针对大规模点集三维重建问题的分布式捆绑调整方法

刘 鑫 [1] 孙凤梅 [2] 胡占义 [1]

摘 要 捆绑调整 (Bundle adjustment, BA) 是三维重建中的关键步骤, 它需要消耗大量的计算时间和内存存储空间. 本文旨在处理三维点数比相机模型数多很多的捆绑调整问题, 我们称之为针对大规模三维点集的捆绑调整 (Massive-points bundle adjustment, MPBA) 问题. 此类问题在对高分辨率图像进行三维重建时会经常出现. 为了高效地解决 MPBA 问题, 本文提出一种分布式的捆绑调整算法. 通过基于三维点集划分的分解方法, 原 MPBA 问题被分成若干子问题. 该分解方法不依赖于输入参数的内在联系, 因而分解结果与具体 BA 问题无关. 算法被映射于两个集群上, 一个集群有 5 台计算机, 另一个集群有 3 台计算机, 其中每台机器都配置一块图形处理器 (Graphic processing unit, GPU). 通过对若干 MPBA 问题的实验, 与经典捆绑调整算法 SBA (Sparse bundle adjustment) 相比, 本文算法获得了最高达 75 倍的加速比, 并保持了算法的高精确度. 而且, 本文算法的两个实现所消耗的单机内存存储空间, 仅为 SBA 实现的 1/7 和 1/4.

关键词 捆绑调整, 计算机集群, 图形处理器, 运动相机重建三维结构

引用格式 刘鑫, 孙凤梅, 胡占义. 针对大规模点集三维重建问题的分布式捆绑调整方法. 自动化学报, 2012, **38**(9): 1428−1438

DOI 10.3724/SP.J.1004.2012.01428

Distributed Bundle Adjustment in 3D Scene Reconstruction with Massive Points

LIU Xin[1] SUN Feng-Mei[2] HU Zhan-Yi[1]

Abstract Bundle adjustment (BA) is a crucial step in 3D scene reconstruction but time and memory consuming. In this paper, we try to tackle a frequently encountered BA problem where the reconstructed 3D points are more numerous than the camera parameters, namely massive-points BA problem. This is often the case when high-resolution images are used. We present a novel distributed bundle adjustment (DBA) algorithm for efficiently solving the massive-points BA problem, where the original BA problem is divided into sub-problems by partitioning the 3D reconstructed points. Such a partition scheme is in dependent of the input parameters, it could be applied to various BA problems. Two specific implementations, one on a shared memory cluster of 5 computers and the other on a cluster of 3 GPU (Graphic processing unit)-integrated computers, are developed. These implementations are experimentally compared with the classical single-thread sparse bundle adjustment (SBA). Experimental results show that our algorithm is up to 75 times faster than SBA, while maintaining comparable precision. And the one-computer memory requirements of these two implementations are just 1/7 and 1/4 of the original SBA.

Key words Bundle adjustment (BA), cluster, graphic processing unit (GPU), structure from motion (SFM)

Citation Liu Xin, Sun Feng-Mei, Hu Zhan-Yi. Distributed bundle adjustment in 3D scene reconstruction with massive points. *Acta Automatica Sinica*, 2012, **38**(9): 1428−1438

近年来, 基于图像的大场景三维重建一般采用由运动相机重建三维结构 (Structure from motion, SFM) 的方法. 为了提高该方法的效率, Agarwal 等将 SFM 过程映射到了计算机集群上[1]. 计算机集群 (简称集群) 是一种计算机系统, 它通过连接一组松散集成的计算机软件和/或硬件, 来紧密地协作完成计算工作. 集群在 SFM 中的成功应用, 引起了计算机视觉研究团体的广泛关注. 值得注意的是, 文献 [1] 报道的 SFM 系统并没有采用分布式的捆绑调整算法.

捆绑调整 (Bundle adjustment, BA) 是 SFM 的核心问题, 它通过最小化一个代价函数, 对若干相机模型参数和若干三维点参数同时进行优化调整. 随着这些参数数量的增加, BA 需要消耗大量的运算时间, 因此成为了 SFM 的性能瓶颈. 当前的大规模 BA 问题可以根据其输入参数分为两类. 一类问题是海量的三维点和大量的相机模型 (几千个

收稿日期 2011-08-31 录用日期 2012-02-09
Manuscript received August 31, 2011; accepted February 9, 2012
国家自然科学基金 (60973005) 和中国科学院战略性先导科技专项 (XDA06030300) 资助
Supported by National Natural Science Foundation of China (60973005) and Strategic Priority Research Program of the Chinese Academy of Sciences (XDA06030300)
本文责任编委 周杰
Recommended by Associate Editor ZHOU Jie
1. 中国科学院自动化研究所模式识别国家重点实验室 北京 100190 2. 北方工业大学理学院 北京 100144
1. National Laboratory of Pattern Recognition, Institute of Automation, Chinese Academy of Sciences, Beijing 100190 2. Faculty of Sciences, North China University of Technology, Beijing 100144

或上万个相机模型). 从图片网站 (例如 flick.com) 采集大量的图片数据, 并将 SFM 应用于这些图片, 便能生成此类问题[1−3]. 另一类大规模 BA 问题是海量的三维点和中等数量的相机模型. 在此类问题中, 输入三维点集的规模与第一类问题相同, 但相机模型数往往只有几百个. 这里称之为针对大规模三维点集的捆绑调整 (Massive-points bundle adjustment, MPBA) 问题[4]. 此类问题常见于在 SFM 中使用高分辨率图像或稠密三维重建中[4−5]. 随着相机技术的不断进步, 高分辨率图像的运用越来越普遍. 运用少量的高分辨率图像, 便能重建出稠密的三维场景, 从而提高重建的效率. 所以, MPBA 是一个在具体应用中亟待解决的问题.

对于将捆绑调整应用于基于图像的三维重建, 有代表性的工作有: 使用最广泛的 BA 算法是经典捆绑整算法 (Sparse bundle adjustmeny, SBA)[6], 它利用增量标准方程的稀疏结构来提高解方程的效率. 但 SBA 仅适用于参数数量较少的 BA 问题, 对于大规模 BA 问题, 该算法非常耗时. 因此, 为减小问题的规模, Ni 等[7] 将原 BA 问题分解成若干子问题 (Submaps), 并将 BA 算法迭代地应用于每个子问题和全局问题. 问题分解提高了 BA 算法的效率. 但是该分解方法依赖于输入参数的内在联系, 因而其分解结果与具体 BA 问题相关. Fang 等[5] 为减少 BA 算法的时间开销及内存存储开销, 对输入的三维点集进行了重采样 (Resampling). 但是, 由于大量有价值的信息随三维点数量的减少而丢失, 因而降低了重建三维点云的精确度. 另一方面, 为了解决大规模 BA 问题, 研究者提出了许多高效的算法. Agarwal 等[8] 生成了若干 BA 问题来测试 6 种 BA 算法的性能, 并得出 “BA 算法的效率与输入问题的结构相关” 的结论. 他们的实验表明, 对于相机数量仅数百个的问题, SBA 是高效的解决方案, 而对于有更多相机的问题, 基于预处理共轭梯度法 (Preconditioned conjugate gradients, PCG) 的 BA 算法[8−11] 则更加高效. 另一个趋势是利用并行化的硬件来加速捆绑调整. Choudhary 等[12] 设计了一个混合了 CPU 计算和 GPU 计算的 BA 算法. 在他们的算法中, Hessian 矩阵和 Schur 完形 (Schur complements) 都在 GPU 端生成. 对于有大量三维点参数的 MPBA 问题, 即使使用最新式的 GPU 工作站显存, 也不够存储这些中间结果. Wu 等[13] 改进了文献 [8] 中的两个基于 PCG 的 BA 算法, 并将算法分别在多核 CPU 上和单个 GPU 上实现. 但他们的算法主要针对第一类大规模 BA 问题, 并不适用于 MPBA 问题[4, 13]. 而且, 这些算法将所有参数和中间结果都存储在显存上, 存在不易扩展的缺点. Hu 等[4] 为解决 MPBA 问题设计了一种 BA 算法, 能够充分利用多核 CPU 和多个 GPU 的并行化硬件资源. 但由于此种算法将消耗大量的内存, 也不适合大规模的 MPBA 问题. 本文的主要贡献有: 1) 针对 MPBA 问题的特点, 提出一种基于三维点集划分的 BA 问题分解方法. 由于该方法不依赖于输入参数的内在联系, 因而其分解结果与具体 BA 问题无关, 通用性强. 2) 提出一种分布式的捆绑调整算法. 据我们所知, 对于基于图像的三维重建, 这是第一个分布式的捆绑调整算法. 通过对若干 MPBA 问题的实验, 相比经典算法 SBA, 本文算法获得了 25 倍 ∼ 75 倍的加速比, 并保持了算法的高精确度. 单机内存消耗也降为 SBA 的 1/7 ∼ 1/4, 使算法适用于更大规模的 MPBA 问题.

1　理论背景

1.1　捆绑调整及 SBA 算法简述

算法 1. SBA 算法

1) 初始化;

2) 计算 Jacobian 矩阵

$$A_{ij}=\frac{\partial Q(\boldsymbol{a}_j,\boldsymbol{b}_i)}{\partial \boldsymbol{a}_j},\quad B_{ij}=\frac{\partial Q(\boldsymbol{a}_j,\boldsymbol{b}_i)}{\partial \boldsymbol{b}_i}$$

3) 计算 $J^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}}{\sum}}J=\begin{bmatrix} U & W \\ W^{\mathrm{T}} & V \end{bmatrix}$, 其中

$$U_j=\sum_{i=1}^{n}A_{ij}^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}_{ij}}{\sum}}A_{ij}$$

$$V_i=\sum_{j=1}^{m}B_{ij}^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}_{ij}}{\sum}}B_{ij},\quad W_{ij}=A_{ij}^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}_{ij}}{\sum}}B_{ij}$$

计算 $J^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}}{\sum}}\boldsymbol{\varepsilon}=\begin{bmatrix} \boldsymbol{\varepsilon_a} \\ \boldsymbol{\varepsilon_b} \end{bmatrix}$

$$\boldsymbol{\varepsilon}_{\boldsymbol{a}_j}=\sum_{i=1}^{n}A_{ij}^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}_{ij}}{\sum}}\boldsymbol{\varepsilon}_{ij}$$

$$\boldsymbol{\varepsilon}_{\boldsymbol{b}_i}=\sum_{j=1}^{m}B_{ij}^{\mathrm{T}}\overset{-1}{\underset{\boldsymbol{x}_{ij}}{\sum}}\boldsymbol{\varepsilon}_{ij}$$

4) 在矩阵 U_j 和 V_i 的对角线上增加 μ 得到 U_j^* 和 V_i^*;

5) 计算 S 和 $\boldsymbol{e_a}$

计算 Y: $Y_{ij}=W_{ij}^{*-1}$;

计算 S: $S_{jk}=U_j^*-\sum_{i=1}^{n}Y_{ij}W_{ik}^{\mathrm{T}}$;

计算 $\boldsymbol{e_a}$: $\boldsymbol{e_a}=\boldsymbol{\varepsilon_a}-WV^{*-1}\Sigma_{\boldsymbol{b}}$;

6) 解线性方程 $S\boldsymbol{\delta_a}=\boldsymbol{e_a}$, 解方程失败则转步骤 4);

7) 回代 (由 $\boldsymbol{\delta_a}$ 构造 $\boldsymbol{\delta}$);

8) 假如达到了算法终止条件则算法结束, 否则

转步骤 2).

这里首先对捆绑调整和 SBA 算法作简单介绍, 更详细的介绍可参考文献 [6, 9]. 输入 m 个相机模型和 n 个三维点, 捆绑调整旨在最小化重投影误差, 如式 (1) 所示.

$$\min_{\boldsymbol{a}_j,\boldsymbol{b}_i}\sum_{j=1}^{m}\sum_{i=1}^{n}d\left(Q\left(\boldsymbol{a}_j,\boldsymbol{b}_i\right),\boldsymbol{x}_{ij}\right) \tag{1}$$

$Q\left(\boldsymbol{a}_j,\boldsymbol{b}_i\right)$ 表示第 i 个三维点到第 j 个相机的投影函数, 其中 $\boldsymbol{a}_j$ $(j=1,2,\cdots,m)$ 和 $\boldsymbol{b}_i$ $(i=1,2,\cdots,n)$ 分别是相机模型参数和三维点坐标参数的向量表示. 若 $\boldsymbol{x}$ 和 $\boldsymbol{y}$ 分别是测量投影点坐标和实际投影点坐标的向量表示, 则 $d(\boldsymbol{x},\boldsymbol{y})$ 表示它们的欧氏距离. 对于输入参数, 定义向量 $\boldsymbol{P}\in\mathbf{R}^M$ $(M=pnp\times p+cnp\times m)$ 表示所有三维点向量和相机模型向量的组合:

$$\boldsymbol{P}=\left(\boldsymbol{a}_1^{\mathrm{T}},\boldsymbol{a}_2^{\mathrm{T}},\cdots,\boldsymbol{a}_m^{\mathrm{T}},\boldsymbol{b}_1^{\mathrm{T}},\boldsymbol{b}_2^{\mathrm{T}},\cdots,\boldsymbol{a}_n^{\mathrm{T}}\right)^{\mathrm{T}}$$

定义向量 $\boldsymbol{X}\in\mathbf{R}^N$ $(N=nvis\times mnp)$ 表示所有实际投影点向量 $\boldsymbol{x}_{ij}$ $(i=1,2,\cdots,n;j=1,2,\cdots,m)$ 的组合:

$$\boldsymbol{X}=\left(\boldsymbol{x}_{11}^{\mathrm{T}},\cdots,\boldsymbol{x}_{1m}^{\mathrm{T}},\boldsymbol{x}_{21}^{\mathrm{T}},\cdots,\boldsymbol{x}_{2m}^{\mathrm{T}},\cdots,\boldsymbol{x}_{n1}^{\mathrm{T}},\cdots,\boldsymbol{x}_{nm}^{\mathrm{T}}\right)^{\mathrm{T}}$$

对于测量投影点, 定义向量表示所有测量投影点向量的组合:

$$\hat{\boldsymbol{X}}=\left(\hat{\boldsymbol{x}}_{11}^{\mathrm{T}},\cdots,\hat{\boldsymbol{x}}_{1m}^{\mathrm{T}},\hat{\boldsymbol{x}}_{21}^{\mathrm{T}},\cdots,\hat{\boldsymbol{x}}_{2m}^{\mathrm{T}},\cdots,\hat{\boldsymbol{x}}_{n1}^{\mathrm{T}},\cdots,\hat{\boldsymbol{x}}_{nm}^{\mathrm{T}}\right)^{\mathrm{T}}$$

其中, $\hat{\boldsymbol{x}}_{ij}=Q\left(\boldsymbol{a}_j,\boldsymbol{b}_i\right)$. 则 $\hat{\boldsymbol{x}}$ 可表示为 $\hat{\boldsymbol{X}}=f\left(\boldsymbol{P}\right)$, 其中, f 表示投影函数.

SBA 算法通过反复地解如下的线性方程:

$$\left(J^{\mathrm{T}}\sum_{\boldsymbol{X}}^{-1}J+\mu I\right)\boldsymbol{\delta}=J^{\mathrm{T}}\sum_{\boldsymbol{X}}^{-1}\boldsymbol{\varepsilon} \tag{2}$$

来得到式 (1) 的解. 其中, $\boldsymbol{\delta}$ 是向量 $\boldsymbol{P}$ 的改变量, 其结果在每次迭代后得到; $J=\frac{\partial\boldsymbol{X}}{\partial\boldsymbol{P}}$ 是 f 的 Jacobian 矩阵; μ 是阻尼因数; $\varepsilon=\boldsymbol{X}-\hat{\boldsymbol{X}}$. 式 (2) 被称为增量标准方程 (Augmented normal equation)[9]. 为便于记述, 给出一些相关的定义:

m: 相机模型数;

n: 三维点数;

mnp: 每个投影点的参数个数, 一般为 2;

$nvis$: 所有投影点个数的总和;

cnp: 每个相机模型的参数个数. 一般为 9, 包括姿态信息 (6) 和相机内参数 (3);

pnp: 每个三维点的参数个数, 欧拉点一般为 3.

1.2 PCG 简述

共轭梯度法 (Conjugate gradients, CG) 是解线性方程 $A\boldsymbol{x}=\boldsymbol{b}$ 的一种迭代方法. 其中, A 表示一个对称正定矩阵, $\boldsymbol{b}$ 是向量, $\boldsymbol{x}$ 表示待求未知数. 对于 BA 问题, CG 是一种一阶方法 (First order method)[9], 其基本形式仅需计算矩阵 A 与某一向量的乘积和其他向量运算, 而不需要进行矩阵乘法运算和矩阵分解运算. 相比二阶方法 (Second order method), CG 每次迭代运算时间很短, 但算法收敛需要更多次的迭代[9]. 在某些应用中, 为保证 CG 的快速收敛, 需进行预处理 (Precondition). 算法 2 是预处理共轭梯度法 (Precondition conjugate gradients, PCG) 的简要描述. 其中, $\boldsymbol{x}$, $\boldsymbol{r}$, $\boldsymbol{z}$, $\boldsymbol{p}$ 均表示向量; α 和 β 是标量; M 表示预处理矩阵; l 表示当前迭代次数. 在捆绑调整中应用 PCG 有两种方式: 一是设 $A=J^{\mathrm{T}}\Sigma_{\boldsymbol{X}}^{-1}J+\mu I$ 和 $b=J^{\mathrm{T}}\Sigma_{\boldsymbol{X}}^{-1}\boldsymbol{\varepsilon}$ (式 (2)); 另一种是设 $A=S$ 和 $\boldsymbol{b}=\boldsymbol{e_a}$ (SBA 算法步骤). 详情可参考文献 [8 − 11].

算法 2. PCG 算法

1) 初始化:

$\boldsymbol{x}_0=0$, $\boldsymbol{r}_0=\boldsymbol{b}-A\boldsymbol{x}_0$

$\boldsymbol{z}_0=M^{-1}\boldsymbol{r}_0$, $\boldsymbol{p}_0=\boldsymbol{z}_0$, $k=0$;

2) 计算 $\alpha_l=\frac{\boldsymbol{r}_l^{\mathrm{T}}\boldsymbol{z}_l}{\boldsymbol{p}_l^{\mathrm{T}}A\boldsymbol{p}_l}$;

3) 计算 $\boldsymbol{x}_{l+1}=\boldsymbol{x}_l+\alpha_l\boldsymbol{p}_l$,

$\boldsymbol{r}_{l+1}=\boldsymbol{r}_l-\alpha_lA\boldsymbol{p}_l$, $\boldsymbol{z}_{l+1}=M^{-1}\boldsymbol{r}_l+1$

$\beta_l=\frac{\boldsymbol{r}_{l+1}^{\mathrm{T}}\boldsymbol{z}_{l+1}}{\boldsymbol{r}_l^{T}\boldsymbol{z}_l}$, $\boldsymbol{p}_{l+1}=\boldsymbol{z}_{l+1}+\beta_l\boldsymbol{p}_l$;

4) 假如没有达到终止条件, 则转步骤 2);

5) 算法结束. $\boldsymbol{x}_{l+1}$ 是最后结果.

2 分布式 BA 算法

在本节中, 将详细描述本文提出的分布式 BA 算法 (Distributed bundle adjustment, DBA): 首先, 给出算法的设计原则; 然后, 给出 BA 问题的分解方法和 DBA 的初始化步骤; 最后, 对 DBA 的其他 6 个主要步骤分别介绍. 算法假设集群有 r 个节点: 节点 1 作为控制节点, 负责问题分解和任务分配; 其他节点分别表示为节点 2, 节点 3, $\cdots$, 节点 r. DBA 的算法流程见图 1. 其中, 方框表示 DBA 算法的各个步骤: (a) 初始化; (b) 计算 Jacobian 矩阵、计算 $J^{\mathrm{T}}\Sigma_{\boldsymbol{X}}^{-1}J$ 和计算 S&$\boldsymbol{e_a}$; (c) 解线性方程; (d) 计算 $\boldsymbol{\delta}$ 和计算终止条件; (e) 终止条件判断. 虚线框表示运行各个步骤的节点. N_i 表示第 i 个节点; M 表示控制节点; A 表示所有节点. 箭头表示各个节点的数据依赖关系. 在箭头描述项上, 第一个下标表示数据流经过的节点; 第二个下标表示当前算法的迭代次数.

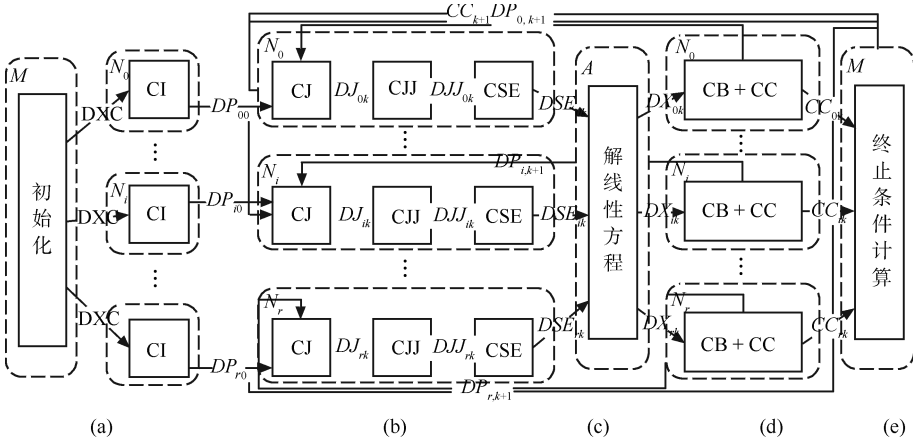


图 1　DBA 算法

Fig. 1　DBA algorithm

2.1　算法设计原则

以下是 DBA 算法设计需考虑的两个主要因素:

1) MPBA 问题的特点[4];

2) 增加集群节点的吞吐量和减少集群节点的通信开销.

对于基于图像的三维重建, 当前最流行的 BA 算法主要有两类: 一类是基于 Schur 完形和简化捆绑方程系统 (Reduced bundle system) 的二阶方法[4, 6, 8−9]; 另一类则是基于共轭梯度法的一阶方法[8−11]. MPBA 问题一般有上百万个三维点但仅有数百个相机模型. 当相机模型数量较少时, 文献 [8] 给出了三种高效算法: SBA、Explicit-Jacobi 和 Implicit-ssor. 其中, SBA 和 Explicit-Jacobi 均属于第一类算法, 但它们分别采用稠密矩阵分解和 PCG 来求解线性方程 $S\boldsymbol{\delta_a} = \boldsymbol{e_a}$. 而 Implicit-ssor 属于第二类算法, 它直接使用 PCG 来求解式 (2). 相比前两种算法, Implicit-ssor 每次迭代运算时间更短, 但算法收敛需要更多次的迭代. 将三种算法应用于若干 MPBA 问题, 实验表明, 若将前两种算法映射到集群上, 每次迭代的运算时间要远远高于节点间通信的时间, Implicit-ssor 却正好相反. 因此, 分布式的第一类算法将更加适合于 MPBA 问题.

此外, 如何对原 BA 问题进行分解, 是算法设计面临的另一个挑战. 考虑到以下三个原因, 本文采用基于三维点集划分的问题分解方法: 1) 该方法与问题无关且容易实现. 2) 对于 MPBA 问题, 三维点数比相机模型数多 3 ∼ 4 个数量级. 基于三维点集划分的分解方法能够更加均匀地分配计算任务, 增加分布式硬件的吞吐量. 3) 分解得到的子问题能够直接映射到各个计算节点上. 各节点的运算独立, 节点间仅需进行少量通信.

最后, 由于 S 是一个 $N \times N$ $(N = cnp \times m)$ 的矩阵, 收集各个节点的计算结果来构造 S 将造成不可接受的通信开销. 我们设计了一种无需直接计算矩阵 S 的分布式算法来解线性方程 $S\boldsymbol{\delta_a} = \boldsymbol{e_a}$.

2.2　问题分解和初始化

定义三维点集合 $D = \boldsymbol{b}_0, \boldsymbol{b}_1, \cdots, \boldsymbol{b}_n$; 实际投影点集合: $M = \boldsymbol{x}_{11}, \cdots, \boldsymbol{x}_{1m}, \boldsymbol{x}_{21}, \cdots, \boldsymbol{x}_{2m}, \boldsymbol{x}_{n1}, \cdots, \boldsymbol{x}_{nm}$; 相机参数集合 $D = \boldsymbol{a}_0, \boldsymbol{a}_1, \cdots, \boldsymbol{a}_m$. 将三维点集合 D 划分 (Partition of a set)[14] 成 r 个非空无交集子集合, 表示为 $D = D^1 \oplus D^2 \oplus \cdots \oplus D^r$, 其中, $D^k = \{b_{k_s} | s = 1, 2, \cdots, n_k; k_s \in (1, n)\}$. 集合 D_k 包含 n_k 个三维点, 且有 $n = \sum_{k=1}^{r} n_k$. 为了便于表示, 将 n_k 记为 t. 同样地, 集合 M 也分成相应的 r 个子集合, 记为 X_k $(k = 1, 2, \cdots, r)$. D_k 和 X_k 的关系是: 对于每个三维点 $b_{k_s} \in D_k$ $(s = 1, 2, \cdots, n_k)$, 其对应的投影点集合 $E^{k_s} = \{x_{k_s,l} | l = 1, 2, \cdots, m\}$ 满足 $E^{k_s} \in X^k$. 则有 $M = X^1 \oplus X^2 \oplus \cdots \oplus X^r$. 在算法实现中 (第 3 节), 对集合 D 做了随机的平均划分: $n_{\text{ave}} = n_2 = n_3 = \cdots = n_r$; $n_1 = n_{\text{ave}} + n\%r$, 其中 "%" 表示除后的余数. 实验表明, 这种划分方法高效稳定. 对于相机参数集合 C, 仅仅将其广播到了所有计算节点上.

DBA 的初始化步骤分成两个子步. 第一个子步是在控制节点进行集合划分, 并将划分得到的子集拷贝到各个计算节点. 第二个子步是在所有接收到子集的节点进行初始化. 初始化步骤类似于 SBA (SBA 算法步骤 1).

2.3　计算 Jacobian 矩阵

定义向量 $\boldsymbol{P}_k = \left(\boldsymbol{a}_1^{\mathrm{T}}, \cdots, \boldsymbol{a}_m^{\mathrm{T}}, \boldsymbol{b}_{k_1}^{\mathrm{T}}, \cdots, \boldsymbol{b}_{k_t}^{\mathrm{T}}\right)^{\mathrm{T}}, k = 1, 2, \cdots, r$. 则矩阵 J 可表示为

$$J = \begin{bmatrix} J_0 \\ J_1 \\ \vdots \\ J_r \end{bmatrix}$$

其中, $J_k = \frac{\partial \boldsymbol{X}}{\partial \boldsymbol{P}_k}$. 定义矩阵集合 $A^k = \{A_{k_s,j}|s=1,2,\cdots,n_k; j=1,2,\cdots,m\}$ 和 $B^k = \{B_{k_s,j}|s=1,2,\cdots,n_k; j=1,2,\cdots,m\}$. 计算稀疏矩阵 J_k, 仅需计算 A_k 和 B_k. 于是, 算法在节点 k 上计算矩阵集合 A_k 和 B_k $(k=1,2,\cdots,r)$. 由于子集 D_k 和集合 C 已拷贝到第 k 个节点, 因而计算仅需在本地节点进行.

2.4 计算 $\boldsymbol{J}^{\mathrm{T}}\sum_{\boldsymbol{x}}^{-1}\boldsymbol{J}$ 和 $\boldsymbol{J}^{\mathrm{T}}\sum_{\boldsymbol{x}}^{-1}\boldsymbol{\varepsilon}$

$J^{\mathrm{T}}\Sigma_{\boldsymbol{x}}^{-1}J$ 可表示为

$$\begin{bmatrix} U & W \\ W^{\mathrm{T}} & V \end{bmatrix}$$

其中, $U = \mathrm{diag}\{U_1,U_2,\cdots,U_m\}$; $V = \mathrm{diag}\{V_1,V_2,\cdots,V_n\}$; $W=(W_{ij})_{m\times n}$. 另外

$$J^{\mathrm{T}}\sum_{\boldsymbol{X}}^{-1}\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_a \\ \varepsilon_b \end{pmatrix}$$

其中

$$\boldsymbol{\varepsilon_a} = \begin{pmatrix} \boldsymbol{\varepsilon_a}0 \\ \boldsymbol{\varepsilon_a}1 \\ \vdots \\ \boldsymbol{\varepsilon_a}m \end{pmatrix}; \quad \boldsymbol{\varepsilon_b} = \begin{pmatrix} \boldsymbol{\varepsilon_b}0 \\ \boldsymbol{\varepsilon_b}1 \\ \vdots \\ \boldsymbol{\varepsilon_b}n \end{pmatrix}$$

计算 U 和 e_a: 矩阵 U_j $(j=1,2,\cdots,m)$ 可表示为

$$U_j = \sum_{i=1}^{n} A_{ij}^{\mathrm{T}} \sum_{\boldsymbol{x}_{ij}}^{-1} A_{ij} = \sum_{k=1}^{r}\sum_{i=k_1}^{k_t} A_{ij}^{\mathrm{T}} \sum_{\boldsymbol{x}_{ij}}^{-1} A_{ij} \quad (3)$$

记矩阵

$$U_j^k = \sum_{i=k_0}^{k_t} A_{ij}^{\mathrm{T}} \sum_{\boldsymbol{x}_{ij}}^{-1} A_{ij}, \quad k=1,2,\cdots,r \quad (4)$$

则式 (3) 可重写为

$$U_j = \sum_{k=1}^{r} U_j^k \quad (5)$$

定义矩阵集合 $U^k = \left\{U_1^k, U_2^k, \cdots, U_m^k\right\}$ $(k=1,2,\cdots,r)$. 算法在节点 k 上计算矩阵集合 U^k $(k=1,2,\cdots,r)$. 由式 (4) 可知, 计算 U^k 仅需集合 A^k 中的矩阵. 由于 A^k 已在上一步得到, 因而计算仅需在本地节点进行. 此外, 在得到矩阵集合 U^k $(k=1,2,\cdots,r)$ 后, 算法不再需要计算原矩阵 U, 因而无需进行节点间通信. 同理, 算法在节点 k 上得到集合

$$\boldsymbol{\varepsilon}_{\boldsymbol{a}}^k = \left\{ \boldsymbol{\varepsilon}_{\boldsymbol{a}_j}^k | j=1,2,\cdots,m; \boldsymbol{\varepsilon}_{\boldsymbol{a}_j}^k = \sum_{i=k_1}^{k_t} A_{ij}^{\mathrm{T}} \sum_{\boldsymbol{x}_{ij}}^{-1} \boldsymbol{\varepsilon}_{ij} \right\}$$

也不再需要计算向量

$$\boldsymbol{\varepsilon}_{\boldsymbol{a}}j = \sum_{k=1}^{r} \boldsymbol{\varepsilon}_{\boldsymbol{a}_j}^k, \quad j=1,2,\cdots,m$$

计算 V, W 和 $\boldsymbol{e}_b$: 定义矩阵集合 V^A 为

$$V^A = \{V_1,V_2,\cdots,V_n\} = \{V_{k_s}|k=1,2,\cdots,r; s=1,2,\cdots,n_k\}$$

为了得到矩阵 V, 仅需计算出矩阵集合 V^A. 矩阵 V_{k_s} $(k=1,2,\cdots,r; s=1,2,\cdots,n_k)$ 可由下式计算.

$$V_{k_s} = \sum_{j} B_{k_sj}^{\mathrm{T}} \sum_{x_{k_sj}}^{-1} B_{k_sj} \quad (6)$$

其中

$$B_{k_sj} \in B^k, \quad j=1,2,\cdots,m$$

定义矩阵集合 V^k $(k=1,2,\cdots,r)$:

$$V^k = \{V_{k_0}, V_{k_1}, \cdots, V_{k_t}\}$$

则

$$V^A = V^1 \oplus V^2 \oplus \cdots \oplus V^r$$

于是, 算法在节点 k 上可计算矩阵集合 V^k $(k=1,2,\cdots,r)$. 由式 (6) 可知, 计算 V^k 也仅需在本地节点进行, 无需进行节点间的通信.

同理, 为了得到矩阵 W 和向量 $\boldsymbol{\varepsilon_b}$, 仅需计算全部矩阵 W_{ij} $(i=1,2,\cdots,n; j=1,2,\cdots,m)$ 的集合 W^A 和全部向量 $\boldsymbol{\varepsilon_b}i$ $(i=1,2,\cdots,n)$ 的集合 ε_b^A. 定义矩阵集合 W^k $(k=1,2,\cdots,r)$:

$$W^k = \left\{ W_{k_s,j} | s=1,2,\cdots,n_k; \; j=1,2,\cdots,m; W_{k_sj} = A_{k_sj}^{\mathrm{T}} \sum_{x_{k_sj}}^{-1} B_{k_sj} \right\}$$

则 W^A 可表示为

$$W^A = \{W_{11}, W_{12}, \cdots, W_{1m}, \cdots, W_{n1}, W_{n2}, \cdots, W_{nm}\} = W^1 \oplus W^2 \oplus \cdots \oplus W^r$$

定义向量集合 ε_b^k $(k=1,2,\cdots,r)$:

$$\varepsilon_b^k = \left\{ \boldsymbol{\varepsilon}_{\boldsymbol{b}_{k_s}}^k | s=1,2,\cdots,n_k; \; \boldsymbol{\varepsilon}_{\boldsymbol{b}_{k_s}} = \sum_{j} B_{k_sj}^{\mathrm{T}} \sum_{\boldsymbol{x}_{k_sj}}^{-1} \boldsymbol{\varepsilon}_{k_sj} \right\}$$

则 ε_b^A 可表示为

$$\varepsilon_b^A = \{\varepsilon_{b_1}, \varepsilon_{b_2}, \cdots, \varepsilon_{b_n}\} = \varepsilon_b^1 \oplus \varepsilon_b^2 \oplus \cdots \oplus \varepsilon_b^r$$

算法在节点 k 上计算 W^k 和 ε_b^k $(k=1,2,\cdots,r)$.

2.5　增加 $\boldsymbol{U}$ 和 $\boldsymbol{V}$

在节点 k 上, 计算矩阵 $U_j^{k*}\,(k=1,2,\cdots,r;\ j=1,2,\cdots,m)$:

$$U_j^{k*}=U_j^k+\left(\frac{\mu}{r}\right)I$$

则矩阵 U_j^* (算法 2 步骤 4)) 可表示为

$$U_j^*=\sum_{k=1}^{r}U_j^{k*}$$

对矩阵 $V_{k_s}\,(k=1,2,\cdots,r;s=1,2,\cdots,n_k)$ 作增量:

$$V_{k_s}^{k*}=V_{k_s}^k+\mu I$$

2.6　计算 $\boldsymbol{S}$ 和 $\boldsymbol{e_a}$

本步骤可分为两个子步: 第一子步计算 $Y=WV^{*-1}$; 第二子步计算 $S=U^*-YW^{\mathrm{T}}$ 和 $\boldsymbol{e_a}=\boldsymbol{\varepsilon_a}-Y\boldsymbol{\varepsilon_b}$.

2.6.1　计算 $\boldsymbol{Y}$

定义所有矩阵

$$Y_{ij}\,(i=1,2,\cdots,n;j=1,2,\cdots,m)$$

的集合为 Y^A. 为了得到矩阵 Y, 仅需计算出矩阵集合 Y^A. 定义矩阵集合 $Y^k\ (k=1,2,\cdots,r)$:

$Y^k=$

$$\{Y_{k_1,1},Y_{k_1,2},\cdots,Y_{k_1,m},Y_{k_2,1},\\ Y_{k_2,2},\cdots,Y_{k_2,m},\cdots,Y_{k_t,1},Y_{k_t,2},\cdots,Y_{k_t,m}\}$$

则 $Y^A=Y^1\oplus Y^2\oplus\cdots\oplus Y^r$. 而

$$Y_{k_s,j}\,(k=1,2,\cdots,r;s=1,2,\cdots,n_k;\\ j=1,2,\cdots,m)$$

可以通过下式计算:

$$Y_{k_s,j}=W_{k_s,j}V_{k_s}^{*-1}\tag{7}$$

于是, 算法在节点 k 上计算矩阵集合 $Y^k\ (k=1,2,\cdots,r)$. 由式 (7) 可知, 在节点 k 上, 计算 Y^k 所需矩阵均已在前面几步中得到.

2.6.2　计算 $\boldsymbol{S}$ 和 $\boldsymbol{e_a}$

矩阵 S 可记为 $(S_{jl})_{m\times m}$, 其中, S_{jl} 是一个 $cnp\times cnp$ 的对称矩阵. 矩阵 $S_{jl}\ (j=1,2,\cdots,m;l=1,2,\cdots,m)$ 可表示为

$$\begin{aligned}S_{jl}&=U_j^*-\sum_i Y_{ij}W_{il}^{\mathrm{T}}=\\ &\sum_{k=1}^{r}U_j^{k*}-\sum_{k=1}^{r}\sum_{i=k_1}^{k_t}Y_{ij}W_{il}^{\mathrm{T}}=\\ &\sum_k\left(U_j^{k*}-\sum_{i=k_1}^{k_t}Y_{ij}W_{il}^{\mathrm{T}}\right)\end{aligned}\tag{8}$$

记矩阵 $S_{jl}^k\ (k=1,2,\cdots,r)$ 为

$$S_{jl}^k=U_j^{k*}-\sum_{i=k_0}^{k_t}Y_{ij}W_{il}^{\mathrm{T}}\tag{9}$$

记矩阵 S^k 为 $\left(S_{jl}^k\right)_{m\times m}$, 则式 (8) 可重写为

$$S_{jl}=\sum_{k=1}^{r}S_{jl}^k\tag{10}$$

于是

$$S=\sum_{k=1}^{r}S^k\tag{11}$$

算法在节点 k 上计算矩阵 $S^k\ (k=1,2,\cdots,r)$. 由式 (8) ∼ 式 (11) 可知, 计算仅需在本地节点进行.

同理, 记:

$$\boldsymbol{e_a^k}=\begin{pmatrix}\boldsymbol{e_{a0}^k}\\ \boldsymbol{e_{a1}^k}\\ \vdots\\ \boldsymbol{e_{am}^k}\end{pmatrix},\quad k=1,2,\cdots,r$$

其中

$$\boldsymbol{e_{aj}^k}=\boldsymbol{\varepsilon_{aj}^k}-\sum_{i=k_0}^{k_t}Y_{ij}\boldsymbol{\varepsilon_{bi}^k},\quad j=1,2,\cdots,m$$

则向量 $\boldsymbol{e_a}$ 可表示为

$$\boldsymbol{e_a}=\sum_{k=1}^{r}\boldsymbol{e_a^k}\tag{12}$$

算法在节点 k 上计算向量 $\boldsymbol{e_a^k}\ (k=1,2,\cdots,r)$.

由式 (11), 在得到 S^k 和 $\boldsymbol{e_a^k}\ (k=1,2,\cdots,r)$ 后, 要计算矩阵 S, 需要所有计算节点对空间复杂度 $\mathrm{O}(m^2)$ 的数据作规约运算 (Reduction)[15−16]. 规约运算所需的时间开销是不可接受的. 相反, 由式 (12), 要计算向量 $\boldsymbol{e_a}$, 仅需对空间复杂度 $\mathrm{O}(m)$ 的数据进行规约, 规约所需的时间开销极小. 鉴于上述两方面的考虑, 我们为解线性方程 $S\boldsymbol{\delta_a}=\boldsymbol{e_a}$ 设计了分布式 PCG (Distributed PCG, DPCG) 算法, 下面将对此进行介绍.

2.7　解线性方程

针对不同的集群配置, 本文提出两种 DPCG 算法: 简单 DPCG 算法和分组 DPCG 算法. 这里仅考虑采用 Jacobi 预处理的方法. 当然, 其他预处理方法, 如文献 [6] 中所述, 也同样适用.

2.7.1 简单 DPCG 算法

简单 DPCG 算法 (图 2) 的各个步骤为: (a) 初始化; (b) 计算 $\boldsymbol{e_a}$; (c) 计算 $S^k\boldsymbol{p}_l$; (d) 计算 $S\boldsymbol{p}_l$ 和其他. 其中计算最密集的部分为: 计算 $\boldsymbol{e_a}$ 和计算 $S\boldsymbol{p}_l$.

1) 计算 $\boldsymbol{e_a}$.

利用式 (12), 所有计算节点进行规约得到向量 $\boldsymbol{e_a}$. 这一步仅需进行一次.

2) 计算 $S\boldsymbol{p}_l$.

$S\boldsymbol{p}_l$ 可由下式计算:

$$S\boldsymbol{p}_l = \left(\sum_{k=1}^{r} S^k\right)\boldsymbol{p}_l = \sum_{k=1}^{r} S^k\boldsymbol{p}_l \tag{13}$$

于是, 算法先在节点 k 上计算 $S^k\boldsymbol{p}_l$, 然后由式 (13), 所有节点规约得到 $S\boldsymbol{p}_l$. 由于是对线性空间复杂度的数据进行规约, 计算速度很快.

3) 其他步骤. 其他步骤类似于 PCG 算法, 全都在控制节点上进行. 由于是向量运算, 计算速度很快. 在某些集群配置上, 多核 CPU 和多个 GPU 可用于对这一步的加速.

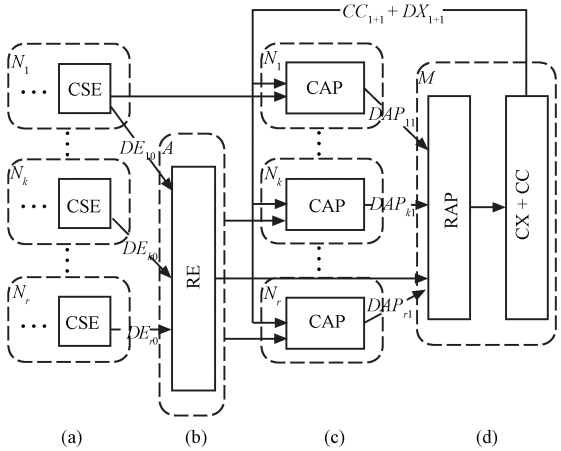


图 2 简单 DPCG 算法

Fig. 2 Simple DPCG algorithm

2.7.2 分组 DPCG 算法

为加速计算 $S\boldsymbol{p}_l$, 这里给出一种分组 DPCG 算法. 将 r 个计算节点分成 q 个计算组, 第 i 组有 r_i $(i = 1, 2, \cdots, q)$ 个计算节点, 表示为节点 i_1, 节点 $i_2, \cdots$, 节点 i_{r_i}, 则:

$$r = r_1 + r_2 + \cdots + r_q$$

称每一组的第一个节点为关键节点. 分组 DPCG 算法见图 3. 其中, 方框表示分组 DPCG 算法的各个步骤: (a) 初始化; (b) 计算本地 S; (c) 计算 $\boldsymbol{e_a}$; (d) 计算 $S^k\boldsymbol{p}_l$; (e) 计算 $S\boldsymbol{p}_l$ 和其他. 虚线框表示运行各个步骤的节点或计算组. N 表示计算节点; M 表示控制节点; A 表示所有节点; CN_i 表示第 i 个计算组. 下面介绍分组 DPCG 算法的三个主要步骤, 算法其他步骤与简单 DPCG 算法类似.

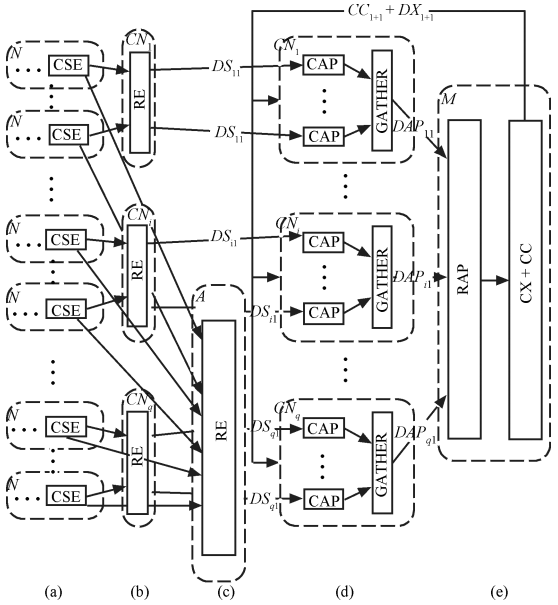


图 3 分组 DPCG 算法

Fig. 3 Group DPCG algorithm

1) 计算 $\boldsymbol{e_a}$.

所有计算节点进行规约得到向量 $\boldsymbol{e_a}$, 计算组并未使用.

2) 计算本地 S^i.

矩阵 S 可由下式计算.

$$S = \sum_{k=1}^{r} S^k = \sum_{i=1}^{q}\sum_{k=i_1}^{i_{r_i}} S^k \tag{14}$$

记矩阵 S^i 为

$$S^i = \sum_{k=i_1}^{i_{r_i}} S^k \tag{15}$$

则式 (14) 可重写为

$$S = \sum_{i=1}^{q} S^i \tag{16}$$

于是

$$S\boldsymbol{p}_l = \sum_{i=1}^{q} S^i\boldsymbol{p}_l \tag{17}$$

由式 (15), 所有第 i 个节点组的计算节点进行规约得到 S^i $(i = 1, 2, \cdots, q)$. 这一步仅需进行一次. 对于某些集群配置, 这一步的规约运算速度很快. 例如, 每个计算节点代表一个 CPU 核, 而一个计算组

代表一台计算机, 或者节点组里的计算机由一些能够进行高速通信的硬件连接.

3) 计算 $S\boldsymbol{p}_l$.

矩阵 S^i $(i=1,2,\cdots,q)$ 可表示为

$$S^i=\begin{bmatrix} S^i_1 \\ S^i_2 \\ \vdots \\ S^i_{r_i} \end{bmatrix}$$

则 $S^i\boldsymbol{p}_l$ 可算为

$$S^i\boldsymbol{p}_l=\begin{bmatrix} S^i_1 \\ S^i_2 \\ \vdots \\ S^i_{r_i} \end{bmatrix}\boldsymbol{p}_l=\begin{bmatrix} S^i_1\boldsymbol{p}_l \\ S^i_2\boldsymbol{p}_l \\ \vdots \\ S^l_{r_i}\boldsymbol{p}_l \end{bmatrix}$$

于是, 计算 $S\boldsymbol{p}_l$ 可分成三步进行: 在第 i 个节点组的第 j 个节点上计算 $S^i_j\boldsymbol{p}_l$ $(i=1,2,\cdots,q; j=1,2,\cdots,r_i)$ (由此, 各个计算节点的计算量减为简单 DPCG 的 $1/r_i$); 第 i 个关键节点 $(i=1,2,\cdots,q)$ 收集计算结果, 得到 $S^i\boldsymbol{p}_l$; 最后, 由式 (17), 对所有关键节点规约得到 $S\boldsymbol{p}_l$.

2.8 回代和算法终止条件判断

算法其他步骤包括回代 (计算 $\boldsymbol{\delta_b}$) 和算法终止条件判断. 记:

$$\boldsymbol{\delta_b}=\begin{pmatrix} \boldsymbol{\delta}^1_{\boldsymbol{b}} \\ \boldsymbol{\delta}^2_{\boldsymbol{b}} \\ \vdots \\ \boldsymbol{\delta}^r_{\boldsymbol{b}} \end{pmatrix}$$

其中

$$\boldsymbol{\delta_b}^k=\begin{pmatrix} \boldsymbol{\delta}^k_{\boldsymbol{b}_{k_0}} \\ \boldsymbol{\delta}^k_{\boldsymbol{b}_{k_1}} \\ \vdots \\ \boldsymbol{\delta}^k_{\boldsymbol{b}_{k_t}} \end{pmatrix},\quad k=1,2,\cdots,r$$

当得到 $\boldsymbol{\delta_a}$ 并将它广播到所有节点后, 可在节点 k 上计算 $\boldsymbol{\delta}^k_{\boldsymbol{b}}$. $\boldsymbol{\delta_a}$ 和 $\boldsymbol{\delta}^k_{\boldsymbol{b}}$ 将作为下次迭代的输入. 最后是算法终止条件判断. 本文算法的终止条件与文献 [6] 基本相同, 具体情况如下: 1) 给定一个最大循环次数, 当迭代次数超过该阈值时, 循环终止; 2) 对如下 4 个变量均预先设置一个阈值, $J^{\mathrm{T}}\Sigma^{-1}_{\boldsymbol{X}}\boldsymbol{\varepsilon}$ 改变量, Σ 改变量, $\Delta\boldsymbol{P}$ 改变量, ARE 相对改变量和 ARE, 当其中任意一个或多个变量小于设定的阈值时, 循环终止.

3 实现

在具体实现中, 将 DBA 算法映射到两个计算机集群上. 第一个集群由 5 台计算机组成, 每台计算机配置有 8 核的 Intel Xeon CPU E5520 2.27 GHz 所有 5 台机器共享一块 16 GB 的物理内存. 操作系统采用 CentOS.intel MPI 库[15] 作为集群通信环境. 每个 CPU 核作为一个计算节点, 共有 $8\times5=40$ 个节点. 由于各个 CPU 核的计算性能相当, 问题分解采用随机平均分配的方案, 即 $n_{\mathrm{ave}}=n_2=n_3=\cdots=n_{40}$ 和 $n_1=n_{\mathrm{ave}}+n\%40$. 对于节点计算性能不等的集群, 也可考虑采用其他分配方案. 解线性方程 $S\boldsymbol{\delta_a}=\boldsymbol{e_a}$ 采用分组 DPCG 算法. 计算节点被分为 5 个计算组, 每一组包括在同一台计算机上的节点, 则有 $r_1=r_2=\cdots=r_5=8$. 分组 DPCG 算法的最大迭代次数设为 50. 第二个集群由 3 台计算机组成, 每台计算机均配置一块 GPU. 每台计算机作为一个计算节点. 其中一台计算机配置有 8 核的 Dual Intel Xeon X5667 CPU 3.06 GHz, 一块 NVIDIA Tesla C1060 GPU 和 24 GB 内存, 将其作为控制节点. 其他两台机器配置为 4 核的 Intel Quad Q9550 CPU 2.83 GHz, 一块 NVIDIA Quadro FX 3700 GPU 和 8 GB 内存. 操作系统为 Windows7. MPICH2 库[16] 作为集群通信环境. 统一计算架构 (Compute unified device architecture, CUDA)[17] 作为 GPU 编程模型, 而 OpenMP[18] 库作为多核 CPU 编程模型. 问题分解采用随机平均分配的方案, 即 $n_{\mathrm{ave}}=n_2=n_3$ 和 $n_1=n_{\mathrm{ave}}+n\%3$. 节点的每一步具体实现见文献 [4]. 解线性方程采用简单 DPCG 算法, 最大迭代次数设为 50, 并用多核 CPU 和 GPU 对其进行加速. 实现细节及代码见: http://vision.ia.ac.cn.

4 实验结果及分析

4.1 算法

为了验证 DBA 算法的效率, 我们使用了 5 种 BA 算法: ORI-SBA、GPU-SBA、PCG-SBA、CPU-DBA 和 GPU-DBA. 它们分别定义为

ORI-SBA: 单线程 SBA 算法, 来自文献 [6].

GPU-SBA: 采用多核 CPU 和多 GPU 加速的 BA 算法[4].

PCG-SBA: 采用多核 CPU 和多 GPU 加速的 BA 算法, 解线性方程使用 PCG[4, 8].

CPU-DBA: DBA 映射到第一个集群 (见第 3 节).

GPU-DBA: DBA 映射到第二个集群 (见第 3 节).

ORI-SBA 是单线程的 SBA 算法, 同时使用了解析表达的 Jacobian 矩阵. GPU-SBA 是采用多核 CPU 和多 GPU 加速的 SBA 算法. PCG-SBA 和

GPU-SBA 的唯一不同是: 在解线性方程 $S\boldsymbol{\delta}_{\boldsymbol{a}} = \boldsymbol{e}_{\boldsymbol{a}}$ 时, PCG-SBA 使用多核 CPU 和多 GPU 加速的 PCG 算法. ORI-SBA、PCG-SBA 和 GPU-SBA 均映射到同一台计算机上, 其配置为 8 核 CPU Dual Intel Xeon X5667 CPU 3.06 GHz 和两块 GPU, 分别为 NVIDIA Tesla C1060 和 NVIDIA Quadro FX 580.CPU-DBA 和 GPU-DBA 见第 3 节.

计算结果的精确度由平均重投影误差 (Average re-projection pixel error, ARE) 来表示. 5 个算法的终止条件均相同. 即: 最大迭代次数为 100 次, 只有当 $\boldsymbol{P} + \boldsymbol{\Delta P}$ 使 ARE 更小时, 对应的 $\boldsymbol{\Delta P}$ 才被接受; 设定的各种终止条件的阈值均相同.

4.2 数据

我们在 6 组高分辨率图像数据库上进行了实验: 云岗石窟、龙泉寺、清华大学、应县木塔、邓州地形和紫霄宫. 数据库的元数据见表 1, 详细数据见我们的网站[1]. 这些图片都拍摄于中国的著名景点和建筑物. 图片纹理丰富度和分辨率各不相同. 我们用 Bundler[19] 对每个图像库进行三维重建, 在算法的最后一次迭代, 保存新添加了相机模型但未进行捆绑调整的相机参数与三维点参数, 便生成了实验所需的 MPBA 问题: 云岗石窟 BA 问题、龙泉寺 BA 问题、清华大学 BA 问题、应县木塔 BA 问题、邓州地形 BA 问题和紫霄宫 BA 问题.

表 1 SBA 算法

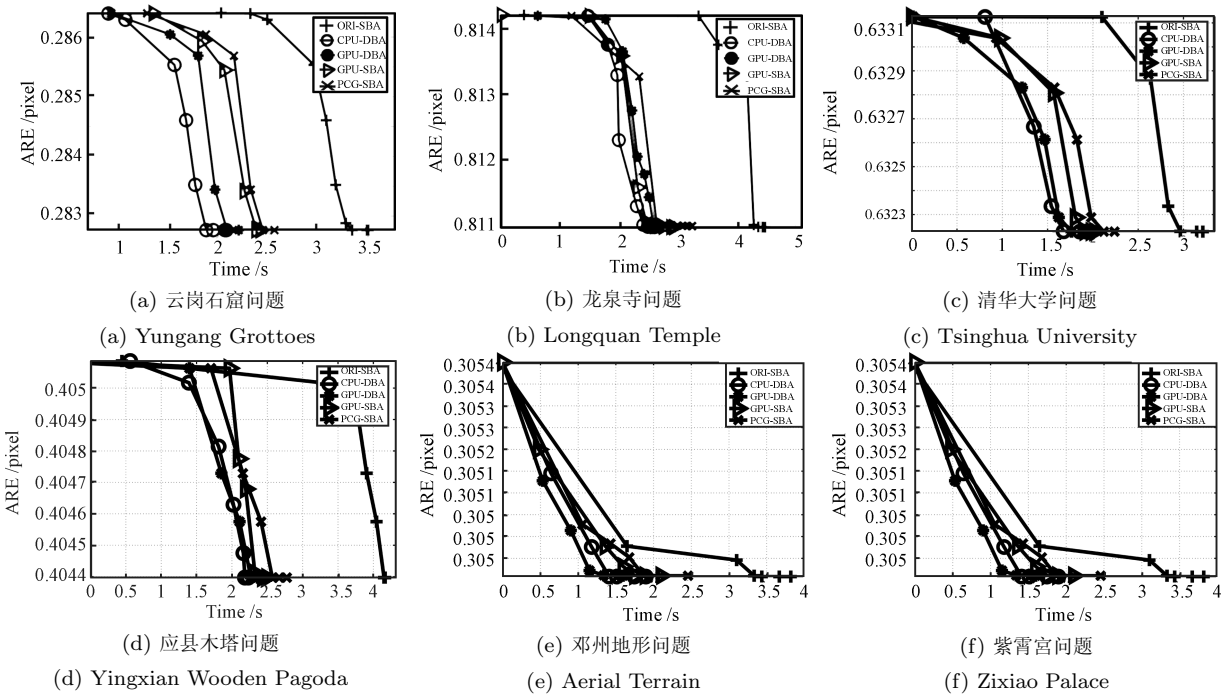
Table 1 SBA algorithms

数据库	m	n	$nvis$	分辨率
云岗石窟	143	2 476 392	9 303 676	4 368 × 2 912
龙泉寺	424	1 078 554	3 746 105	4 368 × 2 912
清华大学	234	482 731	2 538 567	4 368 × 2 912
应县木塔	344	931 078	4 576 143	4 368 × 2 912
邓州地形	776	854 224	2 537 139	1 404 × 936
紫霄宫	572	387 729	1 302 148	2 184 × 1 456

4.3 结果和性能分析

4.3.1 加速比和精确度

我们比较了 5 种算法 ARE 随计算时间的变化曲线, 实验结果见图 4. 其中, 横轴是计算时间, 以秒为单位, 对数尺度, 纵轴是 ARE. 所有算法的最终 ARE 结果相差不超过 0.0003 个像素. 可见, 5 个算法的精确度基本相同. 本文算法取得了最好的计算性能. 相比单线程 SBA, CPU-DBA 获得了 25 倍～75 倍的加速比, 而 GPU-DBA 也获得了 25 倍～55 倍的加速比. 相比以前的算法 GPU-SBA, CPU-DBA 获得了 2 倍～4 倍的速度提升. 此外, 随着 MPBA 问题规模的扩大, DBA 算法的加速比也随之增加. 这里所谓更大规模的 MPBA 问题主要是指有更多的相机模型和/或有更多的三维点. 例如, 在邓州地形 BA 问题和龙泉寺 BA 问题中, 本文算法获得了最大的速度提升.



(a) 云岗石窟问题 (b) 龙泉寺问题 (c) 清华大学问题

(a) Yungang Grottoes (b) Longquan Temple (c) Tsinghua University

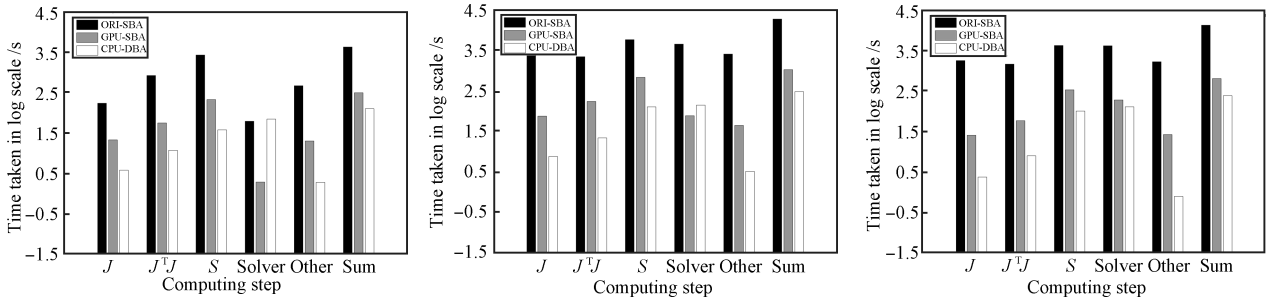
(d) 应县木塔问题 (e) 邓州地形问题 (f) 紫霄宫问题

(d) Yingxian Wooden Pagoda (e) Aerial Terrain (f) Zixiao Palace

图 4 运行时图

Fig. 4 Run-time plots

[1] http://vision.ia.ac.cn/applications/heritage_preservation/index.html



(a) 云岗石窟问题　(b) 龙泉寺问题　(c) 紫霄宫问题

(a) Yungang Grottoes　(b) Longquan Temple　(c) Zixiao Palace

图 5　算法各主要步骤性能比较

Fig. 5　Comparison of the major steps of the algorithms

4.3.2　算法各主要步骤性能比较

我们比较了 ORI-SBA、GPU-SBA 和 CPU-DBA 各个主要步骤的计算时间, 结果见图 5. 其中, 横轴表示算法各主要步骤, 包括: 计算 Jacobian、计算 $J^{\mathrm{T}}\Sigma_{\boldsymbol{X}}^{-1}J$、计算 S、解方程和其他步骤, 分别用 J, $J^{\mathrm{T}}J$, S, Solver, Other 表示, Sum 表示其计算时间之和. 纵轴表示计算时间, 是算法结束时该步计算时间的总和, 对数尺度. 最后一列表示整个算法的计算时间. 一般来说, 将基于 Schur 完形的二阶方法应用于 MPBA 问题, 计算 S 和解线性方程等步骤需要消耗大部分的计算时间[4, 9, 20]. 如图 5 所示, 在大多数的步骤中, CPU-DBA 都提高了计算的时间效率. DBA 算法对计算 S 的加速, 是其获得高加速比的主要原因. 在相机模型数较少时, GPU-SBA 能够更快地解线性方程 $S\boldsymbol{\delta_a} = \boldsymbol{e_a}$. 但随着相机模型数增加, 如在紫霞宫问题中 (图 5 (c)), DPCG 算法的性能已经超过了 GPU-SBA 的相应算法.

4.3.3　内存需求

单机内存需求是衡量 BA 算法对 BA 问题可扩展性的重要因素[12]. 各算法的单机内存需求如图 6 所示.

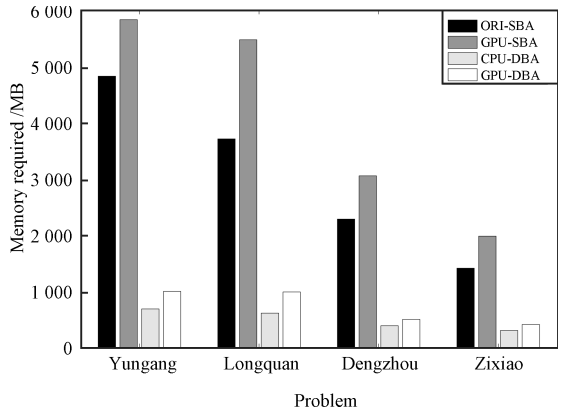


图 6　内存需求

Fig. 6　Memory requirement

其中, 横轴是几个 MPBA 问题: 云岗石窟 BA 问题、龙泉寺 BA 问题、邓州地形 BA 问题和紫霄宫 BA 问题. 纵轴是单机内存需求. 对于 MPBA 问题, 随着三维点数的增加, 各算法的内存消耗量也急剧增加. CPU-DBA 的单机内存消耗仅为 ORI-SBA 的 1/7, 而 GPU-DBA 也仅为 ORI-SBA 的 1/4. 若将 DBA 算法映射于更大规模的集群上, 其单机内存消耗量将更低. 这些特征表明, 本文算法能适用于更大规模的 MPBA 问题.

5　总结

随着计算机成本降低, 构建一个小型集群系统已不是一件非常困难的事. 对于大规模 BA 问题, 单机捆绑调整算法要么运行时间太长, 要么无法运行. 所以, 当图像多或图像分辨率高时, 需要使用集群系统, 从而需要研究分布式 BA 算法. 对于分布式 BA 算法, 所消耗的计算资源主要有两个用处: 一是提升 BA 算法的执行效率, 另一个是处理更大规模的 BA 问题. 而目前对大场景的重建, 单机处理已十分困难, 集群处理应该是一种有效途径.

针对 MPBA 问题, 本文报道了一种分布式的捆绑调整算法. 算法通过一种与问题无关的分解方法, 将原 BA 问题分解为易于计算的子问题, 并映射到各计算节点上. 针对不同的集群配置, 设计了两种分布式的线性方程解法. 算法被映射到两个集群上, 并对若干 MPBA 问题进行了实验. 与经典算法 SBA 相比, 本文算法获得了计算性能的显著提升, 并保持了较高的精确度. 本文算法对存储效率的提升, 使其能适用于更大规模的 MPBA 问题. 同时, 若增加计算资源, 算法的计算性能和存储效率将进一步提升.

另外, 当前关于 BA 问题的快速实现, 还没有一个标准平台, 使其能够在标准数据库上进行结果的定量比较. 本文仅在这方面进行了一些探讨. 对于 MPBA 问题, 基于 PCG 的 BA 算法的性能仍然是一个有待解决的开放问题, 我们将拟对这些问题进

行进一步探索.

References

1 Agarwal S, Snavely N, Simon I, Seitz S M, Szeliski R. Building rome in a day. In: Proceedings of IEEE International Conference on Computer Vision. Kyoto, Japan: IEEE, 2009. 72−79

2 Frahm J M, Georgel P, Gallup D, Johnson T, Raguram R, Wu C C, Jen Y H, Dunn E, Clipp B, Lazebnik S, Pollefeys M. Building Rome on a cloudless day. In: Proceedings of the 11th European Conference on Computer Vision: Part IV. Crete, Greece: Springer, 2010. 368−381

3 Snavely N, Seits S M, Szeliski R. Photo tourism: exploring photo collections in 3D. In: Proceedings of ACM Transactions on Graphics. New York, USA: ACM, 2006. 835−846

4 Hu Z Y, Gao W, Liu X, Guo F S. 3D Reconstruction for Heritage Preservation [Online], available: http://vision.ia.ac.cn/, March 29, 2012

5 Fang T, Quan L. Resampling structure from motion. In: Proceedings of the 11th European Conference on Computer Vision: Part II. Crete, Greece: Springer, 2010. 1−14

6 Lourakis M I A, Argyros A A. SBA: a software package for generic sparse bundle adjustment. *ACM Transactions on Mathematical Software*, 2009, **36**(1): 1−30

7 Ni K, Steedly D, Dellaert F. Out-of-core bundle adjustment for large-scale 3D reconstruction. In: Proceedings of IEEE 11th International Conference on Computer Vision. Rio de Janeiro, Brazil: IEEE, 2007. 1−8

8 Agarwal S, Snavely N, Seitz S M, Szeliski R. Bundle adjustment in the large. In: Proceedings of the 11th European Conference on Computer Vision: Part II. Crete, Greece: Springer, 2010. 29−42

9 Triggs B, McLauchlan F, Hartley R I, Fitzgibbon A W. Bundle adjustment — a modern synthesis. In: Proceedings of Vision Algorithms 1999, Lecture Notes in Computer Science (LNCS). London, UK: Springer-Verlag, 2000, 1883: 298−372

10 Byröd M, Åström K. Conjugate gradient bundle adjustment. In: Proceedings of the 11th European Conference on Computer Vision: Part II. Crete, Greece: Springer, 2010. 114−127

11 Jeong Y, Nister D, Steedly D, Szeliski R, Kweon I S. Pushing the envelope of modern methods for bundle adjustment. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. San Francisco, USA: IEEE, 2010. 1474−1481

12 Choudhary S, Gupta S, Narayanan P J. Practical time bundle adjustment for 3D reconstruction on the GPU. In: Proceedings of European Conference on Computer Vision (ECCV) 2010 Workshop on Computer Vision on GPUs. Crete, Greece: Springer, 2010. 3059

13 Wu C C, Agarwal S, Curless B, Seitz S M. Multicore bundle adjustment. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. Colorado Springs, USA: IEEE, 2011. 3057−3064

14 Rosen K H. *Discrete Mathematics and Its Applications (Fourth edition)*. USA: McGraw-Hill, 2002. 69−83

15 Intel MPI Library. Intel MPI library for Linux OS reference manual [Online], available: http://software.intel.com/en-us/articles/intel-mpi-library/, June 12, 2011

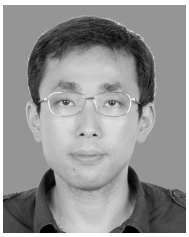
16 MPICH2 Library. MPICH2 library user′s guide [Online], available: http://www.mcs.anl.gov/research/projects/mpich2/, August 9, 2011

17 CUDA Library. CUDA programming guide 3.0 [Online], available: http://developer.download.nvidia.com/compute/cuda/30/, January 2, 2010

18 OpenMP Library. OpenMP library 3.0 user′s guide [Online], available: http://openmp.org/wp/, December 7, 2010

19 Snavely N. Bundler. The bundler user′s manual [Online], available: http://phototour.cs.washington.edu/bundler/, March 25, 2010

20 Engels C, Stewénius H, NistéŕD. Bundle adjustment rules. In: Proceedings of Photogrammetric Computer Vision. Bonn, Germany: Springer, 2011. 266−271



刘 鑫 中国科学院自动化研究所博士研究生. 2004 年获得北京师范大学信息学院计算机系学士学位. 主要研究方向为大场景三维重建和 GPU 通用计算. 本文通信作者.
E-mail: liux_skylark@tom.com
(**LIU Xin** Ph. D. candidate at the Institute of Automation, Chinese Academy of Sciences. He received his bachelor degree from Beijing Normal University in 2004. His research interest covers 3D reconstruction and GPU computing. Corresponding author of this paper.)

孙凤梅 北方工业大学教授. 主要研究方向为光学和智能信号处理.
E-mail: fmsun@163.com
(**SUN Feng-Mei** Professor at the Faculty of Sciences, North China University of Technology. Her research interest covers optics and intelligent signal processing.)

胡占义 中国科学院自动化研究所研究员. 主要研究方向为摄像机标定, 三维重建, Hough 变换, 视觉机器人导航.
E-mail: huzy@nlpr.ia.ac.cn
(**HU Zhan-Yi** Professor at the Institute of Automation, Chinese Academy of Sciences. His research interest covers camera calibration, 3D reconstruction, Hough transform, and vision guided robot navigation.)