计算机科学
COMPUTER SCIENCE

基于鲲鹏处理器的LU并行分解优化算法

徐鹤, 周涛, 李鹏, 秦芳芳, 季一木

引用本文

徐鹤, 周涛, 李鹏, 秦芳芳, 季一木. 基于鲲鹏处理器的LU并行分解优化算法[J]. 计算机科学, 2024, 51(9): 51-58.

XU He, ZHOU Tao, LI Peng, QIN Fangfang, JI Yimu. LU Parallel Decomposition Optimization Algorithm Based on Kunpeng Processor [J]. Computer Science, 2024, 51(9): 51-58.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230900079

基于鲲鹏处理器的 LU 并行分解优化算法

徐 鹤[1,2] 周 涛[1,2] 李 鹏[1,2] 秦芳芳[3] 季一木[1,2]
1 南京邮电大学计算机学院、软件学院、网络空间安全学院 南京 210023
2 江苏省高性能计算与智能处理工程研究中心 南京 210023
3 南京邮电大学理学院 南京 210023
(xuhe@njupt.edu.cn)

摘 要 ScaLAPACK(Scalable Linear Algebra PACKage)是并行计算软件包,适用于分布式存储的 MIMD(Multiple Instruction,Multiple Data)并行计算机,被广泛应用于基于线性代数运算的并行应用程序开发。然而在进行 LU 分解过程中,ScaLAPACK 库中的例程并不是通信最优的,没有充分利用当前的并行架构。针对上述问题,提出一种基于鲲鹏处理器的 LU 并行分解优化算法(Parallel LU Factorization,PLF),实现了负载均衡,适配国产鲲鹏环境。PLF 对不同进程的不同分区的数据进行差异化处理,并将每个进程所拥有的部分数据分配给根进程进行计算,之后再由根进程散播回各个子进程,这有利于充分利用 CPU 资源,实现负载均衡。在单节点 Intel 9320R 处理器以及鲲鹏(Kunpeng) 920 处理器环境中进行测试,其中,Intel 平台下使用 Intel MKL(Math Kernel Library),Kunpeng 平台下使用 PLF 算法。对比两个平台关于不同规模的方程组求解的性能发现,Kunpeng 平台的求解性能有显著优势。在 NUMA 数进程和单线程的情况下,优化后的计算效率在小规模平均达到 4.35%,相比 Intel 的 1.38%提升了 215%;中规模平均达到 4.24%,相比 Intel 平台的 1.86%提升了 118%;大规模平均达到 4.24%,相比 Intel 的 1.99%提升了 113%。

关键词:ScaLAPACK;LU 分解;并行计算;MKL
中图分类号 TP311

LU Parallel Decomposition Optimization Algorithm Based on Kunpeng Processor

XU He[1,2],ZHOU Tao[1,2],LI Peng[1,2],QIN Fangfang[3] and JI Yimu[1,2]
1 School of Computer Science,Nanjing University of Posts and Telecommunications,Nanjing 210023,China
2 Jiangsu HPC and Intelligent Processing Engineer Research Center,Nanjing 210023,China
3 College of Science,Nanjing University of Posts and Telecommunications,Nanjing 210023,China

Abstract Scalable linear algebra PACKage(ScaLAPACK) is a parallel computing package suitable for MIMD(multiple instruction,multiple data) parallel computers with distributed storage. It is widely used in parallel application program development based on linear algebra operation. However,during the LU decomposition process,the routines in the ScaLAPACK library are not communication optimal and do not take full advantage of the current parallel architecture. To solve the above problems,a parallel LU factorization optimization algorithm(PLF) based on Kunpeng processor is proposed to achieve load balancing and adapt to domestic Kunpeng environment. PLF processes the data of different partitions of different processes differently. PLF allocates part of the data of each process to the root process for calculation. After the calculation is completed,the root process spreads the data back to each sub-process,which helps to fully utilize CPU resources and achieve load balancing. Tests are performed on single-node Intel 9320R processors and Kunpeng 920 processors. Intel MKL(Math Kernel Library) is used on the Intel platform,and PLF algorithm is used on the Kunpeng platform. After comparing the performance of solving equations of different scales on two platforms,it is found that the performance of solving equations on Kunpeng platform has a significant advantage compared with

到稿日期:2023-09-14 返修日期:2023-12-22
基金项目:国家自然科学基金(62102194,62102196);江苏省六大人才高峰高层次人才项目(RJFW-111);江苏省研究生实践创新计划(SJCX22_0267,SJCX22_0275);华为鲲鹏众智计划(2022 外 241,2022 外 243)
The work was supported by the National Natural Science Foundation of China(62102194,62102196),Six Talent Peaks Project of Jiangsu Province(RJFW-111),Postgraduate Research and Practice Innovation Program of Jiangsu Province(SJCX22_0267,SJCX22_0275) and Huawei Kunpeng Zhongzhi Plan(2022w241,2022w243).
通信作者:李鹏(lipeng@njupt.edu.cn)

Intel platform. In the case of NUMA process and single thread, the optimized computing efficiency reaches 4.35% on a small scale on average, which is 215% higher than Intel's 1.38%. The average size of the medium scale reaches 4.24%, compared with 1.86% of Intel platform, an increase of 118%. The large-scale average reaches 4.24%, compared to Intel's 1.99%, an increase of 113%.

Keywords ScaLAPACK, LU factorization, Parallel computing, MKL

1 引言

LAPACK[1]是一个在节点上利用优化的 BLAS(Basic Linear Algebra Subprograms)[2]的开源库,自首次发布以来,已在单个节点上广泛使用。ScaLAPACK[3-4](Scalable Linear Algebra PACKage)与 LAPACK 功能相同,并且是为分布式内存系统设计的。它同时使用并行 BLAS 和显式分布式内存并行性[5]来扩展多节点和分布式内存结构的 LAPACK。ScaLAPACK 库[1]是密集线性代数计算的行业标准,它支持 LU,QR 和 Cholesky 分解[6]等算法,解决了奇异值、特征值、线性方程组求解和线性最小二乘等问题。ScaLAPACK 包含在苹果、AMD、康柏、富士通、日立、惠普和英特尔等公司产品的商业软件包中。大多数 Linux 发行版,包括 Cygwin, Debian 和 Fedora,也都含有 ScaLAPACK 包。ScaLAPACK 可以应用于许多领域,如科学计算、图像处理、机器学习和数据分析等[7]。

ScaLAPACK 接受密集、三对角线、双对角线、带状、对称以及三角形矩阵作为输入。为了实现高性能,ScaLAPACK 并行利用多个计算单元的计算能力,在将数据从一种布局移动到另一种布局时,以及在数据传输过程中重新排列打包和解包时,需要注意数据的分布和对齐,从而允许每个计算单元按顺序从存储器访问所需的数据。规则的数组分布可以以块、循环和块循环格式分布[8]。循环分块通过把循环划分成若干个循环块来挖掘循环中的局部性,是程序优化中的常用技术[9]。ScaLAPACK 使用块循环数据分发格式,同时,它可以将一个大型任务分解成多个子任务,由多个处理器或核心同时处理,从而加快任务的处理速度。

ScaLAPACK 中进行线性方程组求解 $\boldsymbol{AX}=\boldsymbol{B}$ 的过程分为两步:第一步是对原始矩阵 $\boldsymbol{A}$ 进行 LU 分解,即 $\boldsymbol{A}=\boldsymbol{LU}$。其中,$\boldsymbol{A}$ 是大小为 N 的方阵;$\boldsymbol{L}$ 是下三角矩阵,对角线以上的元素均为 0;$\boldsymbol{U}$ 是上三角矩阵,对角线以下的元素均为 0。第二步是使用 $\boldsymbol{L}$ 和 $\boldsymbol{U}$ 两个三角矩阵进行求解,即计算 $\boldsymbol{LUX}=\boldsymbol{B}$。ScaLAPACK 中使用的是并行化的列主元高斯消元法来进行 LU 分解。

LU 分解算法策略对线性方程组求解的性能影响较大,目前主要的 LU 分解方法有高斯消元法[10]、列主元高斯消元法[11]、追赶法[12]和雅可比迭代法[13]等。其他影响因素如寄存器块大小、缓存块大小等都会影响方程组求解的性能[14]。为了适应现代微处理器体系结构,一般通过改变算法的细节来实现算法的并行化处理,让算法在多个处理器以及多个核心上运行。

对于一些数据规模较大的矩阵,早期发行的 ScaLAPACK 库无法充分利用当前的并行架构,也很难在现在的体系结构上充分利用软件系统的硬件能力从而实现最大性能。对此,ScaLAPACK 对于具有分层内存架构的处理器,已经开发了自动内核调优软件,如 PHiPAC[15]和 ATLAS[16]。它们通过自调优线性代数软件来生成核,不同的算法产生的核对不同的矩阵维度可能并不总是最优的,这取决于所涉及的矩阵的形状。为了优化性能,必须对手工编码的微核进行量化。目前,Intel AVX 指令被用于提高内核的性能[17-21]。由于 ScaLAPACK 的算法并不是通信最优的,因此本文从算法这一方面进行优化。

本文第 2 章介绍了 LU 分解算法的相关工作;第 3 章介绍了 ScaLAPACK 中线性方程组求解使用的算法、流程等;第 4 章介绍了算法的实验设计及结果分析;最后总结全文,并提出下一步工作的方向。

2 相关工作

2.1 并行 LU 分解算法

LU 分解的过程涉及大量的矩阵乘法,其性能决定了整个线性方程组的求解性能。对于 LU 分解算法来说,最常用的是高斯消元法,但是其计算复杂度较高,特别是在矩阵规模较大时,计算效率较低。因此,在实际应用中,一般采用并行化 Hines 算法[22]等,以提高计算效率和数值稳定性。

Gnanasekaran 等[23]提出了一种低秩矩阵的分解优化算法,主要是改变了原始矩阵的分解策略,使得每个分解下来的矩阵的秩尽可能小。Dongarra 等[24]提出一种在多核架构上并行计算 LU 分解的新方法,该方法在使用面板分解的同时使用 tile 算法实现尾随子矩阵的更新,与等效的 Intel MKL 例程相比,速度提高了 40%,比采用多线程 BLAS 的 LAPACK 快了 3 倍。其主要是通过优化 BLACS(Basic Linear Algebra Communication Subprograms)中的代码,重新设计块算法,使用有效的数值内核操作小块,并通过动态调度器进行调度。Kwasniewski 等[25]提出了 conflux 算法——一种近 I/O 最优并行算法,该算法对并行 LU 分解中的不同部分采用不同的处理算法来进行优化。本文对并行 LU 分解中的不同进程采用不同的数据处理算法来进行优化。

2.2 典型 LU 分解算法

ScaLAPACK 采用了部分选主元高斯消元法,即在选主元时,只考虑当前列中除了前 k 个元素外的其余元素,其中 k 为当前消元的次数。这样可以减少主元搜索的时间,同时也可以避免数值上溢或下溢的情况发生,提高数值稳定性。

1) www.netlib.org

图 1 描述了部分选主元的变换过程。

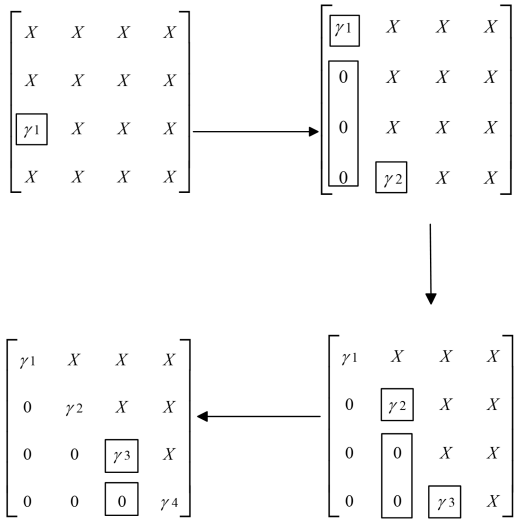

图 1　部分选主元过程

Fig. 1　Procedure of partial principal element selection

3　LU 并行分解优化算法

表 1 列出了本文使用的符号及其含义。

表 1　主要符号及其含义

Table 1　Main symbols and their meanings

符号	含义
$\boldsymbol{A}$	系数矩阵,该矩阵是带状矩阵
A_i,BL_i,BU_i,C_i,DL_i,DU_i	每个进程 i 下对系数矩阵的划分
np	进程数
$oddSize$	主分区大小
$maxBW$	系数矩阵的最大带宽
β_l	系数矩阵的下带宽
β_u	系数矩阵的上带宽
E_i,FL_i,FU_i,GL_i,GU_i,HL_i,HU_i	每个进程 i 生成还原体系过程中计算产生的矩阵
ScaLAPACK	可扩展线性代数包
BLAS	基本线性代数函数子程序
BLACS	基本线性代数通信子程序
PDDBTRF	ScaLAPACK 库中对带状矩阵进行 LU 分解并生成还原体系的例程,支持多进程
DDBTRF	LAPACK 库中对带状矩阵进行 LU 分解的例程,不支持多进程

3.1　负载均衡

负载均衡[26]是并行算法中必须面对的关键问题之一。它的核心思想是将工作量尽可能均匀地分配给各个线程,以确保所有线程在大部分时间里都保持忙碌,从而减少线程的空闲时间。实现负载均衡的方式主要有两种:平均分配任务量和动态任务分配方法。

平均分配任务量是最常见的方式,指将整个工作量平均分配给每个线程。这种方法的优点是简单易行,适用于计算量相对稳定且可预测的情况。然而,如果计算量变化较大或者无法预测,动态任务分配法可能更加合适。

动态任务分配法可以根据每个线程当前的负载情况以及系统资源的状态动态地分配任务,这种方法能够更好地应对计算量变化大或者无法预测的情况。本文采用的就是动态任务分配法。

3.2　ScaLAPACK 中的 LU 并行分解算法

为了方便讨论,本文讨论的是对角占优带状矩阵的线性方程组求解,求解的运算形式是 $\boldsymbol{AX}=\boldsymbol{B}$,其中 $\boldsymbol{A}$ 是大小为 n 的方阵;$\boldsymbol{X}$ 是行数为 n、列数为 $nrhs$ 的矩阵,是方程组求解的结果;$\boldsymbol{B}$ 是行数为 n,列数为 $nrhs$ 的矩阵,是方程组求解的右端项。矩阵 $\boldsymbol{A}$ 是带状的,有下带宽 β_l 以及上带宽 β_u。带状矩阵 $\boldsymbol{A}$ 有以下定义:

$$i>j,i-j>\beta_l\Rightarrow a_{ij}=0 \tag{1}$$

$$i\leqslant j,j-i>\beta_u\Rightarrow a_{ij}=0 \tag{2}$$

图 2 给出了带状矩阵的方程组求解形式。带状矩阵 $\boldsymbol{A}$ 的左下角以及右上角的三角形区域元素均为 0。

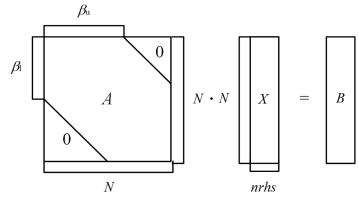

图 2　带状矩阵的方程组求解形式

Fig. 2　Solution form of equation system for band matrix

LU 分解例程 PDDBTRF 分为两个阶段:第一个阶段是还原体系的生成,只需要本地计算,并不需要借助与其他进程的通信来获取数据,该阶段中使用 DTBTRS 进行部分数据的方程组求解,该函数所占用的 CPU 资源比重较大;第二个阶段是还原体系解出,需要借助其他进程经过阶段一计算的数据。

为了调用 ScaLAPACK,首先需要将矩阵处理成 PDDBTRF 例程中所需的数据存储格式。首先将 N 阶带状矩阵 $\boldsymbol{A}$ 分解为 NP 个列块,分布到 NP 个网格上去,除最后一个进程分到 $N-NB(NP-1)$ 列数据,其余每个进程分到的列数为 NB。网格数是根据进程数 NP 划分的,由于使用的是一维网格,因此进程数就等于网格数。下面是一个 7 阶的带状矩阵 $\boldsymbol{A}$:

$$A=\begin{bmatrix} a11 & a12 & a13 & 0 & 0 & 0 & 0 \\ a21 & a22 & a23 & a24 & 0 & 0 & 0 \\ a31 & a32 & a33 & a34 & a35 & 0 & 0 \\ 0 & a42 & a43 & a44 & a45 & a46 & 0 \\ 0 & 0 & a53 & a54 & a55 & a56 & a57 \\ 0 & 0 & 0 & a64 & a65 & a66 & a67 \\ 0 & 0 & 0 & 0 & a75 & a76 & a77 \end{bmatrix}$$

图 3 给出了带状矩阵 $\boldsymbol{A}$ 在 3 个进程下的数据存储格式。其中 * 表示在内存中实际上并没用使用的空间。准备好数据之后就可以传入 PDDBTRF 例程中进行并行 LU 分解。

进程号	0			1			2
数据	*	*	a13	a24	a35	a46	a57
	*	a12	a23	a34	a45	a56	a67
	a11	a22	a33	a44	a55	a66	a77
	a21	a32	a43	a54	a65	a76	*
	a31	a42	a53	a64	a75	*	*

图 3　带状矩阵 $\boldsymbol{A}$ 在内存的存储方式

Fig. 3　Storage mode of band matrix $\boldsymbol{A}$ in memory

第一阶段是还原体系的生成,第二阶段将使用还原体系进行计算。在第一阶段中每个进程独立地与本地存储数据进行计算,计算完成后每个进程都会在本地存储该进程的还原体系。为了精确描述这一阶段的算法,以矩阵 $\boldsymbol{A}$ 为例,如图 4 所示,将每个进程上的数据划分为 A,BL,BU,C,DL,DU

6 个部分。其中 BL,BU,C,DL,DU 的大小为 $\max\{\beta_l,\beta_u\}$,最后一个进程仅有 A。

$$A=\left[\begin{array}{c|cc|ccc|c}
 & 0 & & 1 & & & 2 \\
 & A & BU & & & & \\
\hline
 & a11 & a12\ a13 & 0 & 0 & 0 & 0 \\
BL & a21 & a22\ a23 & a24 & 0 & 0 & 0 \\
 & a31 & a32\ \ C\ \ a33 & a34 & DU\ a35 & 0 & 0 \\
\hline
 & 0 & a42\ a43 & a44 & a45 & a46 & 0 \\
 & 0 & 0\ \ DL\ a53 & a54 & a55 & a56 & a57 \\
 & 0 & 0\ \ 0 & a64 & a65 & a66 & a67 \\
 & 0 & 0\ \ 0 & 0 & a75 & a76 & a77
\end{array}\right]$$

图 4 带状矩阵 $\boldsymbol{A}$ 的划分图

Fig. 4 Partition diagram of band matrix $\boldsymbol{A}$

生成还原体系第一步要对每个进程 i 的各个数据部分 A_i,BL_i,BU_i,C_i,DL_i,DU_i 进行处理。首先是对 A_i 进行 LU 分解,即 $A_i=L_iU_i$。然后利用 L_i,U_i 和 BL_i,BU_i,DL_i,DU_i,通过以下公式求解出 BU_i',GU_i,$(BL_i')^{\mathrm{T}}$,GL_i^{T}:

$$L_iBU_i'=BU_i \tag{3}$$

$$L_iGU_i=DL_i \tag{4}$$

$$U_i(BL_i')^{\mathrm{T}}=BL_i^{\mathrm{T}} \tag{5}$$

$$U_i(GL_i')^{\mathrm{T}}=DU_i^{\mathrm{T}} \tag{6}$$

下面更新 C_i 部分,即 $C_i'=C_i-BL_i'BU_i'$;生成 E_i,即 $E_i=GL_iGU_i$;生成 FL_i,即 $FL_i^{\mathrm{T}}=HU_i^{\mathrm{T}}BL_i'$;生成 FU_i,即 $FU_i=HL_i(BU_i')^{\mathrm{T}}$。至此第一阶段的还原体系生成完毕。第二阶段将还原体系解出,并将结果分发回原先的处理器。第三个阶段将第二阶段的处理结果应用于反解计算中。这种并行 LU 算法可以很好地运用于多进程的环境中,将串行的 LU 算法通过数据传输的方式并行化。

3.3 PLF 并行分解算法

LU 分解例程 PDDBTRF 的第一阶段如算法 1 所示,其中 i 表示各个进程的进程号,np 表示分配的进程数。

算法 1 PDDBTRF 例程第一阶段

功能:将带状矩阵通过 LU 分解转化为还原体系,等待 PDDBTRS 调用得出线性方程组的解

输入:进程号 i,带状矩阵 **A**,进程数 np,**A** 中不同的区域 A,BL,BU,C,DL,DU

输出:A 的还原体系

```
1. for i=0 to np-2
2.     将区域 D 发送到下一个进程
3. end for
4. 主分区 A 的 LU 分解
5. for i=0 to np-2
6.     处理 BL,BU,C
7. end for
8. for i=0 to np-1
9.     处理 DL,DU
10. end for
11. for i=0 to np-2
12.    准备还原体系
13. end for
```

算法 1 的复杂度主要取决于方阵的阶数 n 和每个进程分到的列数 NB。

算法 1 的时间复杂度在单进程环境中为 $O(n^3)$,这是因为 LU 分解的计算复杂度与矩阵的阶数 n 相关,具体为 $O(n^3)$。在多进程环境中,可以将任务分配给不同的进程,每个进程所分得的列数为 NB,时间复杂度降低到 $O(n^2*NB)$。因此,当 NB 较少时,计算效率得到显著的改善。对于相同的参数,该算法在多进程中 NB 较少,性能较好;当进程数较少时,性能较差。然而该算法的性能仍不及 Intel MKL。

算法 1 的空间复杂度主要需要约 n^2 个双精度存储单元来存储矩阵 $\boldsymbol{A}$,以及 n 个整数存储单元来存放临时变量。因此,算法 1 的空间复杂度为 $O(n^2)$。

每个进程分到的列数 NB 指导算法在顺序内存中复制、打包和重新排列子矩阵的大小,以便并行操作。一般来说,NB 是在调用例程之前就被进程数所确定的,因此考虑从算法本身进行优化。从算法 1 可以看出,例程的首个进程和最后一个进程与其他进程的计算过程有所不同。以 4 个进程为例,图 5 给出了各个进程在例程中的流程图。可以看出,在多进程的场景下,首个进程无需处理 G 部分,有一段时间处于闲置状态,因此考虑使用调度算法将其他进程的数据调度给首个进程进行处理,来平衡这一段空闲时间。

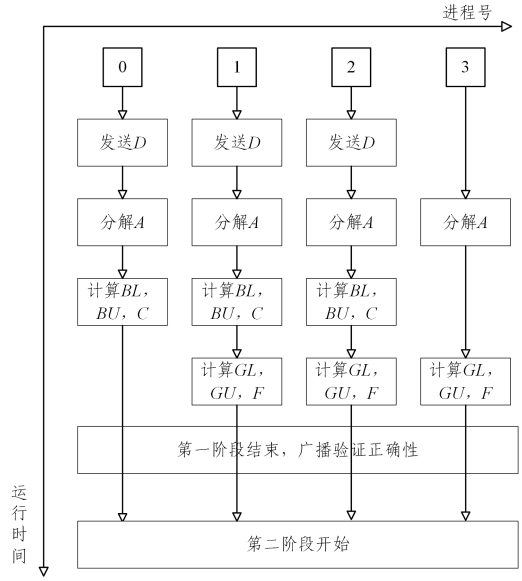

图 5 4 个进程下的各个进程的流程图

Fig. 5 Flowchart of each process under the 4 processes

需要优化的部分主要在于生成还原体系的部分代码,其中 DGEMV 例程所占用的资源较多,其伪代码如算法 2 所示。其中,m 是 $\boldsymbol{A}$ 矩阵的行数,n 是 $\boldsymbol{A}$ 矩阵的列数。从中可以看出,要更新矩阵 $\boldsymbol{Y}$,需要计算 $\boldsymbol{AX}$。$\boldsymbol{X}$ 为 n 维的列向量。在第一阶段计算 GU 的过程中,PDDBTRF 通过循环调用了 DGEMV 例程,在 $\boldsymbol{A}$ 相同的情况下,不断改变 $\boldsymbol{X}$ 来不断更新 $\boldsymbol{Y}$。在这一计算前后分别添加数据重分配操作和数据归并操作,充分利用首个进程的空闲时间。

算法 2 DGEMV 例程

功能:计算 **A**+a**AX**

输入:矩阵 **A**,向量 **X**,系数 a,矩阵 **A** 的行数 m,矩阵 **A** 的列数 n

输出:更新后的 **A**

```
1. for j=0 to n-1
2.     temp=alpha * x[jx-1];//计算出 aX
3.     for i=0 to m-1
```

```
4.      y[i]=y[i]+temp * a[j][i];//计算 A+aAX
5.    end for
6.    jx=jx+incx;//更新 X 的偏移量
7. end for
```

算法 2 的复杂度主要取决于矩阵 $\boldsymbol{A}$ 的行数 m 以及列数 n。

算法 2 的时间复杂度约为 O(mn)。在该算法中,需要对矩阵的每个元素与向量的对应元素进行乘法操作,并将结果累加。这个操作需要执行 mn 次乘法和 $mn-1$ 次加法。因此,时间复杂度为 O(mn)。

算法 2 的空间复杂度主要需要约 mn 个双精度存储单元来存储矩阵 $\boldsymbol{A}$,n 个双精度存储单元来存储向量 $\boldsymbol{X}$,以及 m 个双精度存储单元来存储向量 $\boldsymbol{Y}$。因此,空间复杂度为 O(mn)。

算法 3 展示了对 PDDBTRF 的优化,我们称优化后的算法为 PLF 算法。其中 i 表示进程号,np 代表进程数,$maxBW$ 表示矩阵的最大带宽,$send$ 表示进程需要发送的数据列数,$receive$ 表示进程需要接收的数据列数,$oddSize$ 表示进程的主分区的列数,DEGSD2D 与 DGERV2D 是 BLACS 中库函数可以将数据发送或接收到指定进程。

算法 3 是算法 1 的优化版本。在单进程环境中没有明显的优化,时间复杂度仍为 O(n^3)。主要优化多进程环境下的算法,优化后的时间复杂度由原来的 O($n^2 * NB$)降低到 O($n^2 * compute$),其中 $compute=oddSize-send$,$oddSize$ 是每个进程的主分区大小,$send$ 是需要发送给首进程计算的列数,在多进程环境下 $compute<NB$。对于空间复杂度来说,在原需要 n^2 个双精度存储单元来存储矩阵 $\boldsymbol{A}$ 的基础上,需要增加 $compute^2$ 个双精度存储单元来存放临时变量,最终空间复杂度仍为 O(n^2)。

算法 3　PLF 算法

```
功能:基于动态数据分配的 LU 并行分解算法
输入:进程号 i,进程数 np,最大带宽 maxBW,主分区大小 oddSize
输出:A 的还原体系
1. for i=0 to np-1
2.    ComputePapa(&send,&receive,n,np,oddSize,maxBW);//动
      态计算出各个进程的发送数据量以及接收数据量
3. end for
4. for i=1 to np-1
5.    DEGSD2D(……send,0,0);//除 0 进程外的进程发送数据到首
      进程
6.    ……
7.    for j=send to oddSize
8.      DGEMV();
9.    end for
10.   ……
11.   DEGSV2D(……recieve,0,0);//从首进程获取数据
12. end for
13. if i=0
14.   接收数据
15.   for j=0 to send
16.     DGEMV();
17.   end for
18.   DGESD2D(……,0,targetNP);//将计算完的数据发送回源进程
19. end if
```

首先要确定每个进程需要发送以及接收的数据列数,这与矩阵 $\boldsymbol{A}$ 的大小、带宽以及进程数、线程数有关,然后首进程运行结束后会继续接收别的进程发来的数据,来帮助别的进程进行计算。别的进程只需计算部分数据,然后接收来自首个进程的数据,即可继续执行例程之后的步骤。需要强调的是,除了首进程之外,其他进程只需发送一次数据并且接收一次数据。在计算得到结果后,通过将结果进行合并,来得到最终的计算结果。最后,各个进程根据自己所属分出去数据量的大小,将所需的数据直接接收到内存中,其流程图如图 6 所示。

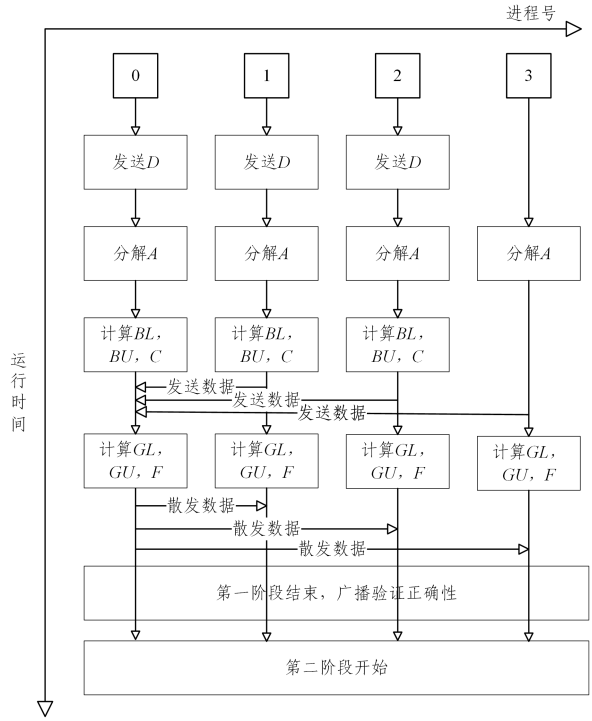

图 6　使用 PLF 算法后 4 个进程下的各个进程的流程图

Fig. 6　Flowchart of each process under the 4 processes with PLF algorithm

修改之后的主要区别是基于 BLACS 的数据进行再分配,并且在不破坏原来算法结构的基础上,应用了 4 个进程和 96 个线程,这使得在鲲鹏上进行方程组求解的性能相比 Intel MKL 提升了 30%。接下来将通过修改不同参数来与 Intel MKL 进行对比实验。

4　实验与分析

4.1　NUMA 架构

NUMA(Non-Uniform Memory Access)是一种计算机内存设计,它被用于多处理器系统。NUMA 架构将内存分为多个 NUMA 节点,每个 NUMA 节点都有自己的处理器和内存控制器。每个处理器都可以访问整个系统的内存,但是访问不同 NUMA 节点的内存延迟会有所不同。

在 NUMA 架构中,访问本 NUMA 节点内的内存比访问其他 NUMA 节点的内存更快,因为数据可以直接通过处理器和内存控制器进行传输,而不需要经过其他 NUMA 节点。这意味着如果一个处理器需要访问另一个 NUMA 节点的内存,那么这个处理器就必须等待数据从其他 NUMA 节点传输回来,这会导致延迟增加。

4.2 实验设置

由于服务器架构以及CPU等参数不同，本文采用以下公式来进行双平台之间计算效率的比较。定义函数计算效率 η 如下：

$$GFLOPS=\frac{FLOPS}{T}*10^{-9} \tag{7}$$

$$\eta=\frac{GFLOPS}{G}$$

其中，$FLOPS$ 表示例程的计算量，T 表示例程运行时间，G 表示对应服务器CPU的理论GFLOPS值。那么计算效率比 ζ 的计算式为：

$$\zeta=\frac{\max\{\eta_k,\eta_i\}}{\min\{\eta_k,\eta_i\}}-1 \tag{8}$$

其中，η_k 表示鲲鹏平台的函数计算效率，η_i 表示Intel平台的函数计算效率。计算效率比 ζ 表明了两个平台之间的函数性能差距。

选取PDDBTRF以及PDDBTRS例程进行方程组求解的测试，其中线性方程组的一次求解需要依次调用两个例程。测试规模取大、中、小3个规模，其中小规模测试的 N 的范围为10000～90000，步长为10000，中规模测试的 N 的范围为100000～500000，步长为50000，大规模测试的 N 的范围为500000～1000000，步长为1000000。每个规模下的带宽 BW 取32,48,64,128,192,224,256。对于多处理器系统，测试要考虑NUMA架构的存在，访问本NUMA节点内的内存比访问其他NUMA节点的内存更快。因此，进行测试时首先考虑将所有NUMA节点全部运行的情况，即进程数为NUMA数，让每一个进程在不同的NUMA节点中运行。其次考虑将所有的CPU核心跑满，即进程数为CPU核心数。最后考虑在NUMA节点中充分利用CPU核心，即进程数为NUMA数，线程数为NUMA内核数，这样每个进程在不同的NUMA节点的同时，由进程所产生的线程会在同一个NUMA节点中。具体的进程数、线程数等其他测试选项如表2所列。

表2 测试配置

Table 2 Testing configurations

节点数	进程数	线程数	测试规模	备注
1	NUMA数	1	全规模	每个进程绑定一个NUMA
1	96	1	全规模	
1	NUMA数	NUMA内核数	全规模	

4.3 实验环境

实验使用两台服务器：一台Kunpeng(鲲鹏)服务器，CPU是Kunpeng 920-5250[27]处理器，核数为48*2，工作频率为2.6 GHz；一台x86服务器，CPU是Intel 6230R处理器，核数为26*2，工作频率为2.6 GHz。

Kunpeng 920-5250的架构如图7所示。其中，1片系统级芯片SoC(System on Chip)上包含3个晶粒DIE，2个计算DIE，1个IO DIE。1个计算DIE中包含8个Cluster，1个Cluster中包含4个Core。因此一个Kunpeng 920芯片中包含4*8*2=64个核。计算DIE上的每一个core具有自己的L1和L2级Cache，所有的core共享L3级Cache。IO DIE上集成有网络模块和PCIe模块。这些DIE在芯片内部通过高速内部总线进行连接。Kunpeng 920-5250的单核单精度的理论算力为41.6 Gflops，单核双精度的理论算力为10.4 Gflops。Intel 6230R是一款基于Intel Xeon Scalable处理器的服务器处理器，采用了Intel自主设计的Skylake架构。Intel 6230R[28]的单核单精度的理论算力为96Gflops，单核双精度的理论算力为48 Gflops。

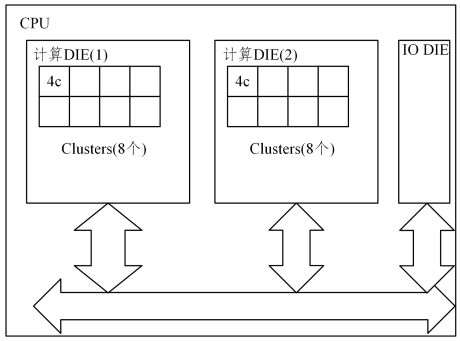

图7 Kunpeng 920-5250架构

Fig. 7 Architecture of Kunpeng 920-5250

MPI库可为超算应用提供高效的通信服务，目前已有HMPI，OpenMPI等多种实现版本[29]。本文在Kunpeng服务器上采用HMPI库，同时在x86服务器上使用OpenMPI库。

操作系统使用openEuler 20.03 LTS SP1。本研究基于ScaLAPACK 2.2.0版本。x86服务器上使用了Intel MKL库，泰山服务器上使用Kunpeng BoostKit 21.0.0中的MKL库。具体的服务器配置如表3所列。

表3 服务器配置

Table 3 Server configurations

类别	Kunpeng服务器	Intel服务器
CPU	Kunpeng920	Intel6230R
核数	48*2	26*2
频率/GHz	2.6(锁频)	2.6(锁频)
内存大小	32G*16	32G*12
操作系统	openEuler 20.03 LTS SP1	openEuler 20.03 LTS SP1
其他库	gomp+HMPI	iomp+OpenMPI

4.4 结果与分析

4.4.1 进程数为96、线程数为1

双平台下的计算效率图如图8所示，随着矩阵规模的增大，计算效率会逐渐降低。在Intel平台中，当矩阵规模到达900000附近时，计算效率会有较大波动，这是由于x86平台核心的负载过大，而鲲鹏平台由于使用了PLF算法，每个CPU核心的负载得到均衡，因此在Kunpeng平台中没有出现这一情况。

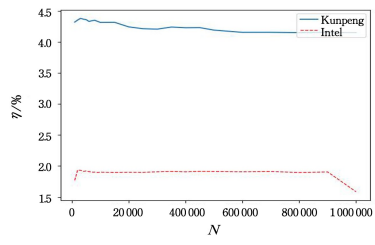

图8 进程数为96、线程数为1配置下的双平台计算效率图

Fig. 8 Computing efficiency diagram of dual-platform with 96 processes and 1 thread

计算效率比如图9所示，由于PLF算法的加入，CPU得到了充分利用，因此计算效率一直稳定在4%～4.5%。在

大规模下，由于 Intel 核数较少，因此 x86 平台下的计算效率呈下降趋势。优化后 Kunpeng 平台在小、中、大规模的计算效率平均达到 4.35%，4.24%，4.24%，相比 x86 平台的 1.38%，1.86%，1.99%分别提升了 215%，118%，113%。

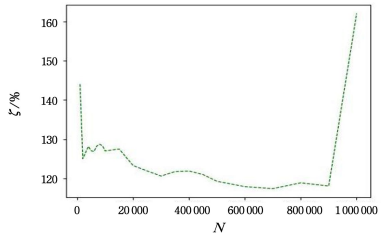

图 9　进程数为 96、线程数为 1 配置下的计算效率比

Fig. 9　Computing efficiency ratio with 96 processes and 1 thread

4.4.2　进程数为 NUMA、线程数为 1

双平台下的计算效率图如图 10 所示，当矩阵规模较小时，由于进程数比较多，每个进程所分得的数据量不会太大，此时使用 PLF 算法并不能明显改善性能。但是随着矩阵规模增大到 100 000 时，每个进程所分得的数据量变大，PLF 算法的优势得以体现。这主要是因为该实验配置下的通信次数较少，同时分配出去的数据量比较大，可以更好地实现负载均衡。

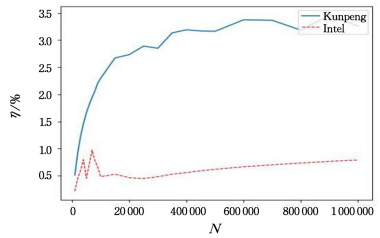

图 10　进程数为 NUMA、线程数为 1 配置下的双平台计算效率图

Fig. 10　Computing efficiency diagram under dual platform with NUMA processes and 1 thread

计算效率比如图 11 所示，在矩阵规模较小时，分配出去的计算量同样很少，所以优化后的计算效率在小规模中并没有明显效果，而在中、大规模矩阵中具有比较明显的优势。优化后 Kunpeng 平台在小、中、大规模的计算效率平均达到 1.54%，2.90%，3.33%，相比 x86 平台的 0.63%，0.52%，0.73%分别提升了 150%，456%，356%。

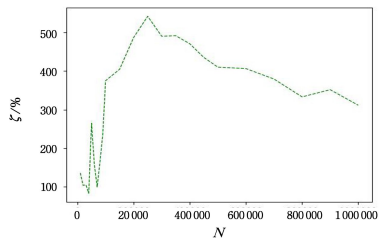

图 11　进程数为 NUMA、线程数为 1 配置下的计算效率比

Fig. 11　Computing efficiency ratio with NUMA processes and 1 threads

4.4.3　进程数为 NUMA 数、线程数为 NUMA 内核数

计算效率图如图 12 所示，由于测试配置的线程数较多，因此在小规模下计算效率会有较大的波动。当矩阵规模变大，计算效率趋于稳定。

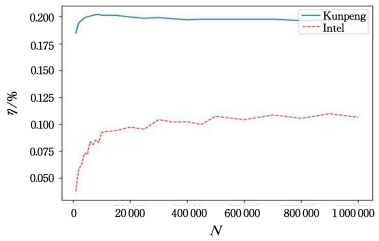

图 12　进程数为 NUMA 数、线程数为 NUMA 内核数配置下的双平台计算效率图

Fig. 12　Dual-platform computing efficiency diagram with the number of processes is NUMA and the number of threads is NUMA kernel

计算效率比如图 13 所示，在大规模下，由于每个进程以及线程所分得的数据量变大，计算效率比趋于稳定。优化后 Kunpeng 平台在小、中、大规模的计算效率平均达到 0.2%，0.2%，0.2%，相比 x86 平台的 0.07%，0.1%，0.11%分别提升了 196%，101%，85%。

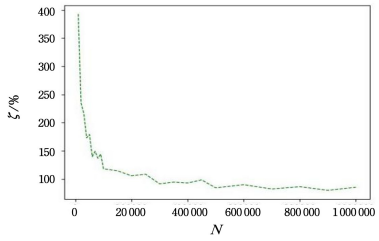

图 13　进程数为 NUMA 数、线程数为 NUMA 内核数配置下的计算效率比

Fig. 13　Computing efficiency ratio with the number of processes is NUMA and the number of threads is NUMA kernel

结束语　本文提出了一种基于鲲鹏处理器的 LU 并行分解优化算法 PLF，能够高效地对线性方程组进行求解。该算法采用动态数据分配的方法对各个进程所拥有的数据进行重新划分，有利于保持负载均衡。通过在 Kunpeng 平台以及 Intel 平台上的实验评价了算法的计算效率，实验结果表明，PLF 算法可以优化 CPU 的利用率。在未来的工作中，将关注多节点环境下使用 CPU 以及 GPU 共同加速线性方程组求解的问题。

参 考 文 献

[1]　DEMMEL J. LAPACK：a portable linear algebra library for supercomputers[C]// IEEE Control Systems Society Workshop on Computer-Aided Control System Design. USA：IEEE，1989：1-7.

[2]　BULUÇ A，GILBERT J R. The Combinatorial BLAS：Design，implementation，and applications[J]. The International Journal of High Performance Computing Applications，2011，25(4)：496-509.

[3]　CHOI J，DONGARRA J J，POZO R，et al. ScaLAPACK：A scalable linear algebra library for distributed memory concurrent computers[C]// The Fourth Symposium on the Frontiers of Massively Parallel Computation. USA：IEEE Computer Society，1992：120-127.

[4]　CHOI J，DONGARRA J J，OSTROUCHOVL S，et al. Design and Implementation of the ScaLAPACK LU，QR，and Cholesky Factorization Routines [J]. Scientific Programming，1996，5(3)：

173-184.
[5] CHOI J, DONGARRAJ J, OSTROUCHOV L S, et al. A Proposal for a set of parallel basic linear algebra subprograms[C]// Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science: Second International Workshop. Denmark: Springer, 1996: 107-114.
[6] DONGARRAJ J, HAMMARLING S, HIGHAM N J, et al. The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems [J]. Procedia Computer Science, 2017, 108: 495-504.
[7] CHEN G L, SUN G Z, ZHANG Y Q, et al. Study on Parallel Computing [J]. Journal of Computer Science and Technology, 2006, 21(5): 665-673.
[8] CHOI J, WALKER D W, DONGARRA J J. Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers [J]. Concurrency Practice & Experience, 1994, 6(7): 543-570.
[9] LUO H W, WU Y J, SHANG H H. Many-core Optimization Method for the Calculation of Ab initio Polarizability[J]. Computer Science, 2023, 50(6): 1-9.
[10] LIU L, LIU Q K, SONG X Y. Study on Parallel of Gaussian Elimination [J]. Computer Engineering, 2011, 37(8): 40-42.
[11] MEAD J L, RENAUT R A, WELFERT B D. Stability of a Pivoting Strategy for Parallel Gaussian Elimination [J]. BIT Numerical Mathematics, 2001, 41(3): 633-639.
[12] LI W Q. A Chasing Method for Solving Cyclic Tridiagonal Equations [J]. Science & Technology Review, 2009, 27(14): 69-72.
[13] KONGHUA G, HU X, ZHANG L. A new iteration method for the matrix equation AX=B [J]. Applied Mathematics and Computation, 2007, 187(2): 1434-1441.
[14] LIU J B, HE M. A Hybrid Parallelization Technique Based on OpenMP and VALU Acceleration for the Method of Moments Solution of the Surface Integral Equations [J]. Beijing Ligong Daxue Xuebao/Transaction of Beijing Institute of Technology, 2014, 34(1): 50-55.
[15] BILMES J, ASANOVIC K, CHIN C W, et al. Author retrospective for optimizing matrix multiply using PHiPAC: a portable high-performance ANSI C coding methodology[C]// ACM International Conference on Supercomputing 25th Anniversary Volume. Germany: Association for Computing Machinery, 2014: 42-44.
[16] WHALEY R C, PETITET A, DONGARRA J J. Automated empirical optimizations of software and the ATLAS project [J]. Parallel Computing, 2001, 27(1): 3-35.
[17] KIM R, CHOI J, LEE M. Optimizing parallel GEMM routines using auto-tuning with Intel AVX-512[C]// Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region. China: Association for Computing Machinery, 2019: 101-110.
[18] HASSAN S A, HEMEIDA A M, MAHMOUD M M M. Performance Evaluation of Matrix-Matrix Multiplications Using Intel's Advanced Vector Extensions (AVX) [J]. Microprocessors & Microsystems, 2016: 47(B): 369-374.
[19] LIM R, LEE Y, KIM R, et al. OpenMP-based parallel implementation of matrix-matrix multiplication on the intel knights landing[C]// Proceedings of Workshops of HPC Asia. Tokyo: Association for Computing Machinery, 2018: 63-66.
[20] GUNEY M E, GOTO K, COSTA T B, et al. Optimizing Matrix Multiplication on Intel © Xeon Phi TH x200 Architecture[C]// 2017 IEEE 24th Symposium on Computer Arithmetic (ARITH). UK: IEEE, 2017: 144-145.
[21] ZHAI X L, ZHANG Y G, JIN A Z, et al. Parallel DVB-RCS2 Turbo Decoding on Multi-core CPU [J]. Computer Science, 2023, 50(6): 22-28.
[22] ZHANG Y C, DU K, HUANG T J. Heuristic Tree-Partition-Based Parallel Method for Biophysically Detailed Neuron Simulation [J]. Neural computation, 2023, 35(4): 627-644.
[23] GNANASEKARAN A, DARVE E. Scalable low-rank factorization using a task-based runtime system with distributed memory [C]// Proceedings of the Platform for Advanced Scientific Computing Conference. Switzerland: Association for Computing Machinery, 2022: Article 8.
[24] DONGARRA J J, FAVERGE M, LTAIEF H, et al. Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting [J]. Concurrency and Computation: Practice and Experience, 2014, 26(7): 1408-1431.
[25] KWASNIEWSKI G, KABIC M, BEN-NUN T, et al. On the parallel I/O optimality of linear algebra kernels: near-optimal LU factorization[C] // Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Republic of Korea: Association for Computing Machinery, 2021: 463-464.
[26] YU T, WANG L S, QIN X L. Lock-free Parallel Semi-naive Algorithm Based on Multi-core CPU[J]. Computer Science, 2023, 50(6): 29-35.
[27] XIA J, CHENG C, ZHOU X, et al. Kunpeng 920: The first 7-nm chiplet-based 64-core arm soc for cloud services [J]. IEEE Micro, 2021, 41(5): 67-75.
[28] MOLDOVANOVA O V, KURNOSOV M G. Auto-vectorization of loops on Intel 64 and Intel Xeon Phi: Analysis and evaluation [C]// International Conference on Parallel Computing Technologies. Switzerland: Springer International Publishing, 2017: 143-150.
[29] FAN L L, QIAO Y H, LI J F, et al. CP2K Software Porting and Optimization Based on Domestic c86 Processor[J]. Computer Science, 2023, 50(6): 58-65.

XU He, born in 1985, Ph.D, professor, master supervisor, is a senior member of CCF (No. 19957S). His main research interests include big data and Internet of Things technology.

LI Peng, born in 1979, Ph.D, professor, Ph.D supervisor, is a member of CCF (No. 48573M). His main research interests include computer communication networks, cloud computing and information security.

(责任编辑:何杨)