利用了 Charm++ 提供的 Chare array 类型，即把每个 Chare 看作一个独立且内存不共享的个体。如何使用 Charm++ 提供的逻辑节点与本地存储来减少通信量，提高并行效率，将是未来的一个尝试方向。另一方面，本文在负载均衡时仅考虑了 PE 上的总粒子数，且假定计算负载与粒子数成正相关，不一定反映了真实的负载情况。因此，使用更贴近 FMM 的负载均衡指标也是一个可以考虑的方向。

针对目前高性能计算系统广泛使用异构计算的情况 [29]，本文提供的 Charm++ 实现也具有一定适应性。只需声明一组绑定 PE 的 Charm group，在其中执行耗时的 P2P 与 M2L 操作。异构加速与具体体系结构有关，涉及内容较多，故本文仅提供接口。

利益冲突声明

所有作者声明不存在利益冲突关系

参考文献

[1] 郑哲. N-Body算法分析及N-Body问题关键算粒在FPGA上的实验验证研究[D]. 上海交通大学,2011.

[2] Colella P. Defining software requirements for scientific computing [EB/OL]. [2020-04-06]. http://www.lanl.gov/conferences/salishan/salishan2005/supinski.pdf.

[3] 冯珑珑,朱维善. 现代宇宙学中的数值模拟技术和应用[J]. 中国科学:物理学、力学、天文学,2013,(06):687-707.

[4] Papa M, Giuliani G, Bonasera A. Constrained molecular dynamics II: An N-body approach to nuclear systems[J]. Journal of Computational Physics, 2005, 208(2): 403-415.

[5] 王岳青. 多体模拟的并行优化及软件架构关键技术研究[D]. 国防科学技术大学,2012.

[6] Barnes J, Hut P. A hierarchical O (N log N) force-calculation algorithm[J]. nature, 1986, 324(6096): 446-449.

[7] Greengard L, Rokhlin V. A fast algorithm for particle simulations[J]. Journal of Computational Physics, 1997, 135(2): 280-292.

[8] 王武,冯仰德,迟学斌. 树结构在N体问题中的应用[J]. 计算机应用研究,2008,(01):42-44.

[9] 曹小林. 基于 JASMIN 框架的粒子模拟并行计算[J]. 科研信息化技术与应用, 2010, 1(2): 28-33.

[10] Cruz F A, Knepley M G, Barba L A. PetFMM—A dynamically load‐balancing parallel fast multipole library[J]. International journal for numerical methods in engineering, 2011, 85(4): 403-428.

[11] Winkel M, Speck R, Hübner H, et al. A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations[J]. Computer physics communications, 2012, 183(4): 880-889.

[12] Lashuk I, Chandramowlishwaran A, Langston H, et al. A massively parallel adaptive fast-multipole method on heterogeneous architectures[C]//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. IEEE, 2009: 1-12.

[13] Yokota R, Barba L A, Narumi T, et al. Petascale turbulence simulation using a highly parallel fast

multipole method on GPUs[J]. Computer Physics Communications, 2013, 184(3): 445-455.

[14] 王武,王舒扬,姜金荣,等. 快速多极子方法在申威众核处理器上的实现和优化[J]. 计算机工程与科学, 2019 (7): 3.

[15] 扶月月,王武,王乔. 基于FMM-PM方法的宇宙N体模拟在GPU上的实现和优化[J]. 数据与计算发展前沿, 2020, 2(2): 155-164.

[16] Kale L V, Krishnan S. CHARM++:a portable concurrent object oriented system based on C++[J]. Acm Sigplan Notices, 1995, 28(10):91-108.

[17] 迟学斌,赵莲,王姗姗,等. 高性能计算框架软件——SC_Tangram[J]. 数据与计算发展前沿, 2019, 1(1): 11-21.

[18] Wang T, Chi X, Zhao L, et al. Parallel Unstructured Grid Partition Algorithm Based on Charm++[C]//2019 IEEE International Conference on Computational Electromagnetics (ICCEM). IEEE, 2019: 1-3.

[19] 徐文哲. 基于谱域球谐展开的多层快速多极子算法研究[D]. 电子科技大学,2007.

[20] Acun B, Gupta A, Jain N, et al. Parallel programming with migratable objects:charm++ in practice[C]// High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for. 2014:647-658.

[21] Gursoy A, Kale L V. Dagger: Combining benefits of synchronous and asynchronous communication styles[C]//Proceedings of 8th International Parallel Processing Symposium. IEEE, 1994: 590-596.

[22] Charm++ Documentation - Parallel Programming Laboratory [EB/OL]. [2020-03-01]. http://charm.cs.illinois.edu/manuals/pdf/charm++.pdf

[23] 杨际祥,谭国真,王荣生. 并行与分布式计算动态负载均衡策略综述[J]. 电子学报, 2010, 38(5): 1122-1130.

[24] Cybenko G. Dynamic load balancing for distributed memory multiprocessors[J]. Journal of parallel and distributed computing, 1989, 7(2): 279-301.

[25] Hui C C, Chanson S T. Hydrodynamic load balancing[J]. IEEE Transactions on Parallel and Distributed Systems, 1999, 10(11): 1118-1137.

[26] Zheng G. Achieving high performance on extremely large parallel machines: performance prediction and load balancing[R]. 2005.

[27] Willebeek-LeMair M H, Reeves A P. Strategies for dynamic load balancing on highly parallel computers[J]. IEEE Transactions on parallel and distributed systems, 1993, 4(9): 979-993.

[28] Xiao X. A Direct Proof of the 4/3 Bound of LPT Scheduling Rule[C]//2017 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology (FMSMT 2017). Atlantis Press, 2017.

[29] 张云泉,袁良,袁国兴,李希代. 2019年中国高性能计算机发展现状分析与展望[J].数据与计算发展前沿,2020,2(1):18-26.

收稿日期：2020 年 2 月 19 日

丁磊，中国科学院计算机网络信息中心，硕士研究生，主要研究方向为高性能计算、并行计算。

本文承担工作为：移植与负载均衡方案设计，代码实现，结果测试。

Ding Lei is a master student at Computer Network Information Center of Chinese Academy of Sciences. His main research interests are high performance computing and parallel computing.

In this paper he undertakes the following tasks: design, implementation and test of the load balance strategy.

E-mail：dinglei2017@cnic.cn

王武，中国科学院计算机网络信息中心，博士，副研究员，研究方向为并行算法，高性能计算。

本文承担工作为：FMM 与 N-body 开发指导。

Wang Wu, Ph.D., is an associate research fellow at Computer Network Information Center, Chinese Academy of Sciences. His main research interests are parallel algorithm and high performance computing.

In this paper, he is the execution director of implementation of FMM and N-body simulation.

E-mail: wangwu@sccas.cn

姜金荣，中国科学院计算机网络信息中心，博士，研究员，主要研究方向为并行算法与框架软件、计算地球科学。

本文承担工作为：整体方案设计指导。

Jiang Jinrong is the research fellow at Computer Network Information Center of Chinese Academy of Sciences. His main research interests are parallel computing algorithms and frameworks.

In this paper he undertakes the following tasks: design of general implementation.

E-mail: jjr@sccas.cn

赵莲，中国科学院计算机网络信息中心，助理研究员，博士，主要研究方向为高性能计算、并行计算。

本文承担工作为：Charm++ 开发指导。

Zhao Lian, Ph.D., is an assistant research fellow at Computer Network Information Center of Chinese Academy of Sciences. Her main research interests are high performance computing and parallel computing.

In this paper she directed the code implementation of Charm++.

E-mail: zhaolian@sccas.cn

引文格式：丁磊,王武,姜金荣,赵莲. 基于Charm++的并行FMM实现[J].数据与计算发展前沿,2020,2(3): 101-112.DOI:10.11871/jfdc.issn.2096-742X.2020.03.009.PID:21.86101.2/jfdc.2096-742X.2020.03.009.

Ding Lei,Wang Wu, Jiang Jinrong, Zhao Lian. Implementation of Parallel FMM Based on Charm++ [J].Frontiers of Data & Computing, 2020,2(3): 101-112. DOI:10.11871/jfdc.issn.2096-742X.2020.03.009.PID:21.86101.2/jfdc.2096-742X.2020.03.009.