DFCPP Runtime Library for Dataflow Programming

摘要

文章设计并实现的c++数据流是一个在通用控制流硬件平台上用于数据流计算的并行编程库。与现有的数据流编程库相比，DFCPP具有易于使用的用户界面和更丰富的表达能力，可以表达不同类型的数据流任务。同时，DFCPP为NUMA(Non-Uniform memory Access)架构设计了一个内存管理器和任务调度程序，填补了该领域内现有数据流运行库的空白。其中，内存管理器可以根据任务的需要在特定的NUMA节点上分配内存空间，而对于任务调度器而言，一方面，它可以根据数据的位置来选择任务的处理节点，同时，它使用改进的任务窃取算法来获取最合适的任务。DFCPP与内存管理器和任务调度器相结合，可以减少远程NUMA节点访问次数，从而提高执行效率。

关键词：数据流；并行计算；内存管理；

1 引言

近年来，计算机处理器的频率提升不明显，处理器的单核性能的提升也越来越困难。为了获得更好的性能，处理器普遍采用了多核设计，计算机的架构也变得越来越复杂。

传统的控制流编程语言如C语言并不适合并行编程，这就导致了Pthreads、OpenMP等并行编程库的出现。然而，这种并行编程库很难处理复杂的计算模式，同时由于控制流模型的限制，很难对任务进行精细分解来提高并行性。理论上，数据流模型更适合并行编程，同时具有开销低和内存访问要求低的优点。

因此，基于数据流模型的并行编程语言/库逐渐发展起来，如DSPatch、 TBB、Taskflow、DFC等。其中，DSPatch和DFC在表达数据流图的语义方面能力有限，不支持创建动态任务。TBB和TaskFlow的任务调度没有考虑NUMA架构的影响，效率在一定程度上受到内存访问的限制。表1总结了这些库的不同之处。

这些数据流语言的核心功能是构建数据流图程序，然后动态调度工作线程来执行程序。例如，我们可以通过这些库的函数接口来构造如图1所示的数据流图程序。通过数据流图程序，可以极大地挖掘程序的并行性。

在数据流模型中，通过数据依赖关系确定任务之间的先后执行顺序，程序运行时前驱任务输出数据使得后继任务被触发。虽然现存的数据流编程语言/库已经可以很高效地处理这种多任务并行的场景，但是当前的数据流编程库存在两个主要问题。首先，它的编程风格与传统的控制流编程有很大的不同，用户的学习成本更高，特别是对于动态数据流图的构建。其次，目前一些高性能计算机使用NUMA内存架构，且目前的数据流语言很少考虑NUMA架构的影响。

表 1. Differences of Libraries

	Support for Dynamic Task	Support for Loop Task	Support for Numa	Required Additional Code for Building Whole Graph
DFCPP	Yes	Yes	Yes	No
TBB	No	Yes	No	Yes
DSPatch	No	No	No	Yes
Taskflow	Yes	Yes	No	Yes
DARTS	Yes	Yes	No	No

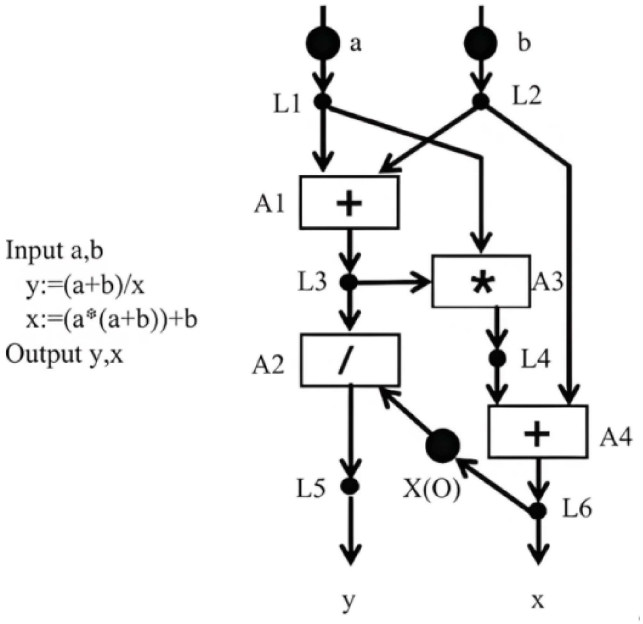

图 1. Dataflow Example

本次选择的课题论文实现了通用数据流编程库DFCPP。一方面，它简化了在数据流编程中创建数据流图的繁琐工作，用户可以通过简单的接口表达丰富的数据流图结构，包括静态任务、动态任务和条件任务。另一方面，DFCPP实现了适合NUMA体系结构的任务调度器和内存分配器，可以合理安排节点任务的分配和执行，以此来充分发挥NUMA体系结构的优势。

2 相关工作

数据流执行模型使用数据驱动的方式执行指令，指令间的相对顺序由数据依赖关系决定。在数据流编程模型中，程序一般上使用数据流图或有向无环图表示。数据流图中的节点表示一个计算任务，边则表示任务之间交换的数据，边的方向表示数据的流动方向。节点的入边表示任务的执行依赖于该数据，而出边则表示任务会在运行过程中或结束后输出该数据，以触发后续任务的运行。根据数据驱动的思想，只有当一个任务所依赖的数据全部就绪时，该任务才可以被调度执行。

2.1 DSPatch

DSPatch是一个简单的C++数据流编程框架。DSPatch围绕着“电路”和“组件”的概念设计，“电路”代表整个数据流图，“组件”则是图中的计算节点。“组件”间通过线路连接在一起，形成节点间的数据依赖。DSPatch使用缓冲区的方法，提前分配好数据的存储空间，并使用软件流水线调度方法执行任务。

2.2 TBB

TBB同样提供了用于数据流编程的库TBB Flow Graph。TBB Flow Graph提供两种任务节点source_node和function_node：source_node表示数据流图的源节点，没有输入数据，有一个输出数据；而function_node则是一般的任务节点，有一个输入数据，有一个或者没有输出数据。为了使得任务能够支持多个输入或者多个输出，还提供了join_node和split_node用于收集数据和分发数据。

2.3 OpenMP

OpenMP是一套支持跨平台共享内存方式的多线程并发编程接口。OpenMP对map、reduce、for等常用的并行算法进行了高层的抽象和封装，隐藏了底层的实现细节，大大降低了并行的难度。程序员只需要在代码通过简单的编译指导注释表明意图，编译器会将程序并行化，GCC、Clang、MSVC等主流的编译器都支持OpenMP。OpenMP主要适用于固定模式的并行任务，基于fork-join模型实现循环的并行化，将循环迭代分成多个部分交由不同的线程并行执行。

2.4 Taskflow

Taskflow是一个通用目的的并行和异构任务的编程库，使用任务依赖图构建并行应用，支持分支和循环的执行，具有丰富的表达性，能够表达复杂依赖模式的图应用。但是taskflow是基于控制依赖图的，任务之间交换数据能力薄弱，只能通过操作全局的变量，这造成了内存管理方面的困难。

3 本文方法

3.1 本文方法概述

DFCPP包含三个模块，包括编程接口模块、任务调度器和内存管理器，如图2所示。编程接口模块为用户提供了创建数据流图程序的接口，允许用户将自己的程序描绘成数据流图。任务调度器利用数据流任务的多线程并行执行，负责触发和运行任务。内存管理器负责NUMA节点中的内存分配和数据释放。其中，任务调度器和内存管理器将根据NUMA体系结构调整任务分配和执行，充分利用其性能优势。

DFCPP依赖于c++ 17中丰富的语法特性，如lambda表达式和自动元组解包，并遵循“关注任务和数据依赖关系”的设计理念。同时，它侧重于任务节点和数据之间的关系，而不是图的整体拓扑。具体来说，在DFCPP中，如果数据是任务A输出的同时又充当着任务B的输

入，则DFCPP会自动推导出任务A和任务B作为前后继的依赖关系，进而自动构建整个数据流图的拓扑结构。它与DSPatch和Taskflow有很大的不同，后者需要手动定义任务节点之间的前驱-后继关系。此外，DFCPP只有一个用于构造任务节点的核心接口位置。DFCPP负责在任务节点和运行时库之间封装和解包数据，用户只需要像编写普通c++函数一样编写任务节点，不需要继承特定的基类型，也不需要实现特定的接口。

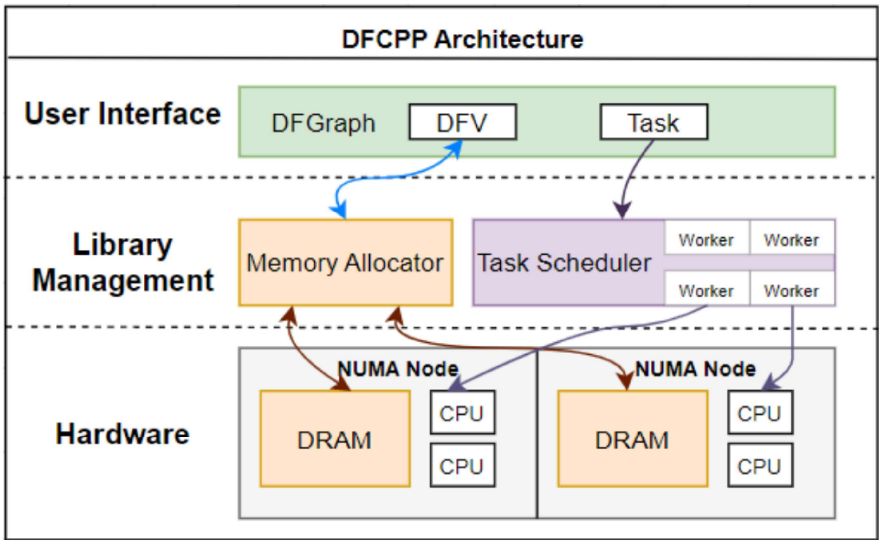

图 2. DFCPP Architecture

3.2 数据流图构建

3.2.1 数据流变量DFV

在数据流计算中有两个核心要素，一个是计算节点，另一个就是数据令牌。在DFCPP中，数据令牌以DFV的形式存在，它首先作为数据容器，其次还作为任务与任务连接关系的记录，参与任务的触发。

DFV以动态的方式管理内存，在DFV被赋值时才会申请内存用于保存数据。此外，为了节省内存开销，DFV会在数据不被访问时，及时释放对应的DFV内存空间。DFV的另外一个功能是任务间拓扑关系的记录，实际上充当着数据流图中有向边的角色，即使DFCPP中没有显式数据流图的存在。但是区别于一般的数据流语言，DFV可以同时代表多条边。更确切地说DFV代表一个具体的数据，而该数据可以同时被多个节点任务所引用，因此映射到数据流图，一个DFV就成为多条不一样的边了。同一个DFV可以被多个任务引用避免了其它数据流编程语言中每一条有向边都是独立的而带来的内存空间和数据复制开销。

3.2.2 任务节点Task实现

在创建数据流图程序时，区别于其它数据流编程库需要显式地创建节点与节点之间的连接从而构建数据流图，DFCPP并不过分关注图的整体拓扑结构，而是聚焦于节点与数据的关系。因此在DFCPP中，通过同一个DFV作为前驱任务的输出，并作为后继任务的输入从而形成前驱任务与后继任务间的数据依赖关系，进而隐式组成数据流图的拓扑结构。

代码1给出了一个Task的数据结构，其中dataDependences表示该任务输入DFV的数量，由于DFCPP引入条件任务而带来了任务间控制依赖，controlDependences用于记录任务的控制依赖计数。joinCounter为任务的触发还需要等待的依赖计数，包括了数据依赖和控制依赖。当joinCounter为零时，表示该任务的全部的输入数据已经产生，控制依赖的前驱任务也已经完成，该任务进入就绪状态，可以被调度执行。inputDFVS则是任务输入DFV的集合，主要参与DFV的内存自动释放。work则为任务的计算，包含了程序所构造的任务函数。当任务运行时，执行的就是work。Task本身没有显式地维护任务的输出DFV，通过C++的模板编程和lambda函数，work记录了任务的输入输出DFV，并将输入DFV的数据取出来交给程序员定义的函数。

Listing 1: Task 数据结构

```
class Task{
    size_t dataDependences;
    size_t controlDependences;
    std::atomic<int> joinCounter;
    std::vector<DFV> inputDFVs;
    Func work;
};
```

3.3 NUMA感知调度

DFCPP调度器以NUMA节点为基础管理工作线程，每个NUMA节点有多个工作线程。任务调度程序有两个主要功能:任务窃取和任务分配。任务窃取:和原来的随机工作窃取算法一样，每个工作线程有一个私有任务队列。工作线程首先从它们的私有任务队列中检索任务。如果私有队列为空，它们将尝试从本地节点上的其他工作线程或本地节点上的共享任务队列窃取任务。如果窃取失败，它们将尝试从其他节点窃取任务。任务分配:工作线程通过确定任务相关数据的位置，将任务存储在最佳Numa共享任务队列中。

图3所示的任务窃取过程是任务调度的核心。在工作线程完成其私有队列中的所有任务后，它就将进入窃取阶段。一般来说，线程首先从本地节点窃取任务，如果失败，则从其他NUMA节点窃取任务，从而避免了访问远程NUMA节点上的数据来执行任务的需要。为了进一步减少通过窃取线程进行连续且无效的窃取操作所造成的CPU资源浪费，线程在每次执行最大指定窃取次数后都会放弃CPU。

3.4 NUMA内存分配

内存管理是DFCPP中任务调度系统的一个重要组成部分。现有主流内存分配算法(亦或是分配器)，如GCC的ptmalloc、jemalloc和tcmalloc，是不支持numa的。默认情况下，Linux操作系统使用first-touch方法分配物理内存，但这是直接映射到内存页中的。在访问小于页面大小的数据时将导致其余数据均存储在同一个NUMA节点，即使产生这些数据的Task不在同一个NUMA节点上执行。此外，像ptmalloc这样的分配器接管了应用程序从操作系统请求和释放内存的操作，因此无法确保数据在第一次访问时存储在本地NUMA节点上。因此，DFCPP必须实现它的内存分配器。

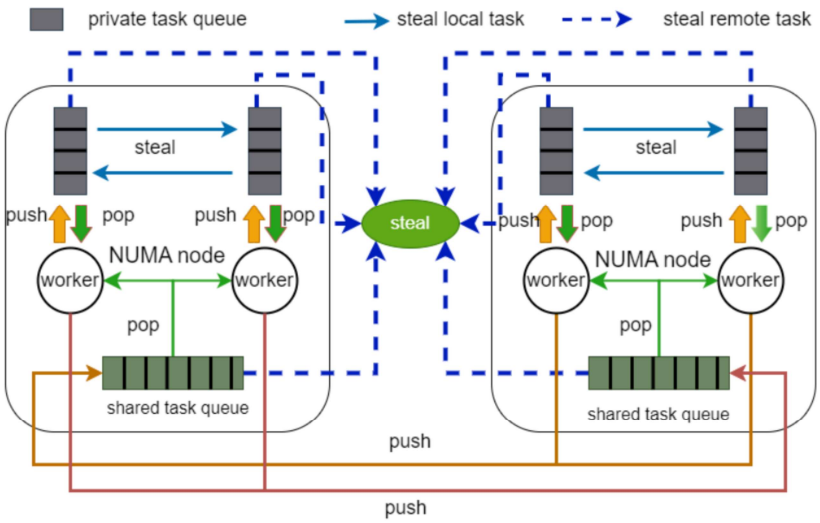

图 3. Task Scheduler Architecture

DFCPP使用Hoard内存分配算法来实现其内存分配器。Hoard是多线程应用程序的内存分配器，速度快且同步开销低，可以有效防止内存膨胀。与许多其他多线程内存分配器一样，Hoard使用多个内存堆。由Hoard管理的内存分为两部分:全局堆和线程局部堆。每个线程只能访问全局堆及其堆。Hoard设计了2P (P是处理器的数量)本地堆，并使用散列函数在线程和本地堆之间建立一对一的映射。当多个线程释放由Hoard分配的内存时，释放的内存被恢复到它所属的本地堆，而不是释放内存的线程的本地堆，以避免由于分配给不同线程而导致同一缓存线上的内存伪共享。

DFCPP在每个NUMA节点上创建Hoard内存分配器对象，每个Hoard对象负责在NUMA节点上分配和恢复内存。Hoard使用numa_alloc_onnode()和numa_free()从操作系统请求和释放固定NUMA节点上的内存(一个numa内存申请库)。当一个DFV被赋值时，如果DFV从未被赋值，它直接从本地NUMA节点上的Hoard中获取一块内存。否则，需要首先将内存恢复到NUMA节点上相应的Hoard中，然后再从本地Hoard中进行分配。简而言之，DFCPP基于Hoard内存分配算法实现了一种适用于NUMA体系结构计算机的内存分配器，可以在指定的NUMA节点上分配和恢复内存。

4 复现细节

4.1 与已有开源代码对比

本次复现工作集中于DFCPP的内存设计，修改了其真正分配DFV内存的时机(原先的设计是当前节点计算任务处理完成之后再申请对应的NUMA内存)：首次声明时与原先做法一致，只分配指针而不分配真正的内存空间，等到任务被唤醒开始执行时，在此刻进行任务节点内存的动态分配，可以有效避免原先做法在栈上分配后再迁移至全局堆时导致的二次拷贝问题(顺带修复了因程序栈限制而导致无法创建大块内存对象的Bug)。这时候，栈上的对象只有全局堆的指针，不会因Block对象过大从而占据整个栈空间引发段错误。

4.2 实验环境搭建

本实验源demo可从git@github.com:SKT-CPUOS/dfcpp.git下载，获取源码后，在根目录执行列表2所示命令即可获得示例demo。下文全部的分析及改进均基于此仓库代码。

Listing 2: DFCPP构建

```
mkdir build && cd build
cmake ..
make -j12
```

4.3 线性数据流节点数据测试

DFCPP给出了创建线性数据流计算节点的示例demo(除首节点和尾节点外，其余节点均只有一个前驱和后继)。线程先在栈上预分配计算对象，待赋值操作完成之后通过重载=号操作符将数据迁移至通过NUMA分配的内存空间中。

Listing 3: 线性数据流结点测试

```
constexpr int BLOCKSIZE = 512*512;
struct Block{
    long long data[BLOCKSIZE];
}

double measure(int n) {

    DFGraph dfGraph;
    auto dfvs = dfGraph.createDFVs<Block>(n-1);
    dfGraph.emplace(
        [](DFV<Block> out){
            Block b;
            for(int i=0; i < BLOCKSIZE; i++) {
                b.data[i] = random() % 100;
            }
           out = b;
        },
        make_tuple(), make_tuple(dfvs[0]));
    for(int i = 0; i < n - 2; i++) {
        dfGraph.emplace(
            [](const Block& b, DFV<Block> out){
                Block a;
                fflush(0);
                for(int j = 0; j < BLOCKSIZE; j++) {
                    a.data[j] = b.data[j] + random() % 100;
```

```
                }
                out = a;
            },
            make_tuple(dfvs[i]), make_tuple(dfvs[i+1]));
    }

    dfGraph.emplace(
        [](const Block& input){
            long long total
            for(int i = 0; i < BLOCKSIZE; i++) {
                total += input.data[i];
            }
        },
        make_tuple(dfvs.back()),make_tuple());

    Executor executor(1);
    auto start = chrono::steady_clock::now();
    executor.run(dfGraph).wait();
    auto end = chrono::steady_clock::now();
    return chrono::duration_cast<chrono::milliseconds> \
                (end - start).count();
}
```

这种创建节点的方式，除了会导致数据的二次拷贝问题外(见列表4)，还会由于限制单一计算节点的计算数据大小。系统中进程栈是用来存放函数调用信息和局部变量等的一块内存区域，栈的大小决定了函数调用的深度和程序使用的栈空间大小(可通过ulimit -s 查看当前终端的进程栈大小)。所以，一旦我们这里将数据块的大小BLOCKSIZE设置超过一定数值，就会因栈空间不足而引发段错误，从而导致整个程序崩溃。

Listing 4: Block 二次拷贝

```
struct Block{
    long long data[BLOCKSIZE];
    Block()
    {
        std::cout << "create function" << std::endl;
    }
    Block(const Block& other){
        std::cout << "copy function" << std::endl;
    }
};

```

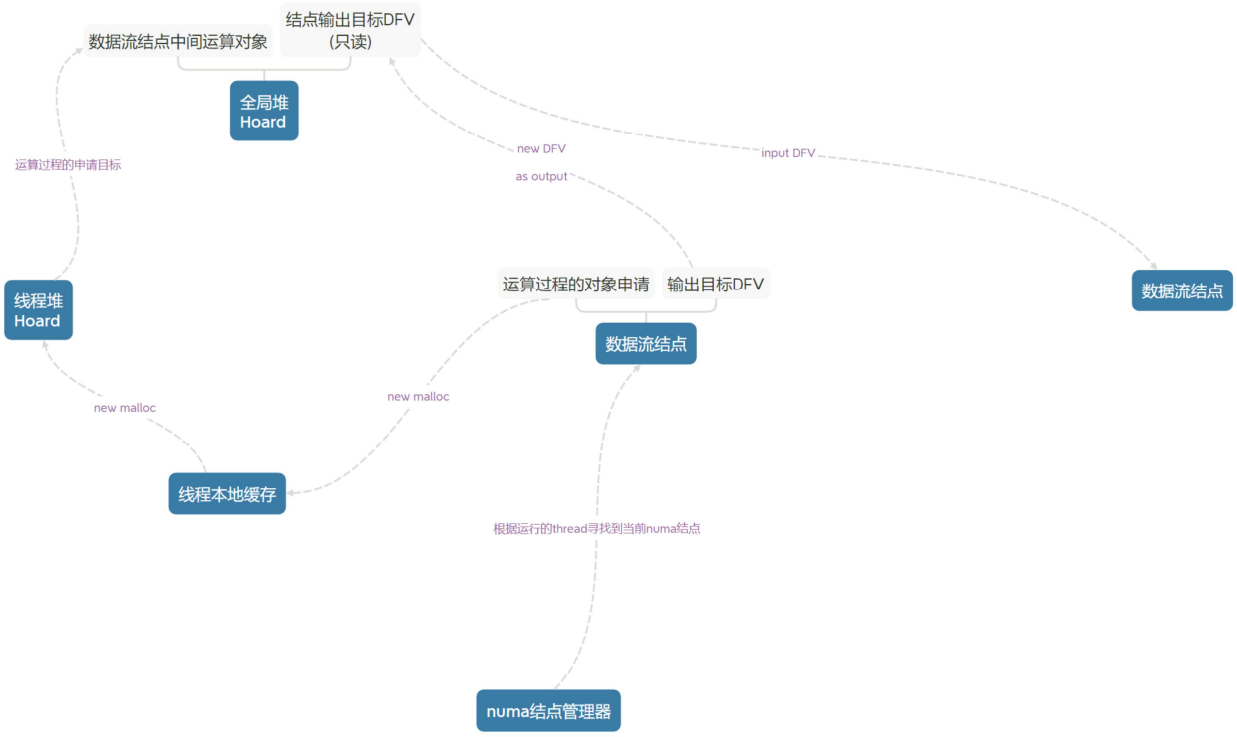

图 4. DFV内存分配

```
// output
tlx@TLX:~/project/origin_dfcpp/dfcpp/build/benchmark$ ./linear 5
pid 31154
create function
copy function
create function
copy function
create function
copy function
create function
copy function
```

经过分析之后得到的解决方案是，既然任务节点在任务完成之后，DFV数据边即为只读状态，那只要在任务激活并分发到指定线程时，在执行具体的计算任务前就提前分配内存，从而通过指针来操作数据的数据对象，即可避免因栈空间限制而段错误的问题。同时，我们是将内存分配的时间节点放置在线程准备执行该节点的时刻，故此刻我们就可以通过线程id来申请对应的NUMA内存而避免远程内存访问带来的时间开销。具体过程如图4示。

4.4 树型数据流计算节点测试

数据流计算中树型节点类型比较常见，DFCPP中也给出了相应的示例demo，如列表5所示。这里输入参数为n，表示总共构造n层的满二叉树，每个计算节点均执行随机取数操作，并与输入数据相加。

Listing 5: 树型数据流计算节点

```
double measure(int n) {
    a = 0;
    DFGraph dfGraph;
    int total = (1 << n) - 1;
    int non_leaves = (1 << (n - 1)) - 1;
    auto dfvs = dfGraph.createDFVs<Block>(non_leaves+1);
    dfGraph.emplace(
        [](DFV<Block> output){
            Block block;
            for(int i = 0; i < BLOCKSIZE; i++) {
                block.data[i] = random() % 100;
                //block.data[i] = dis(gen);
            }
            output = block;
        },
        make_tuple(),
        make_tuple(dfvs[1])
    );
    for(int i = 2; i <= non_leaves; i++) {
        dfGraph.emplace(
            [](const Block& input, DFV<Block> output){
                a++;
                Block block;
                for(int i = 0; i < BLOCKSIZE; i++) {
                    block.data[i] = input.data[i] + random() % 100;
                    //block.data[i] = input.data[i] + dis(gen);
                }
                output = block;
            },
            make_tuple(dfvs[i/2]),
            make_tuple(dfvs[i])
        );
    }
    for(int i = non_leaves+1; i <= total; i++) {
        dfGraph.emplace(
            [](const Block& input){
                long long sumup;
                for(int i = 0; i < BLOCKSIZE; i++) {
                    sumup += input.data[i];
```

```
                }
            },
            make_tuple(dfvs[i/2])
        );
    }

    Executor executor(4);
    auto start = chrono::steady_clock::now();
    executor.run(dfGraph).wait();
    auto end = chrono::steady_clock::now();
    return chrono::duration_cast<chrono::milliseconds> \
                (end - start).count();
}
```

这里的Executor executor(4)是执行器的意思，参数4表示每个NUMA节点上有4个执行线程。在实验的时候发现4线程的运行效率反而比1线程的运行效率慢，故这里使用perf工具来查看具体的原因。这里我们分别给出了1线程和4线程时的CPU火焰图(图5和图6)。

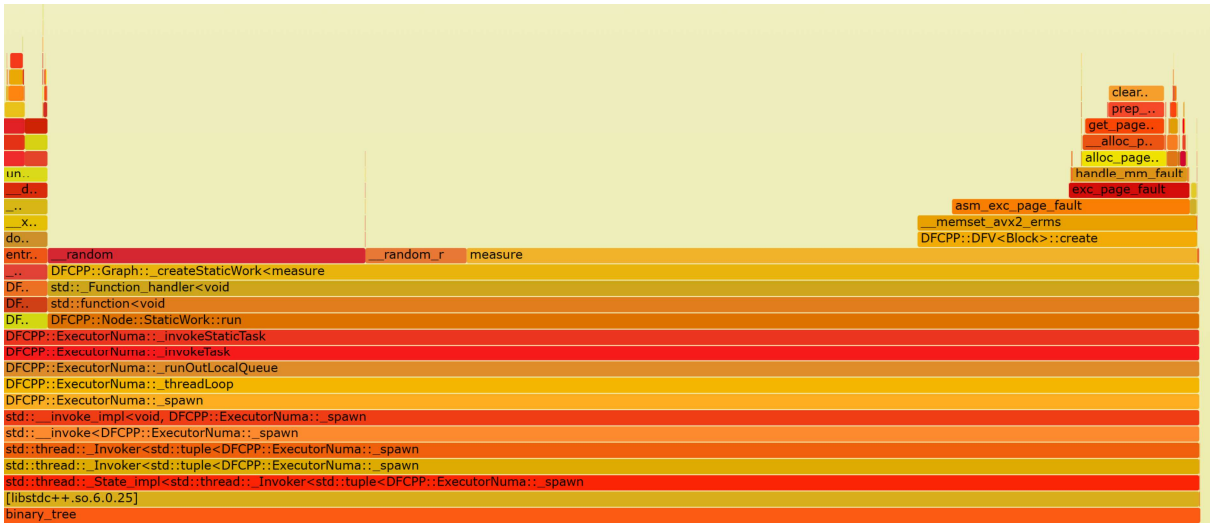

图 5. 单一线程CPU火焰图

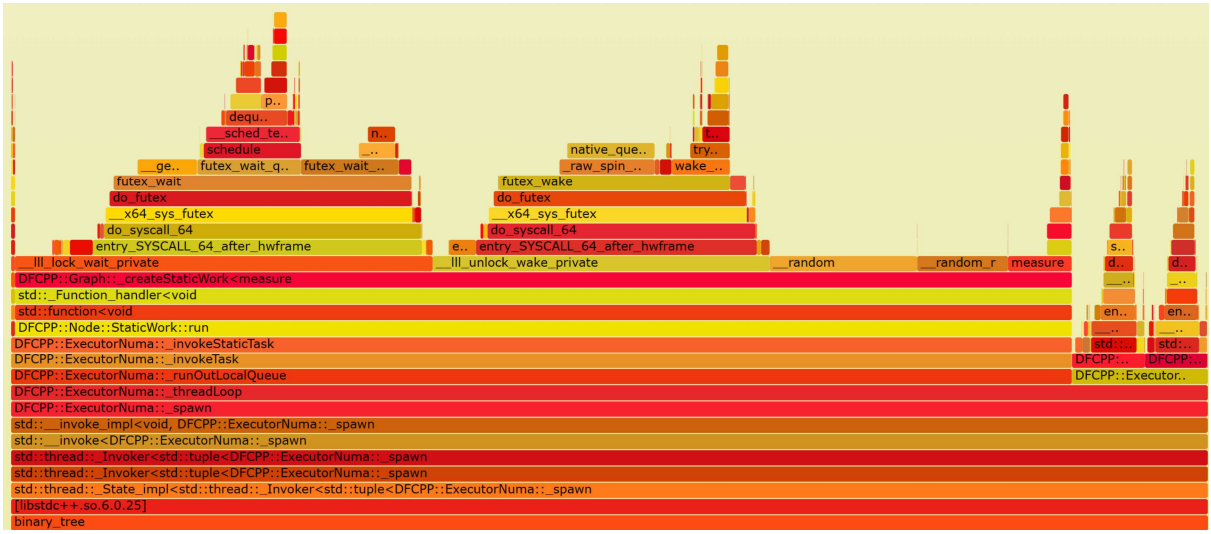

图 6. 四线程CPU火焰图

我们可以观察到，四线程时大部分时间都耗费在了__lll_lock_wait_private上，为进一步清楚调用细节，我们这里使用gdb attach去查看对应程序的调用栈，去看是哪里引发了等待操作。

Listing 6: gdb bt 函数栈分析

```
(gdb) bt
#0  __lll_lock_wait_private ()
    at ../sysdeps/unix/sysv/linux/x86_64/lowlevellock.S:95
#1  0x00007fdcf5e4edda in __random () at random.c:291
#2  0x0000561f4c007f8f in <lambda(const Block&, DFCPP::DFV<Block>)>
        ::operator()(const Block &, DFCPP::DFV<Block>)
        const (__closure=0x561f4e37fbb0, input=..., output=...)
    at /home/tlx/project/origin_dfcpp/dfcpp/benchmark/
        binary_tree.cpp:48
#3  0x0000561f4c008bbd in DFCPP::Graph::<lambda()>::operator()(void)
        (__closure=0x561f4e37fbb0)
    at /home/tlx/project/origin_dfcpp/dfcpp/dfcpp/Graph.hpp:569
#4  0x0000561f4c00a68f in std::_Function_handler<void(), DFCPP:
        :Graph::_createStaticWork(C, std::tuple<DFCPP::DFV<ArgIns>
        ...>, std::tuple<DFCPP::DFV<ArgOuts>...>,
         std::index_sequence<__indices ...>
        ... ...
        at /usr/include/c++/7/bits/std_function.h:316
#5  0x0000561f4c011642 in std::function<void ()>::operator()()
        const (this=0x561f4e37fb00)
        at /usr/include/c++/7/bits/std_function.h:706
#6  0x0000561f4c006336 in DFCPP::ExecutorNuma::_invokeStaticTask
```

```
        (this=0x561f4e4f4cd0, node=0x561f4e37fb00)
...
...
```

经查证分析后，如果数据节点内包含系统调用且当前任务数量较为稀少，而会与DFCPP中的任务窃取机制冲突(见列表7)，导致争用系统调用的互斥锁，从而引起性能下降。

Listing 7: DFCPP 任务窃取函数

```
Node *ExecutorNuma::_stealFromLocal() {
    Node* res = nullptr;
    std::uniform_int_distribution<size_t>
            uid(0, _workers.size() - 1);
    size_t numSteals = 0;
    size_t numYields = 0;
    while(!_done)
    {
        _sharedQueue.try_dequeue(res);
        if(res) break;
        size_t index = uid(_threadWorker->_engine);
        if(index != _threadWorker->_id) {
            res = _workers[index]._queue.steal();
        }
        if(res) break;
        if(numSteals++ > _maxSteals) {
            std::this_thread::yield();
            //std::this_thread::sleep_for
                    (std::chrono::milliseconds(200));
            if(numYields++ > _maxYields) {
                res = nullptr;
                break;
            }
        }
    }
    return res;
}
```

4.5 创新点

由于时间关系，此次复现工作集中在性能分析部分，对DFCPP本身结构的改进还未进行实现，将在后续的工作中重构DFCPP的调度和内存管理部分。

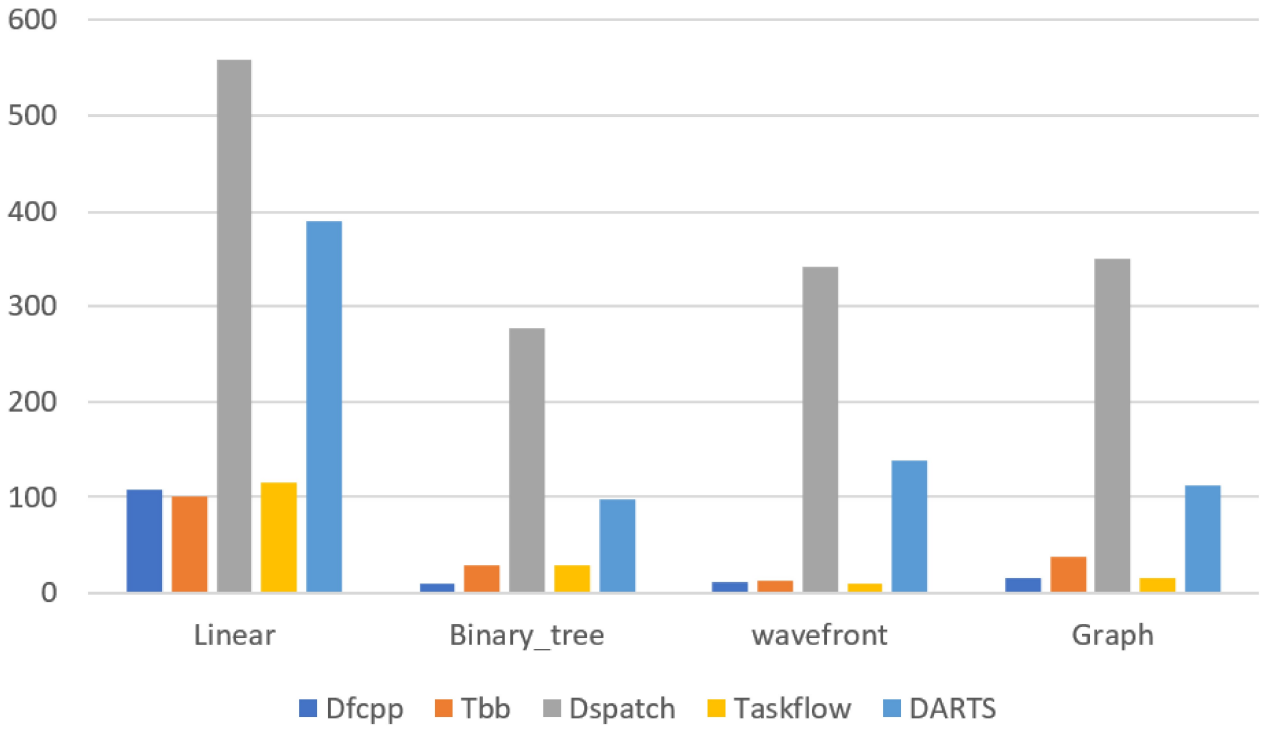

图 7. 性能对比图

5 实验结果分析

此部分的测试环境为Taishan服务器(Cpu核数96 4个numa结点单核最大频率2600MHz),本次实验的测试程序为linear、binary_tree、wavefront、graph，其中linear为96线程，块大小为512B，线性结点个数为10000；binary_tree为48线程，块大小为32B，树的层数为13；wavefront为96线程，块大小为512B，大小为100；graph为48线程，块大小为512B，参数为100 100，图7所展示出来的是DFCPP与TBB、Dspatch、Taskflow、DARTS在测试程序上的运行耗时，可以看出DFCPP在大多数情况下与Taskflow性能相近，在numa架构的机器上相比于其他流行的编程库有速度优势。

6 总结与展望

DFCPP补齐了并行编程库对于numa架构计算机支持不足的缺陷，在计算任务愈加复杂的现实背景下，numa结构的计算机一般都担任着数据计算的角色，如何有效进行计算任务的调度和数据管理是提高计算性能的一个关键部分。DFCPP本身通过numa结点感知来决定任务执行的结点，即将任务发送到依赖数据较多的numa结点上，可以有效减少计算过程中的远程内存访问导致的性能损耗。同时任务窃取算法能充分利用numa机器本身拥有较多cpu核心的优势，通过提高并发度来提升运行速度。不过经测试，无论是DFCPP还是TBB，当我们的计算任务包含有系统调用且调用频率较为频繁时，若此时的线程数远远超过任务的并发量，这时候会因为争夺系统调用锁导致运行速度骤减,这在未来是一个可以继续优化的点，同时，DFCPP只是简单地实现了数据边的内存管理，还没有真正地设计一个适用于numa结构的内存分配器，所以我之后的工作有两个部分，一个是优化任务的调度处理逻辑，另外一个则是设

计一个适用于numa结构的内存分配器，接管程序内的所有内存申请如malloc、new等的调用，将其和DFCPP数据边的管理统一起来，充分发挥numa结构的计算优势。