开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

超节点编程编译关键技术

演讲人
杨扬

演讲人职位
openEuler社区 Compiler SIG Committer

超节点给编程编译带来新的机遇

超节点硬件

TaiShan 950 SuperPoD

2026 Q1

最大16节点

最大48T内存

支持内存/SSD/DPU池化

超节点编程挑战

- **现有分布式编程模式无法利用超节点总线优势：**超节点是区别于传统集群的新硬件形态，灵衢总线提供的高带宽、低延迟和内存语义特性与共享内存编程范式亲和。传统面向多机的分布式编程模式(MPI、RDMA等)，编程无法利用灵衢总线的优势。

- **应用适配存在挑战：**通用计算/高性能计算领域，有大量存量C/C++单机多线程应用。随着业务量的增大，单机多线程已经无法满足业务需求。对应用进行“颠覆”式分布式改造会带来巨大的成本。如何将这些应用低成本、快速、高效适配超节点，是重要问题。

- **编程语言不支持超节点：C++原有接口不支持多机分布式编程，需结合第三方库/框架实现分布式并发。**这种编程方式在**编程效率**(需要组合两种编程接口)、**可移植性**(第三方库/框架缺少跨平台可移植性保证)和**性能**(节点内/节点间无法拉通统一调度)方面存在缺陷。

C++超节点并发编程编译：支持超节点应用高性能、高易用

架构图

C++超节点并发编程

C++标准兼容 | 超节点异步任务编程 | 超节点STL容器数据结构 | 超节点STL并行算法

毕昇超节点编译

单机编译优化 | 内存池化优化 | 灵衢特定优化

C++超节点运行时

全局亲和任务调动 | 超节点数据布局管理 | 灵衢亲和任务分发 | 超节点调试调优

关键技术

①C++兼容分布式编程接口：扩展C++标准多线程并发编程接口，使能分布式编程能力。使能存量多线程C++代码微改造达成分布式化。

②毕昇超节点编译：在单机编译基础上，面向灵衢内存池化场景和灵衢访存硬件特性，针对性的进行编译优化，提升灵衢内存层次下的访存效率。

③C++超节点运行时：将超节点作为一个整体，基于亲和性等信息，进行数据和任务的调度，并根据灵衢特性实现高速任务分发，提升任务执行效率。

C++超节点并发编程：单机到超节点迁移工作量<1周

超节点并发编程示例

```
template <typename T>
class BuildGraph : public bisheng::components::component_base<BuildGraph<T>> {
public:

    BuildGraph(LpasConfig *lpasConfig);
    ...
    // 任务分发函数
    int32_t TryExecuteTask(uint32_t taskNum)

    // 具体任务执行函数
    int32_t DoVectorCluster(SingleTaskInfo clusterVectorTask);
    int32_t DoBucketVector(SingleTaskInfo clusterVectorTask);
    ...
    BISHENG_DEFINE_COMPONENT_ACTION(BuildGraph, DoVectorCluster, DoVectorClusterAction)
    BISHENG_DEFINE_COMPONENT_ACTION(BuildGraph, DoBucketVector, DoBucketVectorAction)
...
}

template <typename T>
int32_t BuildGraph<T>::TryExecuteTask(uint32_t taskNum)
{

    int32_t rc = LPAS_CODE_SUCCESS;
    std::vector<bisheng::future<int32>> futures;
    for (uint64_t task_id = 0; task_id < taskNum; task_id++) {
        switch (fullTaskDesc->taskType) {
            case CLUSTER_VECTOR_TASK: {
                bisheng::future<int32_t> f = bisheng::async<DoVectorClusterAction>(targetTaskDesc);
                futures.push_back(std::move(f));
            } break;
            ...
    }
    ...
        bisheng::wait_all(futures);
        ...
}
```

通过继承，将class定义成component，使class可远程创建和访问

注册class中的相关成员函数，使能远程执行

拉起远程异步任务

等待远程异步任务完成

超节点并发编程优势

●完全基于标准C++语法

●核心函数/数据结构无需修改

●新增分布式原语基于C++标准扩展，保证最大兼容性

开放原子开源基金会 OPENATOM FOUNDATION | OpenEuler

功能介绍：异步执行

Matrix C++异步执行介绍

Matrix C++提供了单个locality之内以及多个locality之间的异步编程接口以及同步操作。

- 单个locality之内：通过bisheng::async声明一个异步操作，然后返回一个bisheng::future。对返回的bisheng::future调用get()即可进行同步。
- 多个locality之间：需要先将想要执行的函数声明为一个action，之后再以action为参数调用bisheng::async

bisheng::future定义

```
template<typename R>
class future : public bisheng::lcos::detail::future_base<future<R>, R>#
```

bisheng::async定义

```
template<typename F, typename ...Ts> decltype(auto) async(F &&f, Ts&&... ts)
```

Matrix C++编程示例

单节点内部的async异步编程以及同步操作

```
std::uint64_t fibonacci(std::uint64_t n) {
    if (n < 2)
        return n;
    bisheng::future<std::uint64_t> n1 = bisheng::async(fibonacci, n - 1);
    std::uint64_t n2 = fibonacci(n - 2);

    return n1.get() + n2;    // wait for the Future to return their values
}
```

多个locality之间的async异步编程以及同步操作：需要先将fibonacci函数声明为一个action

```
std::uint64_t fibonacci(std::uint64_t n) {
    if (n < 2)
        return n;
    bisheng::id_type const locality_id = bisheng::find_here();

    fibonacci_action fib;
    bisheng::future<std::uint64_t> n1 = bisheng::async(fib, locality_id, n - 1);
    bisheng::future<std::uint64_t> n2 = bisheng::async(fib, locality_id, n - 2);

    return n1.get() +
        n2.get();    // wait for the Futures to return their values
}
```

功能介绍：超节点STL容器

功能介绍

Matrix C++拥有自己的partition容器，这些容器同时支持local以及remote操作。每个容器可以包含多个来自不同节点的component，每个component则包含了partitioned data

支持的容器类型

容器	功能	对应的std容器
partitioned_vector	分布式vector	std::vector
unordered_map	分布式unordered map	std::unordered_map

➢每个容器的数据量可以通过构造函数中的参数进行控制。每个容器中的component的构造函数中可以控制它属于哪个locality

编程示例

```
using iterator = bisheng::partitioned_vector<T>::iterator;
using traits = bisheng::traits::segmented_iterator_traits<iterator>;

std::vector<bisheng::id_type> locs = bisheng::find_all_localities();
auto layout = bisheng::container_layout(nPart, locs);
bisheng::partitioned_vector<T> v(size, layout);
std::size_t count = 0;

auto seg_begin = traits::segment(v.begin());
auto seg_end = traits::segment(v.end());
// Iterate over partitions
for (auto seg_it = seg_begin; seg_it != seg_end; ++seg_it) {
  auto loc_begin = traits::begin(seg_it);
  auto loc_end = traits::end(seg_it);
  // Iterate over elements inside partitions
  for (auto lit = loc_begin; lit != loc_end; ++lit, ++count) {
    *lit = count;
  }
}

auto begin = v.begin();
auto end = v.end();
for (auto it = begin; it != end; ++it, ++count) {
  *it = count;
}
```

声明一个有size个元素的vector，分成nPart个partition存储，这些partition以round-robin的方式分布在所有的locality上

两段式遍历vector中的所有元素

以展平方式遍历vector中的所有元素

功能介绍：超节点STL算法

扩展C++已有executor：支持多机/异构

executor	执行语义
std::execution::seq	串行执行
std::execution::par	多线程并行
std::execution::par_unseq	多线程+向量化
std::execution::unseq	向量化
bisheng::execution::supernode	超节点并行
bisheng::execution::npu	异构

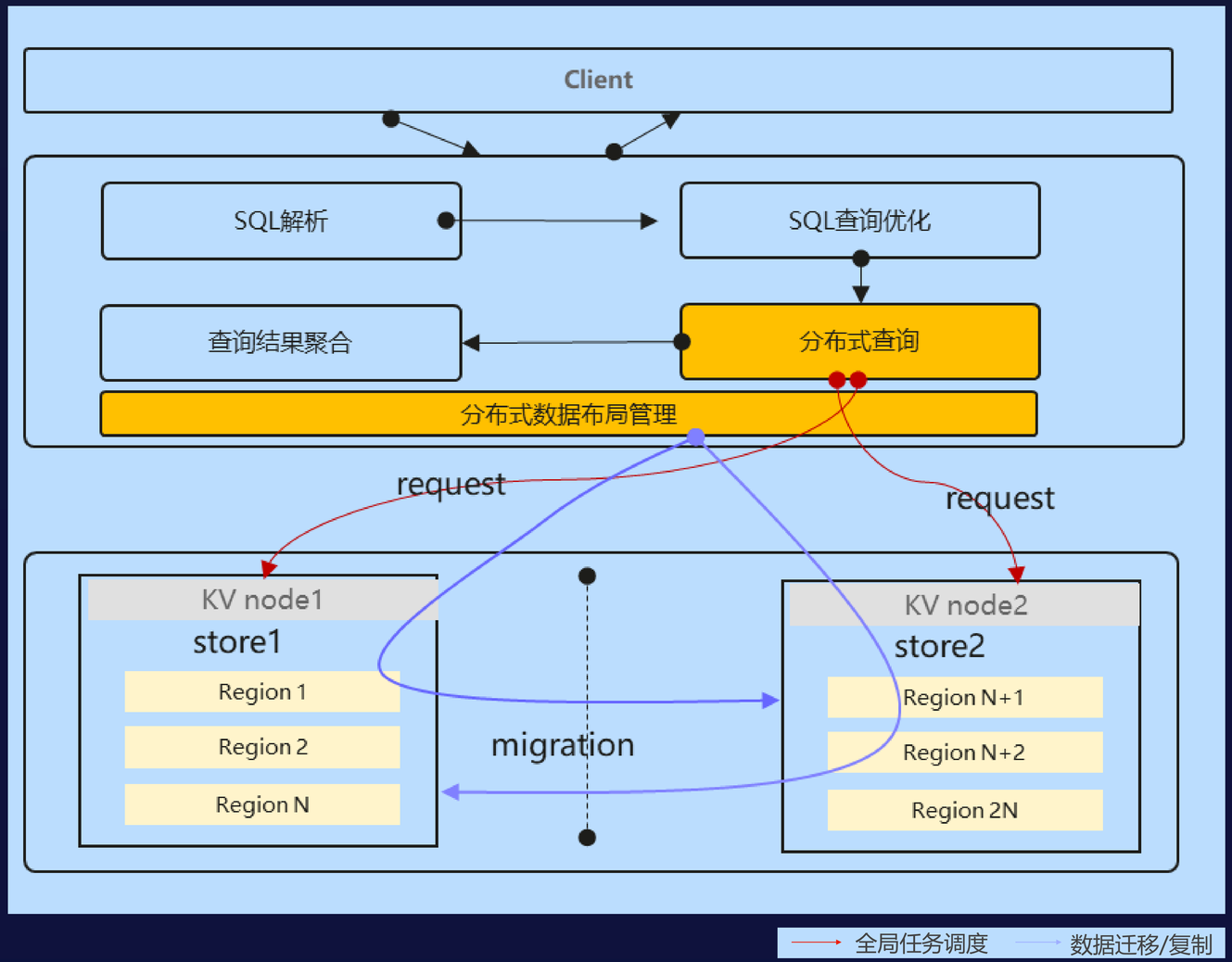
STL Parallel Algorithms

adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

my_exec = bisheng::execution::supernode;
parallel::fill(par.on(**my_exec**), begin(d), end(d), 0.0);

C++超节点运行时：全局任务调度和分布式数据布局管理

超节点运行时框架

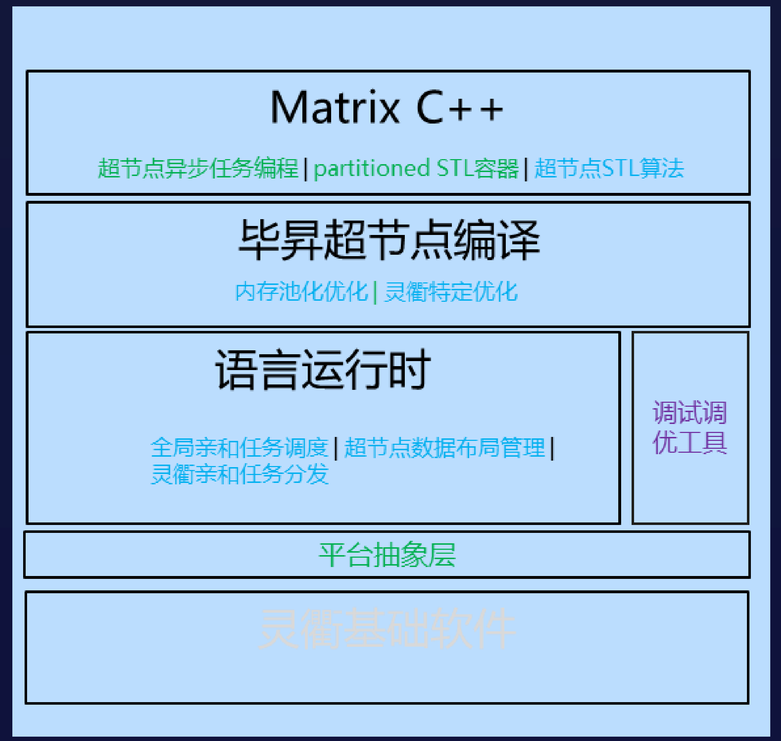


关键技术

●**数据亲和全局任务调度**：充分利用集群中的算力资源，兼顾数据访问的亲和性和全局负载平衡，通过任务调度最大化集群吞吐率

●**分布式动态数据布局管理**：根据数据访问的冷热和亲密度，在运行时动态迁移数据，以达到全局负载平衡并提供数据访问效率

分批开源计划



- 首期开源范围　首期开源代码量约3.5KLOC (25年11月)：使能内存借用场景用户无感优化；为分布式计算提供基本编程接口，使能可用性
- 二期开源范围　二期开源代码量约15KLOC(26H1)：　为分布式计算提供高级编程接口和运行时能力，使能高性能
- 三期开源范围　三期开源代码量约5KLOC(待定)：　提供调试调优能力，提升易用性

https://gitee.com/openeuler/llvm-ub

Thank You

加入我们

openEuler
Compiler SIG

毕昇编译公众号

Q&A

?