# 集合通信算法研究分析报告

*summer 2025*

**WizHUA**
NUDT 计算机学院

最初写作于：2025 年 07 月 06 日
最后更新于：2025 年 07 月 15 日

# 目录

# 章节 1. 摘要

# 章节 2. 项目概述与研究背景

# 章节 3. Open MPI 集合通信算法源码分析

基于 The Open MPI Project。（仓库链接）

## 3.1. Open MPI 架构概述

Open MPI 作为高性能计算领域的主流 MPI 实现，采用了分层的模块化组件架构(Modular Component Architecture, MCA)设计。整个架构分为三个核心层次：

- **OPAL**(Open Portability Access Layer) 提供操作系统和硬件抽象
- **OMPI** 实现 MPI 标准的核心功能
- **OSHMEM** 提供 OpenSHMEM 支持

这种分层设计确保了代码的模块化和可移植性，并且为集合通信算法的实现和优化提供了灵活的框架支持。

集合通信的核心实现位于 `ompi/mca/coll/` 目录下，包含了 `base`、`basic`、`tuned`、`han`、`xhc` 等多个专门化组件。这种组件化设计通过 `mca_coll_base_comm_select()` 机制，能够根据消息大小、进程数量、网络拓扑等运行时参数动态选择最优算法。

在此次任务中，我们专注于研究消息大小和进程数量这两个核心因素对集合通信算法选择和性能的影响，暂不涉及网络拓扑、硬件特性等复杂环境因素。因此，我们在源码分析部分将重点关注以下三个核心组件：

- `base` 组件 - 提供基础算法实现和算法选择框架
- `basic` 组件 - 包含简单可靠的参考算法实现
- `tuned` 组件 - 集成多种优化算法和智能选择机制

MCA 架构的另一个关键特性是其参数化配置系统，通过 MCA 参数可以在运行时动态调整算法选择策略、消息分片大小、通信拓扑等关键参数，同时 MPI_T 接口提供了运行时性能监控和参数调优的能力。这种设计不仅为我们的参数配置分析提供了完整的参数空间，也为机器学习模型的训练数据收集和在线预测部署提供了技术基础。通过深入分析这些组件的实现机制和参数影响，我们可以系统地理解集合通信性能的影响因素，为后续的数据集构建、特征工程和预测模型设计奠定坚实的理论基础。

## 3.2. 集合通信框架分析

集合通信框架的核心实现位于 ompi/mca/coll/base/ 目录，采用动态组件选择机制为每个通信子配置最优的集合通信实现。

### 3.2.1. 框架核心文件结构

框架的关键文件包括：

- `coll_base_functions.h` - 定义基础算法接口
  该部分定义所有集合通信操作的函数原型和参数宏（`typedef enum COLLTYPE`）；提供算法实现的标准化接口；并声明各种拓扑结构的缓存机制，提供通用的工具函数和数据结构，如二叉树（binary tree）、二项树（binomial tree）、k 进制树（k-nomial tree）、链式拓扑（chained tree）、流水线拓扑（pipeline）等。
- `coll_base_comm_select.c` - 实现组件选择机制
  该部分为每个通信子动态选择最优的集合通信组件；处理组件优先级和兼容性检查；支持运行时组件偏好设置（通过`comm->super.s_info`等机制）。
- `coll_base_util.h` - 工具函数定义
  该部分支持配置文件解析和参数处理；提供调试和监控支持。

### 3.2.2. 核心文件分析

具体而言，粗略分析这部分代码可以观察到：

1. 框架支持 MPI 标准定义的 22 种集合通信操作，通过`COLLTYPE`枚举类型进行分类管理：

```
typedef enum COLLTYPE {
    ALLGATHER = 0,       ALLGATHERV,          ALLREDUCE,
    ALLTOALL,            ALLTOALLV,           ALLTOALLW,
    BARRIER,             BCAST,               EXSCAN,
    GATHER,              GATHERV,             REDUCE,
    REDUCESCATTER,       REDUCESCATTERBLOCK,  SCAN,
    SCATTER,             SCATTERV,            NEIGHBOR_ALLGATHER,
    NEIGHBOR_ALLGATHERV, NEIGHBOR_ALLTOALL,   NEIGHBOR_ALLTOALLV,
    NEIGHBOR_ALLTOALLW,  COLLCOUNT
} COLLTYPE_T;
```

代码 3.1 `coll_base_functions.h`集合通信操作类型枚举定义

2. 每种集合通信操作都提供三个层次的接口：

- 阻塞接口：如 BCAST_ARGS，标准的同步集合通信
- 非阻塞接口：如 IBCAST_ARGS，支持异步执行和重叠计算
- 持久化接口：如 BCAST_INIT_ARGS，支持 MPI-4 的持久化集合通信

3. 为每个操作提供了丰富的算法变体实现，以 Broadcast 操作为例，框架提供了 10 种不同的算法实现：

```
/* Bcast */
int ompi_coll_base_bcast_intra_generic(BCAST_ARGS, uint32_t
count_by_segment, ompi_coll_tree_t* tree);
int ompi_coll_base_bcast_intra_basic_linear(BCAST_ARGS);
int ompi_coll_base_bcast_intra_chain(BCAST_ARGS, uint32_t segsize,
int32_t chains);
int ompi_coll_base_bcast_intra_pipeline(BCAST_ARGS, uint32_t segsize);
int ompi_coll_base_bcast_intra_binomial(BCAST_ARGS, uint32_t segsize);
int ompi_coll_base_bcast_intra_bintree(BCAST_ARGS, uint32_t segsize);
int ompi_coll_base_bcast_intra_split_bintree(BCAST_ARGS, uint32_t
segsize);
int ompi_coll_base_bcast_intra_knomial(BCAST_ARGS, uint32_t segsize,
int radix);
int ompi_coll_base_bcast_intra_scatter_allgather(BCAST_ARGS, uint32_t
segsize);
int ompi_coll_base_bcast_intra_scatter_allgather_ring(BCAST_ARGS,
uint32_t segsize);
```

代码 3.2 `coll_base_functions.h` 中的 `bcast` 的算法实现变体

同样地，Allreduce 操作提供了 7 种算法，Allgather 提供了 8 种算法，覆盖了从延迟优化到带宽优化的各种应用场景。

4. 多数算法函数支持参数化配置，如：

- segsize 参数：控制消息分段大小，影响内存使用和流水线效率
- radix 参数：控制树形算法的分支数，平衡通信轮数和并发度
- max_requests 参数：控制并发请求数量，影响内存和网络资源使用

5. 以及其它相关拓展和配置接口（如拓扑感知的算法优化等），此处略。

通过上面的分析，结合官方文档 The Modular Component Architecture (MCA) — Open MPI 5.0.x documentation，我们可以较好的理解 Open MPI 集合通信框架，并通过配置参数使用特定的算法实现来优化通信操作的性能。具体的参数配置分析见于章节 4. 集合通信参数配置分析。

## 3.3. 集合通信操作示例

为了更好地理解 Open MPI 集合通信框架的工作原理，以一个具体的 `MPI_Reduce` 操作为例，粗略分析从用户调用到底层算法执行的几个关键过程。

### 3.3.1. 用户代码调用

在用户的代码中调用 MPI 函数，假设 8 进程对一个 `int` 数据执行 Reduce 操作，有如下示例代码：

```
// size = 8
int main(int argc, char** argv) {
```

```
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int local_data = rank + 1;
    int result = 0;

    // 执行Reduce操作，求和到进程0
    MPI_Reduce(&local_data, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Sum result: %d\n", result);
    }

    MPI_Finalize();
    return 0;
}
```

代码 3.3 `MPI_Reduce` 示例代码

### 3.3.2. 初始调用暂记

该节存在较多困惑尚未解决，仅作记录用。实际的操作示例描述从 初始化阶段：组件选择机制 开始

当 `MPI_Init()` 调用时，首先被 `monitoring_prof.c` 中的监控层拦截：

```
int MPI_Init(int* argc, char*** argv)
{
    // 1. 调用实际的MPI实现
    result = PMPI_Init(argc, argv);

    // 2. 获取通信子基本信息
    PMPI_Comm_size(MPI_COMM_WORLD, &comm_world_size);
    PMPI_Comm_rank(MPI_COMM_WORLD, &comm_world_rank);

    // 3. 初始化MPI_T监控接口
    MPI_T_init_thread(MPI_THREAD_SINGLE, &provided);
    MPI_T_pvar_session_create(&session);

    // 4. 注册集合通信监控变量
    init_monitoring_result("coll_monitoring_messages_count", &coll_counts);
    init_monitoring_result("coll_monitoring_messages_size", &coll_sizes);
    start_monitoring_result(&coll_counts);
    start_monitoring_result(&coll_sizes);
}
```

代码 3.4 `MPI_Init()` 的过程

补充：这里所指的“实际的 `MPI` 实现”，区别于此处 `monitoring_prof.c` 中的 `MPI_Init()`：

1. 监控库定义：`MPI_Init` (拦截版本)
2. 真实库定义：MPI_Init (真实实现)

3. 通过 `#pragma weak: PMPI_Init -> MPI_Init` (实际实现)，实际实现在 `ompi/mpi/c/init.c.in` 中通过模板文件实现，在编译时生成实际的 c 代码。[1]
4. 通过 `LD_PRELOAD`: 用户调用先到监控版本

`init.c.in` 如下：

```
PROTOTYPE INT init(INT_OUT argc, ARGV argv)
{
    int err;
    int provided;
    int required = MPI_THREAD_SINGLE;

    // 1. 检查环境变量设置的线程级别
    if (OMPI_SUCCESS > ompi_getenv_mpi_thread_level(&required)) {
        required = MPI_THREAD_MULTIPLE;
    }

    // 2. 调用后端初始化函数
    if (NULL != argc && NULL != argv) {
        err = ompi_mpi_init(*argc, *argv, required, &provided, false);
    } else {
        err = ompi_mpi_init(0, NULL, required, &provided, false);
    }

    // 3. 错误处理
    if (MPI_SUCCESS != err) {
        // return ompi_errhandler_invoke(NULL, NULL,
OMPI_ERRHANDLER_TYPE_COMM,
                                      err < 0 ?
ompi_errcode_get_mpi_code(err) : err,
                                      FUNC_NAME);
    }

    // 4. 初始化性能计数器
    SPC_INIT();
    return MPI_SUCCESS;
}
```

代码 3.5 模板生成的 `MPI_Init` 实现

在 `ompi_mpi_init.c` 中的核心初始化过程如下：

```
int ompi_mpi_init(int argc, char **argv, int requested, int *provided, bool
reinit_ok)
{
    // 1. 状态检查和线程级别设置
    ompi_mpi_thread_level(requested, provided);

    // 2. 初始化MPI实例
    ret = ompi_mpi_instance_init(*provided, &ompi_mpi_info_null.info.super,
```

[1]报告中对此类实现相关的问题只作了简单的追踪，未进行进一步的探究

```
                                    MPI_ERRORS_ARE_FATAL,
&ompi_mpi_instance_default,
                                    argc, argv);

    // 3. 初始化通信子子系统
    if (OMPI_SUCCESS != (ret = ompi_comm_init_mpi3())) {
        error = "ompi_mpi_init: ompi_comm_init_mpi3 failed";
        goto error;
    }

    // 4. 为预定义通信子选择集合通信组件
    if (OMPI_SUCCESS != (ret = mca_coll_base_comm_select(MPI_COMM_WORLD))) {
        error = "mca_coll_base_comm_select(MPI_COMM_WORLD) failed";
        goto error;
    }

    if (OMPI_SUCCESS != (ret = mca_coll_base_comm_select(MPI_COMM_SELF))) {
        error = "mca_coll_base_comm_select(MPI_COMM_SELF) failed";
        goto error;
    }

    // 5. 标记初始化完成
    opal_atomic_swap_32(&ompi_mpi_state, OMPI_MPI_STATE_INIT_COMPLETED);
}
```

代码 3.6 `ompi_mpi_init.c` 中的核心初始化过程

### 3.3.3. 组件选择的核心机制

在 `mca_coll_base_comm_select()` 中执行具体的组件选择：

```
int mca_coll_base_comm_select(ompi_communicator_t *comm)
{
    // 1. 初始化通信子的集合通信结构
    comm->c_coll = (mca_coll_base_comm_coll_t*)calloc(1,
sizeof(mca_coll_base_comm_coll_t));

    // 2. 查询所有可用的集合通信组件（basic, tuned, han, xhc等）
    selectable =
check_components(&ompi_coll_base_framework.framework_components, comm);

    // 3. 按优先级排序并启用最高优先级组件
    for (item = opal_list_remove_first(selectable);
         NULL != item;
         item = opal_list_remove_first(selectable)) {

        mca_coll_base_avail_coll_t *avail = (mca_coll_base_avail_coll_t *)
item;
        ret = avail->ac_module->coll_module_enable(avail->ac_module, comm);

        if (OMPI_SUCCESS == ret) {
            // 4. 设置函数指针到具体实现
            if (NULL == comm->c_coll->coll_reduce) {
```

```
                comm->c_coll->coll_reduce = avail->ac_module->coll_reduce;
                comm->c_coll->coll_reduce_module = avail->ac_module;
            }
            if (NULL == comm->c_coll->coll_allgather) {
                comm->c_coll->coll_allgather = avail->ac_module->coll_allgather;
                comm->c_coll->coll_allgather_module = avail->ac_module;
            }
            // ... 为所有集合通信操作设置函数指针

            opal_list_append(comm->c_coll->module_list, &avail->super);
        }
    }

    // 5. 验证完整性 - 确保所有必需的集合通信操作都有实现
    #define CHECK_NULL(what, comm, func) \
        ((what) = # func, NULL == (comm)->c_coll->coll_ ## func)

    if (CHECK_NULL(which_func, comm, allgather) ||
        CHECK_NULL(which_func, comm, allreduce) ||
        CHECK_NULL(which_func, comm, reduce) ||
        // ... 检查其他操作
        ) {
        opal_show_help("help-mca-coll-base.txt",
                       "comm-select:no-function-available", true, which_func);
        return OMPI_ERR_NOT_FOUND;
    }

    return OMPI_SUCCESS;
}
```

代码 3.7 集合通信组件选择过程

### 3.3.4. 初始化阶段：组件选择机制

在 `MPI_Init()` 调用时，系统为 `MPI_COMM_WORLD` 选择合适的集合通信组件，这将影响后续 `MPI_Reduce` 的实现方式。

从 `init.c.in` 模板开始的调用链：

```
用户调用: MPI_Init(&argc, &argv)
    ↓
模板生成: init.c.in → MPI_Init()
    ↓
核心初始化: ompi_mpi_init(*argc, *argv, required, &provided, false)
    ↓
通信子初始化: ompi_comm_init_mpi3() [创建MPI_COMM_WORLD]
    ↓
组件选择: mca_coll_base_comm_select(MPI_COMM_WORLD)
```

在组件选择过程中，系统为 `MPI_COMM_WORLD` 设置 reduce 函数指针：

```
// 在mca_coll_base_comm_select()中
if (OMPI_SUCCESS == ret) {
    // 关键：为reduce操作设置函数指针
    if (NULL == comm->c_coll->coll_reduce) {
        comm->c_coll->coll_reduce = selected_module->coll_reduce;
        comm->c_coll->coll_reduce_module = selected_module;
    }
    // ... 其他集合通信操作的设置
}
```

代码 3.8 为 `MPI_Reduce` 设置函数指针

假设系统选择了 `basic` 组件，则根据通信子的大小决定通信算法，譬如：

```
// 在basic组件模块启用时
if (ompi_comm_size(comm) <= mca_coll_basic_crossover) {
    // 小规模通信子：使用线性算法
    BASIC_INSTALL_COLL_API(comm, basic_module, reduce,
                           ompi_coll_base_reduce_intra_basic_linear);
} else {
    // 大规模通信子：使用对数算法
    BASIC_INSTALL_COLL_API(comm, basic_module, reduce,
                           mca_coll_basic_reduce_log_intra);
}
```

代码 3.9 `basic` 组件的通信算法选择机制

对于 8 进程的情况，`mca_coll_basic_crossover (config = 4) < 8`，则会选择对数算法。

### 3.3.5. `MPI_Reduce` 的调用过程

当用户调用

`MPI_Reduce(&local_data, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)`

**第 1 步：MPI 接口层**

```
// 在ompi/mpi/c/reduce.c中
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
{
    // 参数验证：检查buffer、datatype、root等参数
    if (MPI_PARAM_CHECK) { /* ... */ }

    // 关键调用：通过函数指针调用已选定的reduce实现
    return comm->c_coll->coll_reduce(sendbuf, recvbuf, count, datatype,
                                     op, root, comm,
                                     comm->c_coll->coll_reduce_module);
}
```

代码 3.10 MPI 接口层的分发

**第 2 步：组件实现层**

假设在组件选择过程中选择了 basic 组件[2]，调用转入`mca_coll_basic_reduce_log_intra`：

```
// 在ompi/mca/coll/basic/coll_basic_reduce.c中
int mca_coll_basic_reduce_log_intra(const void *sbuf, void *rbuf,
                                    size_t count,
                                    struct ompi_datatype_t *dtype,
                                    struct ompi_op_t *op,
                                    int root,
                                    struct ompi_communicator_t *comm,
                                    mca_coll_base_module_t *module)
{
    int size = ompi_comm_size(comm);  // 8
    int rank = ompi_comm_rank(comm);
    int dim = comm->c_cube_dim;       // log2(8) = 3

    // 检查操作是否可交换
    if (!ompi_op_is_commute(op)) {
        // MPI_SUM是可交换的，所以不会走这个分支
        return ompi_coll_base_reduce_intra_basic_linear(sbuf, rbuf, count,
dtype, op, root, comm, module);
    }

    // 使用超立方体算法，执行log(N)轮通信
    int vrank = (rank - root + size) % size;  // 虚拟rank，以root为0重新编号

    for (int i = 0, mask = 1; i < dim; ++i, mask <<= 1) {
        if (vrank & mask) {
            // 高位进程向低位进程发送并停止
            int peer = ((vrank & ~mask) + root) % size;
            MCA_PML_CALL(send(snd_buffer, count, dtype, peer,
                              MCA_COLL_BASE_TAG_REDUCE,
                              MCA_PML_BASE_SEND_STANDARD, comm));
            break;
        } else {
            // 低位进程接收并归约
            int peer = vrank | mask;
            if (peer < size) {
                peer = (peer + root) % size;
                MCA_PML_CALL(recv(rcv_buffer, count, dtype, peer,
                                  MCA_COLL_BASE_TAG_REDUCE, comm,
                                  MPI_STATUS_IGNORE));
                // 执行归约操作
                ompi_op_reduce(op, rcv_buffer, rbuf, count, dtype);
            }
        }
    }
}
```

代码 3.11 `Basic`组件的`mca_coll_basic_reduce_log_intra`算法

[2]如果系统选择了不同的组件，`MPI_Reduce`的执行方式会完全不同：

- **basic 组件**：使用简单的线性收集算法（如上所示）
- **tuned 组件**：根据消息大小和进程数量选择二叉树、流水线等优化算法
- **han 组件**：使用层次化算法，先在节点内归约，再在节点间归约

但对用户而言，调用接口完全相同，这正体现了 Open MPI 组件架构的优势。

### 3.3.6. 具体执行过程

对于上述对于 8 进程的 reduce 操作（root=0），超立方体算法的通信过程如下：

轮次 1（mask=1）：处理 X 维度

```
进程1 → 进程0: 发送数据2, 结果: 进程0有(1+2)
进程3 → 进程2: 发送数据4, 结果: 进程2有(3+4)
进程5 → 进程4: 发送数据6, 结果: 进程4有(5+6)
进程7 → 进程6: 发送数据8, 结果: 进程6有(7+8)
```

轮次 2（mask=2）：处理 Y 维度

```
进程2 → 进程0: 发送(3+4), 结果: 进程0有(1+2+3+4)
进程6 → 进程4: 发送(7+8), 结果: 进程4有(5+6+7+8)
```

轮次 3（mask=4）:处理 Z 维度

```
进程4 → 进程0: 发送(5+6+7+8), 结果: 进程0有(1+2+3+4+5+6+7+8)=36
```

```
*          6----<---7    proc_0: 1+2
*         /|       /|    proc_1: 2
*        / |      / |    proc_2: 3+4
*       /  |     /  |    proc_3: 4
*      4----<---5   |    proc_4: 5+6
*      |   2--< |---3    proc_5: 6
*      |  /     |  /     proc_6: 7+8
*      | /      | /      proc_7: 8
*      |/       |/
*      0----<---1
```

代码 3.12 轮次 1 计算过程示意

### 3.3.7. 调用链总结

完整的 MPI_Reduce 调用链：

```
用户调用: MPI_Reduce(&local_data, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
    ↓
MPI接口: ompi/mpi/c/reduce.c::MPI_Reduce()
    ↓
函数指针分发: comm->c_coll->coll_reduce() [在MPI_Init时设置]
    ↓
组件实现: mca_coll_basic_reduce_intra() [basic组件为例]
    ↓
算法选择: 超立方体Reduce算法 [8进程, 基于crossover阈值]
    ↓
底层通信: MCA_PML_CALL(send/recv) [点对点消息传递]
    ↓
结果输出: 进程0获得最终结果36
```

## 3.4. 主要集合通信算法实现

该部分仅以 `Bcast`, `Scatter`, `Gather`, `Allgather`, `Reduce` 为例进行示例性的讨论。[3]

### 3.4.1. Bcast

`Bcast` 的函数原型如下：

```
MPI_Bcast(
    void* buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

其中：`buffer` 参数在根进程上包含要广播的数据，在其他进程上将接收广播的数据。`count` 参数指定数据元素的数量，`datatype` 指定数据类型，`root` 指定广播的根进程，`communicator` 指定参与通信的进程组。

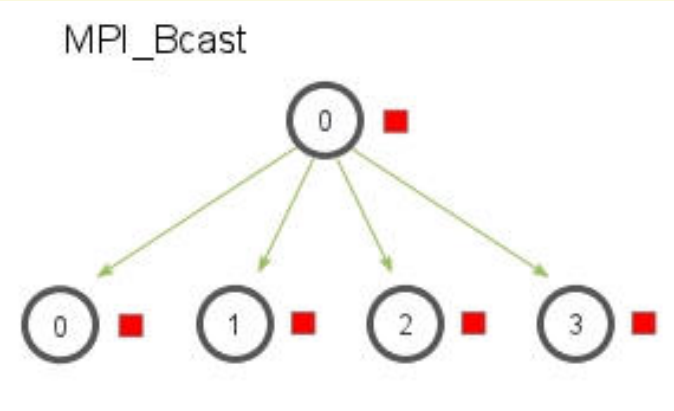



图 3.1 MPI_Bcast 通信模式图示

Open MPI 实现了多种 Bcast 算法：

#### 3.4.1.1. 线性算法（Linear Algorithm）

函数：`ompi_coll_base_bcast_intra_basic_linear()`

源码文件路径：ompi/mca/coll/base/coll_base_bcast.c

其主要原理是：根进程直接向所有其他进程发送数据。

```
// 根进程使用非阻塞发送向所有其他进程发送数据
if (rank == root) {
    // 分配请求数组
```

[3] 本章节以 Open MPI 中的 **intra-communicator**（通信子内部）为例进行集合通信算法实现的分析。这些算法用于单个通信子内部的进程间集合通信操作，如 `MPI_COMM_WORLD` 内的所有进程参与的 Broadcast、Reduce 等操作。

相对的，**inter-communicator**（通信子间）算法用于两个不同通信子之间的集合通信，属于更高级的 MPI 特性，此处不作更多讨论。

```
4       preq = reqs = ompi_coll_base_comm_get_reqs(module->base_data, size-1);
5       // 向所有非根进程发送
6       for (i = 0; i < size; ++i) {
7           if (i == rank) continue;
8           MCA_PML_CALL(isend(buff, count, datatype, i,
9                              MCA_COLL_BASE_TAG_BCAST,
10                             MCA_PML_BASE_SEND_STANDARD,
11                             comm, preq++));
12      }
13      // 等待所有发送完成
14      ompi_request_wait_all(size-1, reqs, MPI_STATUSES_IGNORE);
15  } else {
16      // 非根进程接收数据
17      MCA_PML_CALL(recv(buff, count, datatype, root,
18                        MCA_COLL_BASE_TAG_BCAST, comm,
19                        MPI_STATUS_IGNORE));
20  }
```

代码 3.13　代码示例

图示如下：

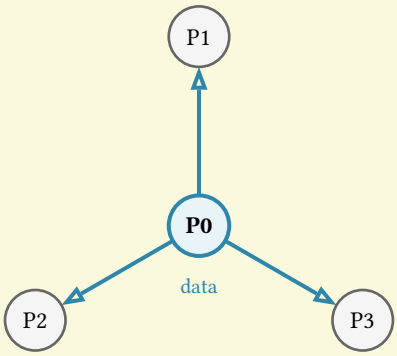



图 3.2　Broadcast 线性算法图示

算法复杂度分析：线性算法的时间复杂度为$O((p-1)\alpha+(p-1)\beta m)$，其中$\alpha$为通信启动开销，$\beta$为带宽的倒数，$p$为进程数，$m$为消息大小。该算法延迟复杂度为$O(p)$，带宽复杂度为$O(pm)$，根进程成为通信瓶颈。[4] 空间复杂度为$O(1)$，无额外空间需求。

[4]

1. 延迟复杂度（Latency Complexity）
   定义：算法中通信轮数的度量，表示串行通信步骤的数量
   表示：$O(p)$ 表示需要 $p$ 轮串行通信
   影响因素：网络启动开销 $\alpha$（每次通信的固定延迟）
2. 带宽复杂度（Bandwidth Complexity）
   定义：算法中总的数据传输量
   表示：$O(pm)$ 表示总共传输 $pm$ 单位的数据
   影响因素：网络带宽的倒数 $\beta$（传输单位数据的时间）
3. 时间复杂度（Time Complexity）
   定义：算法总执行时间的上界估计
   组成：延迟复杂度 + 带宽复杂度 + 计算复杂度，具体的，有：

$$\begin{aligned} T_{\text{total}} &= T_{\text{latency}} + T_{\text{bandwidth}} + T_{\text{computation}} \\ &= (\text{通信轮数} \cdot \alpha) + (\text{总传输量} \cdot \beta) + (\text{计算时间}) \end{aligned} \tag{3.1}$$

适用场景[5]包括小规模通信子($p \leq 4$)、极小消息大小接近延迟开销的情况、作为复杂算法的回退选择，以及网络连接性差的环境。该算法实现简单且无拓扑构建开销，但根进程瓶颈导致扩展性较差，在大规模或大消息场景下性能显著劣于树形算法。

#### 3.4.1.2. K 项树算法（K-nomial Tree Algorithm）

函数：`ompi_coll_base_bcast_intra_knomial()`

源码文件路径：ompi/mca/coll/base/coll_base_bcast.c

按照 K-nomial 树[6]结构进行数据传递，根进程作为树根，每个内部节点最多有 k 个子节点，按照树的层次结构进行数据广播。

```c
/*
 * K-nomial tree broadcast algorithm
 * radix参数控制树的分支因子
 */
int ompi_coll_base_bcast_intra_knomial(
    void *buf, size_t count, struct ompi_datatype_t *datatype, int root,
    struct ompi_communicator_t *comm, mca_coll_base_module_t *module,
    uint32_t segsize, int radix)
{
    // 构建k-nomial树
    COLL_BASE_UPDATE_KMTREE(comm, module, root, radix);
    if (NULL == data->cached_kmtree) {
        // 如果构建失败，回退到二项树
        return ompi_coll_base_bcast_intra_binomial(buf, count, datatype,
                                                   root, comm,
module, segcount);
    }

    // 使用通用的树形广播算法
    return ompi_coll_base_bcast_intra_generic(buf, count, datatype, root,
comm,
                                              module, segcount,
data->cached_kmtree);
}
```

代码 3.14　K 项树 Broadcast 算法核心代码

图示如下：

---

而在此处对通信操作复杂度的讨论中，未考虑计算时间的影响

[5]报告中此处的“适用场景”为源码注释中提到的经验结果。

[6]对于给定的 radix（分支因子）和进程数量，K 项树按以下规则构建：

1. 根节点：进程 0 作为树根
2. 子节点计算：对于节点 rank，其子节点按公式计算：

$$\text{child\_rank} = \left(\text{rank} + \frac{\text{size}}{\text{radix}^{\text{level}}} \cdot \text{i}\right) \bmod \text{size} \tag{3.2}$$

其中$i = 1, 2, ..., \min(\text{radix}, \text{remaining_nodes})$

3. 层次分配：节点按二进制表示的最高位分组到不同层次

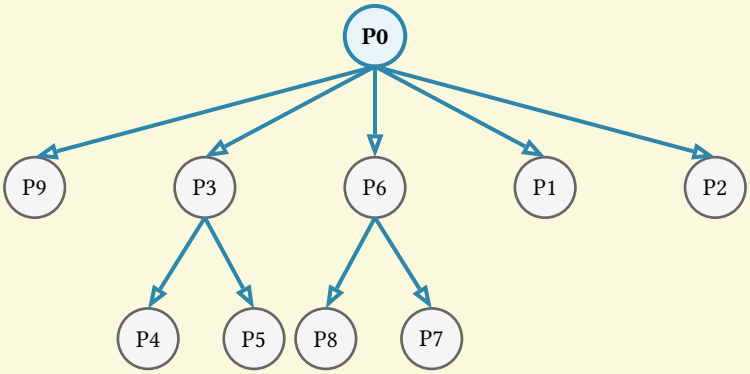



图 3.3 K 项树 Broadcast 算法的树形结构（radix=3）

算法复杂度分析：K 项树算法的时间复杂度为$O(\log_k(p)\alpha + \beta m)$，其中`radix`参数$k$控制分支因子。延迟复杂度为$O(\log_k(p))$，随着$k$增大而减小，但单节点负载增加；带宽复杂度为$O(m)$，每个消息只传输一次，具有最优的带宽效率。当$k = 2$时退化为二叉树，延迟最小；当$k = \sqrt{p}$时理论上达到最优权衡。

适用场景包括中大规模通信子($p > 8$)、需要调节延迟-带宽权衡的场景，以及层次化网络架构中`radix`可匹配网络拓扑的情况。该算法通过参数化设计在不同网络环境下具有良好的适应性，是 Open MPI 中重要的可调优广播算法实现。

#### 3.4.1.3. 二叉树广播算法

函数：`ompi_coll_base_bcast_intra_bintree`

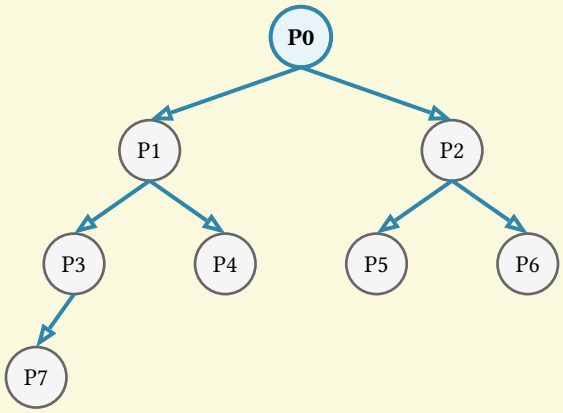
源码文件路径：ompi/mca/coll/base/coll_base_bcast.c

使用二叉树结构传播数据，每个节点向两个子节点传递数据。

```
int
ompi_coll_base_bcast_intra_bintree ( void* buffer,
                                     size_t count,
                                     struct ompi_datatype_t* datatype,
                                     int root,
                                     struct ompi_communicator_t* comm,
                                     mca_coll_base_module_t *module,
                                     uint32_t segsize )
{
    size_t segcount = count;
    size_t typelng;
    mca_coll_base_comm_t *data = module->base_data;

    COLL_BASE_UPDATE_BINTREE( comm, module, root );

    /**
     * Determine number of elements sent per operation.
     */
    ompi_datatype_type_size( datatype, &typelng );
    COLL_BASE_COMPUTED_SEGCOUNT( segsize, typelng, segcount );
```

```
21
22      OPAL_OUTPUT((ompi_coll_base_framework.framework_output,"......",
                     ompi_comm_rank(comm), segsize, (unsigned long)typelng,
23  segcount));
24
25      return ompi_coll_base_bcast_intra_generic( buffer, count, datatype, root,
    comm, module,
26                                                 segcount,
    data->cached_bintree );
27  }
```

代码 3.15　二叉树 Broadcast 算法核心代码




Binary Tree (8 进程)

图 3.4　二叉树 Broadcast 算法的树形结构

复杂度分析：二叉树广播算法的时间复杂度为$O(\log_2(p)\alpha + \beta m)$，延迟复杂度为$O(\log p)$，带宽复杂度为$O(m)$。相比线性算法，通信轮数从$O(p)$降低到$O(\log p)$，显著减少了延迟开销。该算法支持消息分段处理，通过`segsize`参数控制分段大小，在大消息传输时能够提高内存利用效率。

适用场景包括中等规模通信子($4 \leq p \leq 64$)、延迟敏感应用、中等大小消息(1KB – 1MB)，以及进程数为2的幂次时性能最优的情况。二叉树结构在延迟和实现复杂度之间达到良好平衡，是许多 MPI 实现中的默认选择，特别适合 CPU 密集型应用中的小到中等规模数据广播。

#### 3.4.1.4. 流水线广播算法（Pipeline Algorithm）

函数：`ompi_coll_base_bcast_intra_pipeline()`

源码文件路径：ompi/mca/coll/base/coll_base_bcast.c

其主要原理是：将大消息分割成多个小段（segments），在线性链结构上采用流水线方式传递数据，使不同数据段的传输可以重叠进行，提高带宽利用率。

```
1  int
2  ompi_coll_base_bcast_intra_pipeline( void* buffer,
3                                       size_t count,
4                                       struct ompi_datatype_t* datatype,
5                                       int root,
```

```
6                                       struct ompi_communicator_t* comm,
7                                       mca_coll_base_module_t *module,
8                                       uint32_t segsize )
9   {
10      size_t segcount = count;
11      size_t typelng;
12      mca_coll_base_comm_t *data = module->base_data;
13
14      COLL_BASE_UPDATE_PIPELINE( comm, module, root );
15
16      /**
17       * Determine number of elements sent per operation.
18       */
19      ompi_datatype_type_size( datatype, &typelng );
20      COLL_BASE_COMPUTED_SEGCOUNT( segsize, typelng, segcount );
21
22      OPAL_OUTPUT((ompi_coll_base_framework.framework_output,......
23
24      return ompi_coll_base_bcast_intra_generic( buffer, count, datatype, root,
    comm, module,
25                                                 segcount,
    data->cached_pipeline );
26  }
```

代码 3.16 流水线 Broadcast 算法核心代码

图示如下：

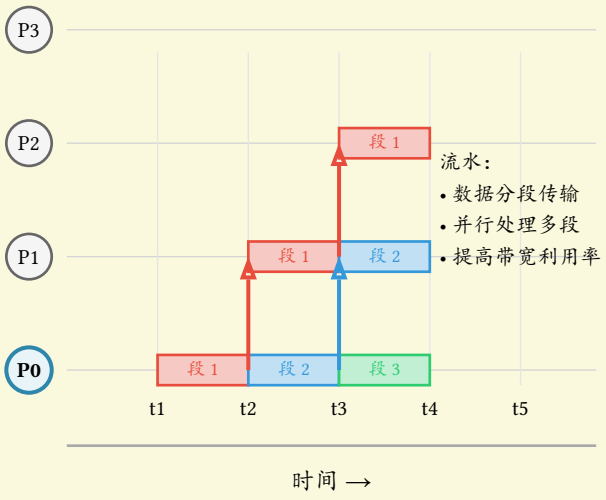



图 3.5 流水线 Broadcast 算法图示

算法复杂度分析：流水线广播算法的时间复杂度为$O((\log_2(p)+S-1)\alpha+\beta m)$，其中$S$为段数。通过消息分割和流水线重叠，延迟复杂度为$O(\log p+S)$，带宽复杂度保持$O(m)$但具有更好的带宽利用率。分段大小(`segsize`)直接影响性能：较小分段提供更好的重叠效果但增加通信开销，较大分段减少开销但降低重叠效益。

适用场景包括大消息广播($>$ 1MB)、带宽充足但延迟较高的网络环境、内存受限环境中分段可减少内存压力，以及需要通信-计算重叠的应用。该算法通过流水线技术有效隐藏通信延迟，在高性能计算中的大规模数据分发场景下表现优异，是带宽密集型应用的理想选择。

#### 3.4.1.5. 分散-聚集广播算法

函数：`ompi_coll_base_bcast_intra_scatter_allgather`

源码文件路径：ompi/mca/coll/base/coll_base_bcast.c

先使用二项树分散数据，再使用递归倍增方式聚集。

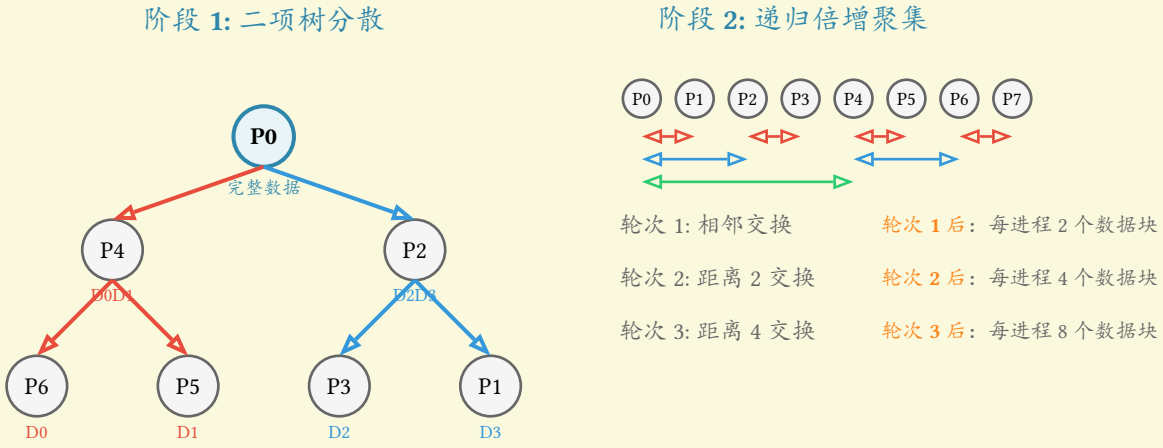
其主要原理是：采用两阶段策略，第一阶段使用二项树将数据分散到各进程（Scatter），第二阶段使用递归倍增算法进行全聚集（Allgather），重构完整数据。

```
/*
 * 限制条件: count >= comm_size
 */
int ompi_coll_base_bcast_intra_scatter_allgather(
    void *buf, size_t count, struct ompi_datatype_t *datatype, int root,
    struct ompi_communicator_t *comm, mca_coll_base_module_t *module,
    uint32_t segsize)
{
    int comm_size = ompi_comm_size(comm);
    int rank = ompi_comm_rank(comm);
    int vrank = (rank - root + comm_size) % comm_size;

    // 计算每个进程应分得的数据块大小
    size_t scatter_count = (count + comm_size - 1) / comm_size;

    /* 第一阶段: 二项树分散 */
    int mask = 0x1;
    while (mask < comm_size) {
        if (vrank & mask) {
            // 从父进程接收数据
            int parent = (rank - mask + comm_size) % comm_size;
            recv_count = rectify_diff(count, vrank * scatter_count);
            MCA_PML_CALL(recv((char *)buf + vrank * scatter_count * extent,
                              recv_count, datatype, parent,
                              MCA_COLL_BASE_TAG_BCAST, comm, &status));
            break;
        }
        mask <<= 1;
    }

    // 向子进程发送数据
    mask >>= 1;
    while (mask > 0) {
        if (vrank + mask < comm_size) {
            int child = (rank + mask) % comm_size;
            send_count = rectify_diff(curr_count, scatter_count * mask);
            MCA_PML_CALL(send((char *)buf + scatter_count * (vrank + mask)
* extent,
                              send_count, datatype, child,
                              MCA_COLL_BASE_TAG_BCAST,
                              MCA_PML_BASE_SEND_STANDARD, comm));
        }
        mask >>= 1;
    }
```

```
44
45      /* 第二阶段: 递归倍增全聚集 */
46      mask = 0x1;
47      while (mask < comm_size) {
48          int vremote = vrank ^ mask;
49          int remote = (vremote + root) % comm_size;
50
51          if (vremote < comm_size) {
52              // 与远程进程交换数据
53              ompi_coll_base_sendrecv((char *)buf + send_offset,
54                                      curr_count, datatype, remote,
55                                      MCA_COLL_BASE_TAG_BCAST,
56                                      (char *)buf + recv_offset,
57                                      recv_count, datatype, remote,
58                                      MCA_COLL_BASE_TAG_BCAST,
59                                      comm, &status, rank);
60              curr_count += recv_count;
61          }
62          mask <<= 1;
63      }
64
65      return MPI_SUCCESS;
66  }
```

代码 3.17 分散-聚集 Broadcast 算法核心代码

图示如下：




分散-聚集广播算法：8 进程，3 轮递归倍增聚集

图 3.6 分散-聚集 Broadcast 算法图示

算法复杂度分析：分散-聚集广播算法的时间复杂度为$O\left(2\log_2(p)\alpha + 2\beta\frac{m(p-1)}{p}\right)$，包含两个阶段：二项树分散阶段$O\left(\log_2(p)\alpha + \beta\frac{m(p-1)}{p}\right)$和递归倍增聚集阶段$O\left(\log_2(p)\alpha + \beta\frac{m(p-1)}{p}\right)$。总延迟复杂度为$O(\log p)$，总带宽复杂度为$O\left(\frac{m(p-1)}{p}\right)$，当$p$较大时接近$O(m)$的最优带宽效率。该算法要求count $\geq$ comm_size，当消息过小时会回退到线性算法。

适用场景包括大消息广播(count $\geq$ comm_size)、大规模通信子($p > 64$)、高带宽网络环境，以及需要避免根节点瓶颈的场景。通过两阶段设计，该算法充分利用聚合带宽并避免单点瓶颈，在大

消息和大规模场景下具有近似线性的带宽效率，是高性能计算中处理大规模数据广播的重要算法选择。

#### 3.4.1.6. 其它 Broadcast 算法

在源码 ompi/mca/coll/base/coll_base_bcast.c 中，除了上述详细介绍的算法外，还实现了以下其它 Broadcast 算法：

- 链式广播算法（ompi_coll_base_bcast_intra_chain）
  形成一个或多个通信链，数据沿链传递。支持通过 fanout 参数控制多链并行，适合特定网络拓扑结构。
- 分裂二进制树算法（ompi_coll_base_bcast_intra_split_bintree）
  将树结构和数据进行分割以优化传输效率，通过更复杂的调度在某些场景下实现更高的性能。
- 分散-环形聚集算法（ompi_coll_base_bcast_intra_scatter_allgather_ring）
  结合二项树分散和环形聚集的混合策略，先使用二项树分散数据，再使用环形算法进行聚集，在特定网络拓扑上更高效。
- 通用树形算法（ompi_coll_base_bcast_intra_generic）
  提供通用的树形广播框架，可以配合不同的树结构（二叉树、k 进制树等）实现灵活的广播策略。

这些算法的设计目标是适应不同的通信规模、消息大小和网络特性。Open MPI 的动态选择机制会根据运行时条件（进程数量、消息大小、网络延迟等）自动选择最优的算法实现，为用户提供透明的性能优化。

#### 3.4.1.7. 总结

基于上述对`MPI_Bcast`的算法的讨论，整理得如下表格：

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
| --- | --- | --- | --- | --- |
| 线性算法 | `ompi_coll_base_bcast_intra_basic_linear` | 无 | $O(N\alpha + N\beta m)$ | 小规模通信子或回退选择 |
| 二叉树算法 | `ompi_coll_base_bcast_intra_bintree` | `segsize` | $O(\log_2(p)\alpha + \beta m)$ | 中等规模通信子延迟敏感应用 |
| 二项式树算法 | `ompi_coll_base_bcast_intra_binomial` | `segsize` | $O(\log_2(p)\alpha + \beta m)$ | 中等规模通信子支持消息分段 |
| K 项树算法 | `ompi_coll_base_bcast_intra_knomial` | `segsize` `radix` | $O\left(\log_{k(p)}\alpha + \beta m\right)$ | 可调节延迟-带宽权衡的中大规模通信 |
| 流水线算法 | `ompi_coll_base_bcast_intra_pipeline` | `segsize` | $O((\log_2(p) + S)\alpha + \beta m)$ | 大消息广播通信-计算重叠 |

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
|---|---|---|---|---|
| 链式算法 | `ompi_coll_base_bcast_intra_chain` | `segsize` `chains` | $O(\frac{N}{\text{chains}} \cdot \alpha + \beta m)$ | 特定网络拓扑<br>多链并行传输 |
| 分散-聚集算法 | `ompi_coll_base_bcast_intra_scatter_allgather` | `segsize` | $O(\alpha \log p + \beta m)$ | 大消息广播<br>避免根节点瓶颈 |
| 分散-环形聚集算法 | `ompi_coll_base_bcast_intra_scatter_allgather_ring` | `segsize` | $O(\alpha(\log(p) + p) + \beta m)$ | 超大规模通信子<br>带宽受限网络 |
| 分裂二叉树算法 | `ompi_coll_base_bcast_intra_split_bintree` | `segsize` | $O(\log_2(p)\alpha + \beta m)$ | 数据和树结构<br>分割优化场景 |
| 通用树形算法 | `ompi_coll_base_bcast_intra_generic` | `tree` `segcount` | 取决于树结构 | 通用框架<br>配合不同树结构 |

表 3.1：Open MPI Broadcast 算法总结

参数说明：

- $S$: 流水线算法中的段数
- $\alpha$: 通信延迟参数，$\beta$: 带宽倒数参数
- $m$: 消息大小，$p$: 进程数量
- `segsize`：控制消息分段大小的参数
- `radix`：K 项树的分支因子（$\geq 2$）
- `chains`：链式算法中并行链的数量
- `tree`：指定使用的树结构类型
- `segcount`：每段传输的元素数量

### 3.4.2. Scatter

`Scatter`的函数原型如下：

```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_type,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_type,
    int root,
    MPI_Comm communicator)
```

其中：`send_data`参数是只在根进程上有效的待分发数据数组。`recv_data`是所有进程接收数据的缓冲区。`send_count`和`recv_count`分别指定发送和接收的数据元素数量。

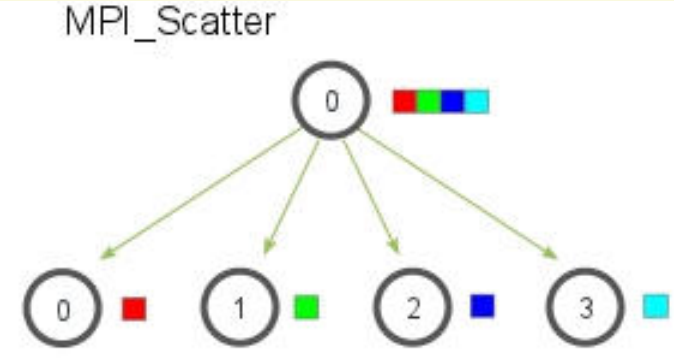



图 3.7 MPI_Scatter 通信模式图示

#### 3.4.2.1. 二项式树算法（Binomial Tree Algorithm）

函数：`ompi_coll_base_scatter_intra_binomial()`

源码文件路径：ompi/mca/coll/base/coll_base_scatter.c

其主要原理是：使用二项式树结构递归分发数据，根进程逐层向下传递数据块，每个内部节点收到数据后保留自己的部分，并将剩余数据继续向子节点分发。

```
int ompi_coll_base_scatter_intra_binomial(
    const void *sbuf, size_t scount, struct ompi_datatype_t *sdtype,
    void *rbuf, size_t rcount, struct ompi_datatype_t *rdtype,
    int root, struct ompi_communicator_t *comm,
    mca_coll_base_module_t *module)
{
    // 创建二项式树
    COLL_BASE_UPDATE_IN_ORDER_BMTREE(comm, base_module, root);
    ompi_coll_tree_t *bmtree = data->cached_in_order_bmtree;

    vrank = (rank - root + size) % size;

    if (vrank % 2) {  // 叶节点
        // 从父进程接收数据
        err = MCA_PML_CALL(recv(rbuf, rcount, rdtype, bmtree->tree_prev,
                                MCA_COLL_BASE_TAG_SCATTER, comm, &status));
        return MPI_SUCCESS;
    }

    // 根进程和内部节点处理数据
    if (rank == root) {
        curr_count = scount * size;
        // 数据重排序以适应分发顺序
        if (0 != root) {
            // 对非0根进程进行数据旋转
            opal_convertor_pack(&convertor, iov, &iov_size, &max_data);
        }
    } else {
        // 非根内部节点：从父进程接收数据
        err = MCA_PML_CALL(recv(ptmp, packed_size, MPI_PACKED, bmtree->tree_prev,
                                MCA_COLL_BASE_TAG_SCATTER, comm, &status));
        curr_count = status._ucount;
```

```
    }

    // 本地复制自己需要的数据
    if (rbuf != MPI_IN_PLACE) {
        err = ompi_datatype_sndrcv(ptmp, scount, sdtype,
                                   rbuf, rcount, rdtype);
    }

    // 向子节点发送数据
    for (int i = bmtree->tree_nextsize - 1; i >= 0; i--) {
        int vchild = (bmtree->tree_next[i] - root + size) % size;
        int send_count = vchild - vrank;
        if (send_count > size - vchild)
            send_count = size - vchild;
        send_count *= scount;

        err = MCA_PML_CALL(send(ptmp + (curr_count - send_count) * sextent,
                                send_count, sdtype, bmtree->tree_next[i],
                                MCA_COLL_BASE_TAG_SCATTER,
                                MCA_PML_BASE_SEND_STANDARD, comm));
        curr_count -= send_count;
    }

    return MPI_SUCCESS;
}
```

代码 3.18 二项式树 Scatter 算法核心代码

算法复杂度分析：二项式树散射算法的时间复杂度为$O\left(\alpha\log(p)+\beta\frac{m(p-1)}{p}\right)$，其中$m=$ scount $\times$ comm_size为总数据量。延迟复杂度为$O(\log p)$，相比线性算法的$O(p)$有显著改善；带宽复杂度为$O\left(\frac{m(p-1)}{p}\right)$，当进程数较大时接近$O(m)$的最优效率。算法内存需求因角色而异：根进程需要scount comm_size $\times$ sdtype_size内存，非根非叶进程需要rcount $\times$ comm_size $\times$ rdtype_size内存。

适用场景包括大规模通信子($p>8$)、大消息分发、延迟敏感应用，以及需要避免根进程成为瓶颈的场景。该算法通过树形结构有效分担根进程负载，在通信轮数和带宽利用率之间达到良好平衡，特别适合高性能计算中的大规模数据分发操作。

#### 3.4.2.2. 线性算法（Linear Algorithm）

函数：`ompi_coll_base_scatter_intra_basic_linear()`

源码文件路径：ompi/mca/coll/base/coll_base_scatter.c

其主要原理是：根进程顺序向每个进程发送对应的数据块，所有非根进程直接从根进程接收数据。

```
int ompi_coll_base_scatter_intra_basic_linear(
    const void *sbuf, size_t scount, struct ompi_datatype_t *sdtype,
    void *rbuf, size_t rcount, struct ompi_datatype_t *rdtype,
    int root, struct ompi_communicator_t *comm,
    mca_coll_base_module_t *module)
{
```

```
    int i, rank, size, err;
    ptrdiff_t incr;
    char *ptmp;

    rank = ompi_comm_rank(comm);
    size = ompi_comm_size(comm);

    // 非根进程：接收数据
    if (rank != root) {
        err = MCA_PML_CALL(recv(rbuf, rcount, rdtype, root,
                                MCA_COLL_BASE_TAG_SCATTER,
                                comm, MPI_STATUS_IGNORE));
        return err;
    }

    // 根进程：循环发送数据
    err = ompi_datatype_type_extent(sdtype, &incr);
    incr *= scount;

    for (i = 0, ptmp = (char *)sbuf; i < size; ++i, ptmp += incr) {
        if (i == rank) {
            // 简单优化：根进程本地复制
            if (MPI_IN_PLACE != rbuf) {
                err = ompi_datatype_sndrcv(ptmp, scount, sdtype,
                                           rbuf, rcount, rdtype);
            }
        } else {
            // 向其他进程发送数据
            err = MCA_PML_CALL(send(ptmp, scount, sdtype, i,
                                    MCA_COLL_BASE_TAG_SCATTER,
                                    MCA_PML_BASE_SEND_STANDARD, comm));
        }
        if (MPI_SUCCESS != err) return err;
    }

    return MPI_SUCCESS;
}
```

代码 3.19　线性 Scatter 算法核心代码

算法复杂度分析：线性散射算法的时间复杂度为$O((p-1)\alpha+(p-1)\beta m')$，其中$m'=\text{scount}$为单个数据块大小。延迟复杂度为$O(p)$，根进程需要进行$p-1$次串行发送操作；带宽复杂度为$O(pm')$，总传输量为所有数据块之和。该算法实现最为简单[7]，无需构建树形拓扑，空间复杂度为$O(1)$。

[7]根据源码注释，线性算法被从 BASIC 组件复制到 BASE 组件中，主要原因是：

1. 算法简单且不进行消息分段
2. 对于小规模节点或小数据量，性能与复杂的树形分段算法相当
3. 可被决策函数选择作为特定场景的最优选择
4. V1 版本的模块选择机制要求代码复制，V2 版本将采用不同方式处理

适用场景包括小规模通信子($p \leq 4$)、小消息分发、网络连接性差的环境，以及作为复杂算法的回退选择。该算法的主要优势是实现简单、无拓扑构建开销，但在大规模场景下根进程会成为严重瓶颈，扩展性较差。

#### 3.4.2.3. 非阻塞线性算法（Linear Non-blocking Algorithm）

函数：`ompi_coll_base_scatter_intra_linear_nb()`

源码文件路径：ompi/mca/coll/base/coll_base_scatter.c

其主要原理是：使用非阻塞发送(`isend`)分发数据，并通过周期性的阻塞发送来刷新本地资源，确保通信进展的同时避免资源耗尽。

```
1  int ompi_coll_base_scatter_intra_linear_nb(
2      const void *sbuf, size_t scount, struct ompi_datatype_t *sdtype,
3      void *rbuf, size_t rcount, struct ompi_datatype_t *rdtype,
4      int root, struct ompi_communicator_t *comm,
5      mca_coll_base_module_t *module, int max_reqs)
6  {
7      int i, rank, size, err, nreqs;
8      ompi_request_t **reqs = NULL, **preq;
9
10     rank = ompi_comm_rank(comm);
11     size = ompi_comm_size(comm);
12
13     // 非根进程：接收数据
14     if (rank != root) {
15         err = MCA_PML_CALL(recv(rbuf, rcount, rdtype, root,
16                                 MCA_COLL_BASE_TAG_SCATTER,
17                                 comm, MPI_STATUS_IGNORE));
18         return MPI_SUCCESS;
19     }
20
21     // 计算请求数量和分配请求数组
22     if (max_reqs <= 1) {
23         nreqs = size - 1;  // 除自己外的所有进程
24     } else {
25         // 周期性使用阻塞发送，减少请求数量
26         nreqs = size - (size / max_reqs);
27     }
28
29     reqs = ompi_coll_base_comm_get_reqs(module->base_data, nreqs);
30
31     // 根进程：循环发送数据
32     for (i = 0, ptmp = (char *)sbuf, preq = reqs; i < size; ++i, ptmp +=
   incr) {
33         if (i == rank) {
34             // 本地复制
35             if (MPI_IN_PLACE != rbuf) {
36                 err = ompi_datatype_sndrcv(ptmp, scount, sdtype,
37                                            rbuf, rcount, rdtype);
38             }
39         } else {
40             if (!max_reqs || (i % max_reqs)) {
```

```
41                  // 使用非阻塞发送
42                  err = MCA_PML_CALL(isend(ptmp, scount, sdtype, i,
43                                           MCA_COLL_BASE_TAG_SCATTER,
44                                           MCA_PML_BASE_SEND_STANDARD,
45                                           comm, preq++));
46              } else {
47                  // 周期性使用阻塞发送作为资源刷新
48                  err = MCA_PML_CALL(send(ptmp, scount, sdtype, i,
49                                          MCA_COLL_BASE_TAG_SCATTER,
50                                          MCA_PML_BASE_SEND_STANDARD, comm));
51              }
52          }
53          if (MPI_SUCCESS != err) goto err_hndl;
54      }
55
56      // 等待所有非阻塞发送完成
57      err = ompi_request_wait_all(preq - reqs, reqs, MPI_STATUSES_IGNORE);
58
59      return MPI_SUCCESS;
60  }
```

代码 3.20　非阻塞线性 Scatter 算法核心代码

算法复杂度分析：非阻塞线性散射算法的时间复杂度与标准线性算法相同，为$O((p-1)\alpha + (p-1)\beta m')$，但通过非阻塞通信获得更好的重叠效果。延迟复杂度理论上仍为$O(p)$，但实际延迟因通信重叠而降低；带宽复杂度为$O(pm')$。该算法通过`max_reqs`参数[8]控制资源使用，在内存需求和性能之间提供可调节的权衡。

适用场景包括中等规模通信子、需要通信-计算重叠的应用、内存资源有限但希望改善性能的环境，以及网络具有良好并发处理能力的场景。该算法在保持线性算法简单性的同时，通过非阻塞技术提升了性能，是资源受限环境下的良好选择。

#### 3.4.2.4. 其它 Scatter 算法

除了上述实现的算法外，base 组件中的的 Scatter 操作中还还包含采用以下算法和优化策略：

- 链式散射算法（ompi_coll_base_scatter_intra_chain）
  形成一个或多个通信链，数据沿链传递，适合特定网络拓扑结构。
- 分段流水线算法（ompi_coll_base_scatter_intra_pipeline）
  将大消息分段，采用流水线方式在链式或树形结构上传递，提升大消息分发效率。
- 通用树形算法（ompi_coll_base_scatter_intra_generic）
  提供通用的树形分发框架，可配合不同树结构（如二叉树、k 进制树等）实现灵活策略。

[8] `max_reqs` 参数的作用机制：
1. 当 `max_reqs ≤ 1` 时，所有发送都使用非阻塞方式
2. 当 `max_reqs > 1` 时，每 `max_reqs` 个发送操作中插入一次阻塞发送
3. 阻塞发送起到“本地资源刷新”的作用，确保通信进展并避免缓冲区溢出
4. 这种混合策略在性能和资源利用之间达到平衡

- 基于网络拓扑的优化算法（如 ompi_coll_base_scatter_intra_topo）
  根据具体网络拓扑（如胖树、环面等）优化数据分发路径，减少拥塞。
- 混合算法策略
  根据消息大小和进程数量动态选择算法，小消息用线性，大消息用树形或链式算法。

#### 3.4.2.5. 总结

基于上述对`MPI_Scatter`的算法的讨论，整理得如下表格：

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
|---|---|---|---|---|
| 二项式树算法 | `ompi_coll_base_scatter_intra_binomial` | 无 | $O\left(\alpha \log(p) + \beta \frac{m(p-1)}{p}\right)$ | 大规模通信子<br>大消息分发 |
| 线性算法 | `ompi_coll_base_scatter_intra_basic_linear` | 无 | $O((p-1)\alpha + (p-1)\beta m')$ | 小规模通信子<br>或回退选择 |
| 非阻塞线性算法 | `ompi_coll_base_scatter_intra_linear_nb` | `max_reqs` | $O((p-1)\alpha + (p-1)\beta m')$ | 中等规模通信子<br>通信-计算重叠 |
| 链式散射算法 | `ompi_coll_base_scatter_intra_chain` | `segsize`<br>`chains` | $O\left(\frac{p}{\text{chains}} \cdot \alpha + \beta m\right)$ | 特定网络拓扑<br>多链并行传输 |
| 分段流水线算法 | `ompi_coll_base_scatter_intra_pipeline` | `segsize` | $O((\log_2(p) + S)\alpha + \beta m)$ | 大消息分发<br>流水线重叠 |
| 通用树形算法 | `ompi_coll_base_scatter_intra_generic` | `tree`<br>`segcount` | 取决于树结构 | 通用框架<br>配合不同树结构 |
| 基于拓扑的算法 | `ompi_coll_base_scatter_intra_topo` | `topology` | 取决于网络拓扑 | 特定网络架构<br>拓扑感知优化 |

表 3.2：Open MPI Scatter 算法总结

参数说明：

- $S$: 流水线算法中的段数
- $\alpha$: 通信延迟参数，$\beta$: 带宽倒数参数
- $m$: 总消息大小（scount × comm_size），
  $m'$: 单个数据块大小（scount）
- $p$: 进程数量
- `max_reqs`: 控制非阻塞发送请求数量
- `chains`: 链式算法中并行链的数量
- `tree`: 指定使用的树结构类型
- `segcount`: 每段传输的元素数量
- `segsize`: 控制消息分段大小的参数
- `topology`: 网络拓扑结构参数

### 3.4.3. Gather

`Gather`的函数原型如下：

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_type,
```

```
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_type,
    int root,
    MPI_Comm communicator)
```

其中：`send_data`是每个进程要发送的数据，`recv_data`是根进程接收所有数据的缓冲区（仅在根进程有效）。`send_count`和`recv_count`分别指定每个进程发送和根进程从每个进程接收的数据元素数量。

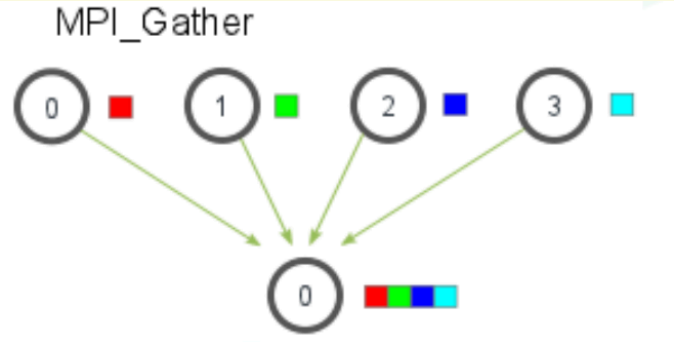



图 3.8 MPI_Gather 通信模式图示

Open MPI 为 Gather 操作提供了多种算法实现：

#### 3.4.3.1. 二项式树算法（Binomial Tree Algorithm）

函数：`ompi_coll_base_gather_intra_binomial()`

源码文件路径：ompi/mca/coll/base/coll_base_gather.c

其主要原理是:使用二项式树结构从叶节点向根节点收集数据,每个内部节点先从子节点收集数据，然后将收集到的数据连同自己的数据一起发送给父节点，最终所有数据汇聚到根进程。

```
int
ompi_coll_base_gather_intra_binomial(const void *sbuf, size_t scount,
                                     struct ompi_datatype_t *sdtype,
                                     void *rbuf, size_t rcount,
                                     struct ompi_datatype_t *rdtype,
                                     int root,
                                     struct ompi_communicator_t *comm,
                                     mca_coll_base_module_t *module)
{
    // 创建二项式树
    COLL_BASE_UPDATE_IN_ORDER_BMTREE(comm, base_module, root);
    bmtree = data->cached_in_order_bmtree;

    vrank = (rank - root + size) % size;

    if (rank == root) {
        // 根进程：分配接收缓冲区
        if (0 == root) {
            ptmp = (char *) rbuf;
```

```
20              if (sbuf != MPI_IN_PLACE) {
21                  err = ompi_datatype_sndrcv((void *)sbuf, scount, sdtype,
22                                             ptmp, rcount, rdtype);
23              }
24          } else {
25              // 非0根进程需要额外缓冲区，最后进行数据旋转
26              tempbuf = (char *) malloc(rsize);
27              ptmp = tempbuf - rgap;
28              if (sbuf != MPI_IN_PLACE) {
29                  err = ompi_datatype_sndrcv((void *)sbuf, scount, sdtype,
30                                             ptmp, rcount, rdtype);
31              }
32          }
33          total_recv = rcount;
34      } else if (!(vrank % 2)) {
35          // 内部节点：分配临时缓冲区用于收集子节点数据
36          tempbuf = (char *) malloc(ssize);
37          ptmp = tempbuf - sgap;
38          // 复制本地数据到临时缓冲区
39          err = ompi_datatype_sndrcv((void *)sbuf, scount, sdtype,
40                                     ptmp, scount, sdtype);
41          total_recv = rcount;
42      } else {
43          // 叶节点：直接使用发送缓冲区
44          ptmp = (char *) sbuf;
45          total_recv = scount;
46      }
47
48      if (!(vrank % 2)) {
49          // 所有非叶节点从子节点接收数据
50          for (i = 0; i < bmtree->tree_nextsize; i++) {
51              int mycount = 0, vkid;
52              vkid = (bmtree->tree_next[i] - root + size) % size;
53              mycount = vkid - vrank;
54              if (mycount > (size - vkid))
55                  mycount = size - vkid;
56              mycount *= rcount;
57
58              err = MCA_PML_CALL(recv(ptmp + total_recv*rextent,
                                        (ptrdiff_t)rcount * size -
59  total_recv, rdtype,
                                        bmtree->tree_next[i],
60  MCA_COLL_BASE_TAG_GATHER,
61                                      comm, &status));
62              total_recv += mycount;
63          }
64      }
65
66      if (rank != root) {
67          // 所有非根节点向父节点发送数据
68          err = MCA_PML_CALL(send(ptmp, total_recv, sdtype,
69                                  bmtree->tree_prev,
70                                  MCA_COLL_BASE_TAG_GATHER,
71                                  MCA_PML_BASE_SEND_STANDARD, comm));
```

```
72      }
73
74      if (rank == root && root != 0) {
75          // 非0根进程需要进行数据旋转
76          err = ompi_datatype_copy_content_same_ddt(rdtype,
77                                    (ptrdiff_t)rcount * (ptrdiff_t)(size - root),
                                      (char *)rbuf + rextent * (ptrdiff_t)root
78  * (ptrdiff_t)rcount,
79                                    ptmp);
80
81          err = ompi_datatype_copy_content_same_ddt(rdtype,
82                                    (ptrdiff_t)rcount * (ptrdiff_t)root,
83                                    (char *) rbuf,
                                      ptmp + rextent * (ptrdiff_t)rcount *
84  (ptrdiff_t)(size-root));
85          free(tempbuf);
86      }
87
88      return MPI_SUCCESS;
89  }
```

代码 3.21　二项式树 Gather 算法核心代码

算法复杂度分析：二项式树聚集算法的时间复杂度为$O(\alpha \log(p) + \beta m'(p-1))$，其中$m' =$ scount为单个进程的数据大小。延迟复杂度为$O(\log p)$，相比线性算法的$O(p)$有显著改善；带宽复杂度为$O(m'p)$，总传输量为所有进程数据之和。该算法内存需求[9]因节点角色而异，通过树形结构有效减少了通信轮数。

适用场景包括大规模通信子($p > 8$)、大消息收集、延迟敏感应用，以及需要减少通信轮数的场景。该算法通过二项式树结构在通信轮数和实现复杂度之间达到良好平衡，特别适合高性能计算中需要高效数据收集的应用场景。

#### 3.4.3.2. 线性同步算法（Linear Sync Algorithm）

函数：`ompi_coll_base_gather_intra_linear_sync()`

源码文件路径：ompi/mca/coll/base/coll_base_gather.c

其主要原理是：增加同步机制的线性收集算法，根进程首先向非根进程发送零字节同步消息，然后非根进程分两阶段发送数据：先发送第一段数据（同步），再发送剩余数据，确保数据传输的有序性和可靠性。

```
1  int
2  ompi_coll_base_gather_intra_linear_sync(const void *sbuf, size_t scount,
3                                          struct ompi_datatype_t *sdtype,
```

[9]二项式树 gather 的内存需求分析：

1. 根进程：需要rcount × comm_size × rdtype_size内存存储最终结果
2. 内部节点：需要scount × comm_size × sdtype_size内存作为临时缓冲区
3. 叶节点：仅需要自身数据大小，无额外内存开销
4. 非 0 根进程：额外需要临时缓冲区用于数据旋转操作

```
4                                           void *rbuf, size_t rcount,
5                                           struct ompi_datatype_t *rdtype,
6                                           int root,
7                                           struct ompi_communicator_t *comm,
8                                           mca_coll_base_module_t *module,
9                                           int first_segment_size)
10  {
11      if (rank != root) {
12          // 非根进程：三步骤通信
13          // 1. 接收根进程的零字节同步消息
14          ret = MCA_PML_CALL(recv(rbuf, 0, MPI_BYTE, root,
15                                  MCA_COLL_BASE_TAG_GATHER,
16                                  comm, MPI_STATUS_IGNORE));
17
18          // 2. 同步发送第一段数据
19          ompi_datatype_type_size(sdtype, &typelng);
20          ompi_datatype_get_extent(sdtype, &lb, &extent);
21          first_segment_count = scount;
22          COLL_BASE_COMPUTED_SEGCOUNT((size_t)first_segment_size, typelng,
23                                      first_segment_count);
24
25          ret = MCA_PML_CALL(send(sbuf, first_segment_count, sdtype, root,
26                                  MCA_COLL_BASE_TAG_GATHER,
27                                  MCA_PML_BASE_SEND_STANDARD, comm));
28
29          // 3. 发送剩余数据
30          ret = MCA_PML_CALL(send((char*)sbuf + extent * first_segment_count,
31                                  (scount - first_segment_count), sdtype,
32                                  root, MCA_COLL_BASE_TAG_GATHER,
33                                  MCA_PML_BASE_SEND_STANDARD, comm));
34
35      } else {
36          // 根进程：与每个非根进程进行复杂的同步通信
37          for (i = 0; i < size; ++i) {
38              if (i == rank) continue;
39
40              // 1. 发布第一段数据的非阻塞接收
41              ptmp = (char*)rbuf + (ptrdiff_t)i * (ptrdiff_t)rcount * extent;
42              ret = MCA_PML_CALL(irecv(ptmp, first_segment_count, rdtype, i,
43                                       MCA_COLL_BASE_TAG_GATHER, comm,
44                                       &first_segment_req));
45
46              // 2. 发送零字节同步消息
47              ret = MCA_PML_CALL(send(rbuf, 0, MPI_BYTE, i,
48                                      MCA_COLL_BASE_TAG_GATHER,
49                                      MCA_PML_BASE_SEND_STANDARD, comm));
50
51              // 3. 发布第二段数据的非阻塞接收
52              ptmp = (char*)rbuf + ((ptrdiff_t)i * (ptrdiff_t)rcount +
    first_segment_count) * extent;
53              ret = MCA_PML_CALL(irecv(ptmp, (rcount - first_segment_count),
54                                       rdtype, i, MCA_COLL_BASE_TAG_GATHER,
    comm,
55                                       &reqs[i]));
```

```
56
57              // 4. 等待第一段数据完成
58              ret = ompi_request_wait(&first_segment_req, MPI_STATUS_IGNORE);
59          }
60
61          // 复制本地数据
62          if (MPI_IN_PLACE != sbuf) {
63              ret = ompi_datatype_sndrcv((void *)sbuf, scount, sdtype,
                                           (char*)rbuf + (ptrdiff_t)rank *
64  (ptrdiff_t)rcount * extent,
65                                         rcount, rdtype);
66          }
67
68          // 等待所有第二段数据完成
69          ret = ompi_request_wait_all(size, reqs, MPI_STATUSES_IGNORE);
70      }
71
72      return MPI_SUCCESS;
73  }
```

代码 3.22 线性同步 Gather 算法核心代码

算法复杂度分析：线性同步聚集算法的时间复杂度为$O(2(p-1)\alpha+(p-1)\beta m')$，其中同步机制引入额外的延迟开销。延迟复杂度为$O(p)$，包含同步消息的往返时间；带宽复杂度为$O(m'p)$，数据传输量与标准线性算法相同。该算法通过`first_segment_size`参数[10]控制同步粒度，在可靠性和性能之间提供可调节的权衡。

适用场景包括需要严格数据顺序的应用、不可靠网络环境、需要错误恢复能力的系统，以及对数据完整性要求极高的场景。该算法通过同步机制确保数据传输的有序性和可靠性，虽然增加了通信开销，但在关键应用中提供了重要的可靠性保证。

#### 3.4.3.3. 线性算法（Linear Algorithm）

函数：`ompi_coll_base_gather_intra_basic_linear()`

源码文件路径：ompi/mca/coll/base/coll_base_gather.c

其主要原理是：所有非根进程依次向根进程发送数据，根进程循环接收来自各进程的数据，按进程号顺序存储到接收缓冲区中。

```
1  int
2  ompi_coll_base_gather_intra_basic_linear(const void *sbuf, size_t scount,
3                                           struct ompi_datatype_t *sdtype,
4                                           void *rbuf, size_t rcount,
5                                           struct ompi_datatype_t *rdtype,
```

[10] `first_segment_size`参数的作用机制：
1. 控制第一段数据的大小，影响同步粒度
2. 较小的段大小提供更细粒度的同步控制
3. 较大的段大小减少通信轮数但降低同步效果
4. 通过`COLL_BASE_COMPUTED_SEGCOUNT`宏根据数据类型大小计算实际段数量

```
                                          int root,
                                          struct ompi_communicator_t *comm,
                                          mca_coll_base_module_t *module)
{
    int i, err, rank, size;
    char *ptmp;
    MPI_Aint incr, extent, lb;

    size = ompi_comm_size(comm);
    rank = ompi_comm_rank(comm);

    // 非根进程：发送数据并返回
    if (rank != root) {
        return MCA_PML_CALL(send(sbuf, scount, sdtype, root,
                                 MCA_COLL_BASE_TAG_GATHER,
                                 MCA_PML_BASE_SEND_STANDARD, comm));
    }

    // 根进程：循环接收数据
    ompi_datatype_get_extent(rdtype, &lb, &extent);
    incr = extent * (ptrdiff_t)rcount;

    for (i = 0, ptmp = (char *) rbuf; i < size; ++i, ptmp += incr) {
        if (i == rank) {
            // 处理本地数据
            if (MPI_IN_PLACE != sbuf) {
                err = ompi_datatype_sndrcv((void *)sbuf, scount, sdtype,
                                           ptmp, rcount, rdtype);
            } else {
                err = MPI_SUCCESS;
            }
        } else {
            // 从其他进程接收数据
            err = MCA_PML_CALL(recv(ptmp, rcount, rdtype, i,
                                    MCA_COLL_BASE_TAG_GATHER,
                                    comm, MPI_STATUS_IGNORE));
        }
        if (MPI_SUCCESS != err) return err;
    }

    return MPI_SUCCESS;
}
```

代码 3.23 线性 Gather 算法核心代码

算法复杂度分析：线性聚集算法的时间复杂度为$O((p-1)\alpha+(p-1)\beta m')$，其中$m'=\text{scount}$为单个进程的数据大小。延迟复杂度为$O(p)$，根进程需要进行$p-1$次串行接收操作；带宽复杂度为$O(m'p)$，总传输量为所有进程数据之和。空间复杂度为$O(1)$。

适用场景包括小规模通信子($p \leq 4$)、小消息收集、网络连接性差的环境，以及作为复杂算法的回退选择。该算法的主要优势是实现简单、无拓扑构建开销，但在大规模场景下根进程会成为严重瓶颈，扩展性较差。

#### 3.4.3.4. 其它 Gather 算法

除了上述实现的算法外，源码注释中提到了以下待实现或优化的 Gather 算法和优化策略：

- 通用树形算法（`ompi_coll_base_gather_intra_generic`）
  提供通用的树形收集框架，可配合不同树结构（如二叉树、k 进制树等）实现灵活的收集策略。
- 二进制树算法（`ompi_coll_base_gather_intra_binary`）
  使用完全二叉树结构进行数据收集，在某些场景下可能比二项式树有更好的负载平衡。
- 链式收集算法（`ompi_coll_base_gather_intra_chain`）
  形成一个或多个通信链，数据沿链向根进程汇聚，适合特定网络拓扑结构。
- 流水线收集算法（`ompi_coll_base_gather_intra_pipeline`）
  将大消息分段，采用流水线方式在树形或链式结构上收集数据，提升大消息处理效率。
- 消息分段优化
  对于超大消息，采用分段传输策略，结合流水线技术实现更好的内存利用和通信重叠。
- 基于网络拓扑的优化算法
  根据具体网络拓扑（如胖树、环面等）优化数据收集路径，减少网络拥塞并提高带宽利用率。

#### 3.4.3.5. 总结

基于上述对`MPI_Gather`的算法[11]的讨论，整理得如下表格：

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
|---|---|---|---|---|
| 二项式树算法 | `ompi_coll_base_gather_intra_binomial` | 无 | $O(\alpha \log(p) + \beta m'(p-1))$ | 大规模通信子<br>大消息收集 |
| 线性同步算法 | `ompi_coll_base_gather_intra_linear_sync` | `first_segment_size` | $O(2(p-1)\alpha + (p-1)\beta m')$ | 可靠性要求高<br>不可靠网络环境 |
| 线性算法 | `ompi_coll_base_gather_intra_basic_linear` | 无 | $O((p-1)\alpha + (p-1)\beta m')$ | 小规模通信子<br>或回退选择 |
| 通用树形算法 | `ompi_coll_base_gather_intra_generic` | `tree`<br>`segcount` | 取决于树结构 | 通用框架<br>配合不同树结构 |
| 二进制树算法 | `ompi_coll_base_gather_intra_binary` | `segsize` | $O(\log_2(p)\alpha + \beta m')$ | 完全二叉树结构<br>负载平衡优化 |
| 链式收集算法 | `ompi_coll_base_gather_intra_chain` | `segsize`<br>`chains` | $O(\frac{p}{\text{chains}} \cdot \alpha + \beta m')$ | 特定网络拓扑<br>多链并行收集 |
| 流水线收集算法 | `ompi_coll_base_gather_intra_pipeline` | `segsize` | $O((\log_2(p) + S)\alpha + \beta m')$ | 大消息收集<br>流水线重叠 |

表 3.3：Open MPI Gather 算法总结

[11] 源码注释中指出其中部分代码仍待开发：

*Todo: gather_intra_generic, gather_intra_binary, gather_intra_chain, gather_intra_pipeline, segmentation?*

参数说明：

- $S$: 流水线算法中的段数
- $\alpha$: 通信延迟参数，$\beta$: 带宽倒数参数
- $m'$: 单个进程数据大小（scount），$p$: 进程数量
- `first_segment_size`: 控制同步算法第一段数据大小
- `chains`: 链式算法中并行链的数量
- `tree`: 指定使用的树结构类型
- `segcount`: 每段传输的元素数量
- `segsize`: 控制消息分段大小的参数

#### 3.4.4. Allgather

`Allgather`的函数原型如下：

```
MPI_Allgather(
    void* send_data,
    int send_count,
    MPI_Datatype send_type,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_type,
    MPI_Comm communicator)
```

其中：`send_data`是每个进程要发送的数据，`recv_data`是所有进程接收所有数据的缓冲区。与 Gather 不同，Allgather 中每个进程都能获得完整的聚集结果，无需指定根进程。`send_count`和`recv_count`分别指定每个进程发送和从每个进程接收的数据元素数量。

Open MPI 为 Allgather 操作提供了丰富的算法实现：

##### 3.4.4.1. 递归加倍算法（Recursive Doubling Algorithm）

函数：`ompi_coll_base_allgather_intra_recursivedoubling()`

源码文件路径：ompi/mca/coll/base/coll_base_allgather.c

其主要原理是：每轮通信距离加倍，数据量加倍，通过$\log_2(p)$轮交换实现全收集。相比 Gather 的单向收集，此算法实现双向并行数据交换。

```
int
ompi_coll_base_allgather_intra_recursivedoubling(const void *sbuf, size_t
scount,
                                                 struct
ompi_datatype_t *sdtype,
                                                 void* rbuf, size_t rcount,
                                                 struct
ompi_datatype_t *rdtype,
                                                 struct
ompi_communicator_t *comm,
```

```
                                                    mca_coll_base_module_t
*module)
{
    // 检查是否为2的幂次进程数
    pow2size = opal_next_poweroftwo (size);
    pow2size >>=1;

    if (pow2size != size) {
        // 非2的幂次时回退到Bruck算法
        int k = 2;
        return ompi_coll_base_allgather_intra_k_bruck(sbuf, scount, sdtype,
                                                      rbuf, rcount, rdtype,
                                                      comm, module, k);
    }

    // 初始化：复制本地数据到接收缓冲区
    if (MPI_IN_PLACE != sbuf) {
        tmpsend = (char*) sbuf;
        tmprecv = (char*) rbuf + (ptrdiff_t)rank * (ptrdiff_t)rcount * rext;
        err = ompi_datatype_sndrcv(tmpsend, scount, sdtype, tmprecv, rcount,
rdtype);
    }

    // 递归加倍通信循环
    sendblocklocation = rank;
    for (distance = 0x1; distance < size; distance<<=1) {
        remote = rank ^ distance;  // XOR操作确定通信伙伴

        if (rank < remote) {
            tmpsend = (char*)rbuf + (ptrdiff_t)sendblocklocation *
(ptrdiff_t)rcount * rext;
            tmprecv = (char*)rbuf + (ptrdiff_t)(sendblocklocation + distance)
* (ptrdiff_t)rcount * rext;
        } else {
            tmpsend = (char*)rbuf + (ptrdiff_t)sendblocklocation *
(ptrdiff_t)rcount * rext;
            tmprecv = (char*)rbuf + (ptrdiff_t)(sendblocklocation - distance)
* (ptrdiff_t)rcount * rext;
            sendblocklocation -= distance;
        }

        // 与远程进程交换数据块
        err = ompi_coll_base_sendrecv(tmpsend, (ptrdiff_t)distance *
(ptrdiff_t)rcount, rdtype,
                                      remote, MCA_COLL_BASE_TAG_ALLGATHER,
                                      tmprecv, (ptrdiff_t)distance *
(ptrdiff_t)rcount, rdtype,
                                      remote, MCA_COLL_BASE_TAG_ALLGATHER,
                                      comm, MPI_STATUS_IGNORE, rank);
    }

    return OMPI_SUCCESS;
}
```

代码 3.24 递归加倍 Allgather 算法核心代码

算法复杂度分析：递归加倍算法的时间复杂度为$O(\alpha \log_2(p) + \beta m'(p-1))$，其中$m' =$ scount。延迟复杂度为$O(\log_2 p)$，是 Gather 线性算法的显著改进；带宽复杂度为$O(m'p)$，接近理论最优。该算法通过 XOR 操作确定通信伙伴，实现完美的负载均衡，但目前限制于 2 的幂次进程数。

适用场景包括 2 的幂次规模通信子、延迟敏感应用、需要最小通信轮数的场景。相比 Gather 算法需要额外的广播阶段，递归加倍直接实现全收集，在支持的进程数范围内提供最优性能。

#### 3.4.4.2. Sparbit 算法

函数：`ompi_coll_base_allgather_intra_sparbit()`

源码文件路径：ompi/mca/coll/base/coll_base_allgather.c

其主要原理是：类似 Bruck 算法但采用反向距离和递增数据大小的对数级算法，通过稀疏位向量优化实现数据局部性感知的全收集。

```
int ompi_coll_base_allgather_intra_sparbit(const void *sbuf, size_t scount,
                                           struct
ompi_datatype_t *sdtype,
                                           void* rbuf, size_t rcount,
                                           struct
ompi_datatype_t *rdtype,
                                           struct
ompi_communicator_t *comm,
                                           mca_coll_base_module_t
*module)
{
    // 初始化通信参数
    comm_size = ompi_comm_size(comm);
    rank = ompi_comm_rank(comm);
    comm_log = ceil(log(comm_size)/log(2));
    distance <<= comm_log - 1;

    // 计算排除步骤的位掩码
    last_ignore = __builtin_ctz(comm_size);
    ignore_steps = (~((uint32_t) comm_size >> last_ignore) | 1) <<
last_ignore;

    // 执行对数级通信循环
    for (int i = 0; i < comm_log; ++i) {
       sendto = (rank + distance) % comm_size;
       recvfrom = (rank - distance + comm_size) % comm_size;
       exclusion = (distance & ignore_steps) == distance;

       // 非阻塞多块数据传输
       for (transfer_count = 0; transfer_count < data_expected - exclusion;
transfer_count++) {
           send_disp = (rank - 2 * transfer_count * distance + comm_size)
% comm_size;
```

```
            recv_disp = (rank - (2 * transfer_count + 1) * distance +
comm_size) % comm_size;

            // 使用不同标签避免消息冲突
            MCA_PML_CALL(isend(tmpsend + (ptrdiff_t) send_disp * scount *
rext, scount, rdtype,
                               sendto, MCA_COLL_BASE_TAG_HCOLL_BASE
- send_disp,
                               MCA_PML_BASE_SEND_STANDARD, comm, requests
+ transfer_count));
            MCA_PML_CALL(irecv(tmprecv + (ptrdiff_t) recv_disp * rcount *
rext, rcount, rdtype,
                               recvfrom, MCA_COLL_BASE_TAG_HCOLL_BASE
- recv_disp,
                               comm, requests + data_expected - exclusion
+ transfer_count));
        }
        ompi_request_wait_all(transfer_count * 2, requests,
MPI_STATUSES_IGNORE);

        distance >>= 1;
        data_expected = (data_expected << 1) - exclusion;
        exclusion = 0;
    }

    free(requests);
    return OMPI_SUCCESS;
}
```

代码 3.25 Sparbit Allgather 算法核心代码

算法复杂度分析：Sparbit 算法的时间复杂度为$O(\alpha \log(p) + \beta m'(p-1))$，与递归加倍相当但数据访问模式更优。该算法通过反向距离[12]和逐步增加的数据传输量实现更好的缓存局部性，在某些架构上可能优于传统算法。

适用场景包括任意进程数的通信子、对内存访问模式敏感的应用、具有复杂内存层次结构的系统。相比递归加倍的进程数限制，Sparbit 提供了更通用的解决方案。

#### 3.4.4.3. 环形算法（Ring Algorithm）

函数：`ompi_coll_base_allgather_intra_ring()`

源码文件路径：ompi/mca/coll/base/coll_base_allgather.c

其主要原理是：每个进程将数据发送给右邻居，从左邻居接收数据，通过$p-1$轮通信实现全收集。与 Gather 的树形收集不同，环形算法提供完美的负载均衡。

```
int ompi_coll_base_allgather_intra_ring(const void *sbuf, size_t scount,
```

[12]源码注释中指出该算法在《Sparbit: a new logarithmic-cost and data locality-aware MPI Allgather algorithm》中详细描述

```
                                          struct ompi_datatype_t *sdtype,
                                          void* rbuf, size_t rcount,
                                          struct ompi_datatype_t *rdtype,
                                          struct ompi_communicator_t *comm,
                                          mca_coll_base_module_t *module)
{
    // 初始化：复制本地数据
    tmprecv = (char*) rbuf + (ptrdiff_t)rank * (ptrdiff_t)rcount * rext;
    if (MPI_IN_PLACE != sbuf) {
        tmpsend = (char*) sbuf;
        err = ompi_datatype_sndrcv(tmpsend, scount, sdtype, tmprecv, rcount,
rdtype);
    }

    // 确定环形通信的邻居
    sendto = (rank + 1) % size;
    recvfrom  = (rank - 1 + size) % size;

    // 执行环形数据传递
    for (i = 0; i < size - 1; i++) {
        recvdatafrom = (rank - i - 1 + size) % size;  // 接收数据的原始来源
        senddatafrom = (rank - i + size) % size;      // 发送数据的原始来源

        tmprecv = (char*)rbuf + (ptrdiff_t)recvdatafrom * (ptrdiff_t)rcount
* rext;
        tmpsend = (char*)rbuf + (ptrdiff_t)senddatafrom * (ptrdiff_t)rcount
* rext;

        // 同时发送和接收数据
        err = ompi_coll_base_sendrecv(tmpsend, rcount, rdtype, sendto,
                                      MCA_COLL_BASE_TAG_ALLGATHER,
                                      tmprecv, rcount, rdtype, recvfrom,
                                      MCA_COLL_BASE_TAG_ALLGATHER,
                                      comm, MPI_STATUS_IGNORE, rank);
    }

    return OMPI_SUCCESS;
}
```

代码 3.26　环形 Allgather 算法核心代码

算法复杂度分析：环形算法的时间复杂度为$O((p-1)\alpha+(p-1)\beta m')$。延迟复杂度为$O(p)$，高于对数级算法但避免了根进程瓶颈；带宽复杂度为$O(m'p)$，每个数据块被传输$p-1$次，接近最优。该算法的主要优势是完美的负载均衡和对任意进程数的支持。

适用场景包括带宽受限的网络环境、需要避免热点节点的场景、不规则进程数的通信子。相比 Gather 需要根进程处理所有数据，环形算法将负载均匀分布到所有进程。

#### 3.4.4.4. 邻居交换算法（Neighbor Exchange Algorithm）

函数：`ompi_coll_base_allgather_intra_neighborexchange()`

源码文件路径：ompi/mca/coll/base/coll_base_allgather.c

其主要原理是：进程与直接邻居交换数据，然后扩大交换范围，通过$\frac{p}{2}$步实现全收集。仅适用于偶数个进程，奇数时自动回退到环形算法。

```
int
ompi_coll_base_allgather_intra_neighborexchange(const void *sbuf, size_t
scount,
                                                struct
ompi_datatype_t *sdtype,
                                                void* rbuf, size_t rcount,
                                                struct
ompi_datatype_t *rdtype,
                                                struct
ompi_communicator_t *comm,
                                                mca_coll_base_module_t
*module)
{
    if (size % 2) {
        // 奇数进程数时回退到环形算法
        return ompi_coll_base_allgather_intra_ring(sbuf, scount, sdtype,
                                                   rbuf, rcount, rdtype,
                                                   comm, module);
    }

    // 根据奇偶性确定邻居和数据流向
    even_rank = !(rank % 2);
    if (even_rank) {
        neighbor[0] = (rank + 1) % size;
        neighbor[1] = (rank - 1 + size) % size;
        recv_data_from[0] = rank;
        recv_data_from[1] = rank;
        offset_at_step[0] = (+2);
        offset_at_step[1] = (-2);
    } else {
        neighbor[0] = (rank - 1 + size) % size;
        neighbor[1] = (rank + 1) % size;
        recv_data_from[0] = neighbor[0];
        recv_data_from[1] = neighbor[0];
        offset_at_step[0] = (-2);
        offset_at_step[1] = (+2);
    }

    // 第一步：与直接邻居交换单个数据块
    tmprecv = (char*)rbuf + (ptrdiff_t)neighbor[0] * (ptrdiff_t)rcount *
rext;
    tmpsend = (char*)rbuf + (ptrdiff_t)rank * (ptrdiff_t)rcount * rext;
    err = ompi_coll_base_sendrecv(tmpsend, rcount, rdtype, neighbor[0],
                                  MCA_COLL_BASE_TAG_ALLGATHER,
                                  tmprecv, rcount, rdtype, neighbor[0],
                                  MCA_COLL_BASE_TAG_ALLGATHER,
                                  comm, MPI_STATUS_IGNORE, rank);

    // 后续步骤：交换逐步增大的数据块
    send_data_from = even_rank ? rank : recv_data_from[0];
```

```
45
46      for (i = 1; i < (size / 2); i++) {
47          const int i_parity = i % 2;
48          recv_data_from[i_parity] =
49              (recv_data_from[i_parity] + offset_at_step[i_parity] + size)
    % size;
50
51          tmprecv = (char*)rbuf + (ptrdiff_t)recv_data_from[i_parity] *
    (ptrdiff_t)rcount * rext;
52          tmpsend = (char*)rbuf + (ptrdiff_t)send_data_from * rcount * rext;
53
54          // 交换两个数据块
55          err = ompi_coll_base_sendrecv(tmpsend, (ptrdiff_t)2 *
    (ptrdiff_t)rcount, rdtype,
56                                        neighbor[i_parity],
    MCA_COLL_BASE_TAG_ALLGATHER,
57                                        tmprecv, (ptrdiff_t)2 *
    (ptrdiff_t)rcount, rdtype,
58                                        neighbor[i_parity],
    MCA_COLL_BASE_TAG_ALLGATHER,
59                                        comm, MPI_STATUS_IGNORE, rank);
60
61          send_data_from = recv_data_from[i_parity];
62      }
63
64      return OMPI_SUCCESS;
65  }
```

代码 3.27 邻居交换 Allgather 算法核心代码

算法复杂度分析：邻居交换算法的时间复杂度为$O(\left(\frac{p}{2}\right)\alpha + (p-1)\beta m')$。延迟复杂度为$O(\frac{p}{2})$，比环形算法减半；带宽复杂度为$O(m'p)$，通过双向同时交换[13]获得更好的带宽利用率。

适用场景包括偶数个进程的通信子、需要减少通信轮数的应用、具有良好双向带宽的网络环境。相比环形算法的单向传递，邻居交换通过双向并行传输提升效率。

#### 3.4.4.5. K-Bruck 算法

函数：`ompi_coll_base_allgather_intra_k_bruck()`

源码文件路径：ompi/mca/coll/base/coll_base_allgather.c

其主要原理是：扩展的 Bruck 算法，支持任意基数 k，通过$\log_{k(p)}$步实现全收集，利用非阻塞通信充分利用多端口优势。

[13]邻居交换算法的特点：

1. 进程数限制：仅适用于偶数个进程
2. 双向交换：每步都进行双向同时交换，提高效率
3. 渐进增大：交换的数据块大小逐步增大
4. 回退机制：奇数进程时自动回退到环形算法

```
int ompi_coll_base_allgather_intra_k_bruck(const void *sbuf, size_t scount,
                                           struct ompi_datatype_t *sdtype,
                                           void* rbuf, size_t rcount,
                                           struct ompi_datatype_t *rdtype,
                                           struct ompi_communicator_t *comm,
                                           mca_coll_base_module_t *module,
                                           int radix)
{
    // 为非0进程分配临时缓冲区用于数据重排
    if (0 != rank) {
        rsize = opal_datatype_span(&rdtype->super, (size_t)rcount * (size - rank), &rgap);
        tmp_buf = (char *) malloc(rsize);
        tmp_buf_start = tmp_buf - rgap;
    }

    // 执行k-进制通信循环
    max_reqs = 2 * (radix - 1);
    reqs = ompi_coll_base_comm_get_reqs(module->base_data, max_reqs);
    recvcount = 1;
    tmpsend = (char*) rbuf;

    for (distance = 1; distance < size; distance *= radix) {
        num_reqs = 0;
        for (int j = 1; j < radix; j++) {
            if (distance * j >= size) break;

            src = (rank + distance * j) % size;
            dst = (rank - distance * j + size) % size;
            tmprecv = tmpsend + (ptrdiff_t)distance * j * rcount * rextent;

            // 计算传输数据量
            if (distance <= (size / radix)) {
                recvcount = distance;
            } else {
                recvcount = (distance < (size - distance * j)?
                             distance:(size - distance * j));
            }

            // 非阻塞发送和接收
            err = MCA_PML_CALL(irecv(tmprecv, recvcount * rcount, rdtype, src,
                                     MCA_COLL_BASE_TAG_ALLGATHER, comm, &reqs[num_reqs++]));
            err = MCA_PML_CALL(isend(tmpsend, recvcount * rcount, rdtype, dst,
                                     MCA_COLL_BASE_TAG_ALLGATHER,
                                     MCA_PML_BASE_SEND_STANDARD, comm, &reqs[num_reqs++]));
        }
        err = ompi_request_wait_all(num_reqs, reqs, MPI_STATUSES_IGNORE);
    }

    // 最终数据重排（除rank 0外）
```

```
50      if (0 != rank) {
51          // 三步数据重排序以获得正确的进程顺序
52          err = ompi_datatype_copy_content_same_ddt(rdtype,
                                                      ((ptrdiff_t)(size - rank)
53  * rcount),
54                                                    tmp_buf_start, rbuf);
55
56          tmpsend = (char*) rbuf + (ptrdiff_t)(size - rank) * rcount * rextent;
            err = ompi_datatype_copy_content_same_ddt(rdtype, (ptrdiff_t)rank
57  * rcount,
58                                                    rbuf, tmpsend);
59
60          tmprecv = (char*) rbuf + (ptrdiff_t)rank * rcount * rextent;
61          err = ompi_datatype_copy_content_same_ddt(rdtype,
                                                      (ptrdiff_t)(size - rank)
62  * rcount,
63                                                    tmprecv, tmp_buf_start);
64      }
65
66      if(tmp_buf != NULL) free(tmp_buf);
67      return MPI_SUCCESS;
68  }
```

代码 3.28 K-Bruck Allgather 算法核心代码

算法复杂度分析：K-Bruck 算法的时间复杂度为$O\left(\alpha \log_{k(p)} + \beta m'(p-1)\right)$。通过调节 radix 参数[14]可以在延迟和带宽之间权衡：较大的 k 减少通信轮数但增加单轮复杂度。该算法支持任意进程数且具有良好的扩展性。

适用场景包括需要调节延迟-带宽权衡的应用、具有多端口网络的系统、中大规模任意进程数的通信子。相比递归加倍的进程数限制，K-Bruck 提供了更灵活的解决方案。

#### 3.4.4.6. 其它 Allgather 算法

除了上述核心算法外，Open MPI 还实现了以下专用算法：

- 两进程优化算法（`ompi_coll_base_allgather_intra_two_procs`）
  专门针对两进程情况的简单交换算法，直接进行单次数据交换。
- 基础线性算法（`ompi_coll_base_allgather_intra_basic_linear`）
  组合使用 Gather 和 Broadcast 实现 Allgather，适合作为复杂算法的回退选择。
- 直接消息传递算法（`ompi_coll_base_allgather_direct_messaging`）
  每个进程直接与所有其他进程通信的贪心算法，避免根节点瓶颈但可能造成网络拥塞。

[14]K-Bruck 算法的优势：
1. 可调基数：通过 radix 参数调节延迟-带宽权衡
2. 非阻塞通信：充分利用网络的多端口能力
3. 数据重排：最终进行本地数据重排获得正确顺序
4. 扩展 Bruck：基于经典 Bruck 算法的多端口扩展版本

#### 3.4.4.7. 总结

基于上述对`MPI_Allgather`的算法的讨论，整理得如下表格：

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
|---|---|---|---|---|
| 递归加倍算法 | `ompi_coll_base_allgather_intra_recursivedoubling` | 无 | $O(\alpha \log_2(p) + \beta m'(p-1))$ | 2 的幂次进程数 延迟敏感应用 |
| Sparbit 算法 | `ompi_coll_base_allgather_intra_sparbit` | 无 | $O(\alpha \log(p) + \beta m'(p-1))$ | 数据局部敏感 任意进程数 |
| 环形算法 | `ompi_coll_base_allgather_intra_ring` | 无 | $O((p-1)\alpha + (p-1)\beta m')$ | 带宽受限网络 负载均衡需求 |
| 邻居交换算法 | `ompi_coll_base_allgather_intra_neighborexchange` | 无 | $O((\frac{p}{2})\alpha + (p-1)\beta m')$ | 偶数进程数 双向带宽充足 |
| K-Bruck 算法 | `ompi_coll_base_allgather_intra_k_bruck` | `radix` | $O\left(\alpha \log_{k(p)} + \beta m'(p-1)\right)$ | 延迟-带宽权衡 多端口网络 |
| 两进程算法 | `ompi_coll_base_allgather_intra_two_procs` | 无 | $O(\alpha + \beta m')$ | 两进程通信子 简单优化 |
| 基础线性算法 | `ompi_coll_base_allgather_intra_basic_linear` | 无 | $O((p-1)\alpha + p\beta m')$ | 回退选择 Gather+Bcast |
| 直接消息算法 | `ompi_coll_base_allgather_direct_messaging` | 无 | $O((p-1)\alpha + (p-1)\beta m')$ | 小消息低延迟 避免中转开销 |

表 3.4：Open MPI Allgather 算法总结

参数说明：

- $\alpha$: 通信延迟参数，$\beta$: 带宽倒数参数
- $m'$: 单个进程数据大小（scount），$p$: 进程数量
- `radix`: K-Bruck 算法的基数参数（k 值）

### 3.4.5. Reduce

`Reduce`的函数原型如下：

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

其中：`send_data` 是每个进程要发送的数据，`recv_data` 是根进程接收归约结果的缓冲区（仅在根进程有效）。`count` 指定参与运算的数据元素数量，`op` 指定归约操作（如 `MPI_SUM`、`MPI_MAX` 等）。与 Gather 操作不同，Reduce 不仅收集数据，还对收集的数据执行指定的归约运算。

Open MPI 为 Reduce 操作提供了多种算法实现：

#### 3.4.5.1. 通用树形算法（Generic Tree Algorithm）

函数：`ompi_coll_base_reduce_generic()`

源码文件路径：ompi/mca/coll/base/coll_base_reduce.c

其主要原理是：提供通用的树形归约框架，支持任意树形拓扑结构和消息分段。非叶节点从子节点接收数据并执行归约操作，然后向父节点发送结果。支持流水线处理和非阻塞通信优化。

```
int ompi_coll_base_reduce_generic( const void* sendbuf, void* recvbuf, size_t original_count,
                                   ompi_datatype_t* datatype, ompi_op_t* op,
                                   int root, ompi_communicator_t* comm,
                                   mca_coll_base_module_t *module,
                                   ompi_coll_tree_t* tree, size_t count_by_segment,
                                   int max_outstanding_reqs )
{
    // 计算分段参数
    num_segments = (int)(((size_t)original_count + (size_t)count_by_segment - (size_t)1) / (size_t)count_by_segment);
    segment_increment = (ptrdiff_t)count_by_segment * extent;

    if( tree->tree_nextsize > 0 ) {
        // 非叶节点：接收子节点数据并执行归约

        // 分配累积缓冲区
        accumbuf = (char*)recvbuf;
        if( (NULL == accumbuf) || (root != rank) ) {
            size = opal_datatype_span(&datatype->super, original_count, &gap);
            accumbuf_free = (char*)malloc(size);
            accumbuf = accumbuf_free - gap;
        }

        // 处理非交换操作的特殊情况
        if (!ompi_op_is_commute(op) && MPI_IN_PLACE != sendbuf) {
            ompi_datatype_copy_content_same_ddt(datatype, original_count,
                                                (char*)accumbuf, (char*)sendtmpbuf);
        }

        // 分段流水线处理
        for( segindex = 0; segindex <= num_segments; segindex++ ) {
            for( i = 0; i < tree->tree_nextsize; i++ ) {
                // 发布非阻塞接收
```

```
33                 if( segindex < num_segments ) {
                       ret = MCA_PML_CALL(irecv(local_recvbuf, recvcount,
34 datatype,
                                                tree->tree_next[i],
35 MCA_COLL_BASE_TAG_REDUCE,
36                                              comm, &reqs[inbi]));
37                 }
38
39                 // 等待前一个请求完成并执行归约
                   ret = ompi_request_wait(&reqs[inbi ^ 1],
40 MPI_STATUSES_IGNORE);
41                 local_op_buffer = inbuf[inbi ^ 1];
42
43                 // 执行归约操作
44                 if( i > 0 ) {
45                     ompi_op_reduce(op, local_op_buffer,
                                       accumbuf + (ptrdiff_t)segindex
46 * (ptrdiff_t)segment_increment,
47                                     recvcount, datatype );
48                 }
49
50                 // 向父节点发送累积结果
51                 if (rank != tree->tree_root && segindex > 0) {
                       ret = MCA_PML_CALL( send( accumulator, prevcount,
52 datatype, tree->tree_prev,
                                                 MCA_COLL_BASE_TAG_REDUCE,
53 MCA_PML_BASE_SEND_STANDARD, comm) );
54                 }
55
56                 inbi = inbi ^ 1;
57             }
58         }
59     } else {
60         // 叶节点：发送数据到父节点
61
           if ((0 == max_outstanding_reqs) || (num_segments <=
62 max_outstanding_reqs)) {
63             // 使用阻塞发送
64             segindex = 0;
65             while ( original_count > 0) {
                   ret = MCA_PML_CALL( send((char*)sendbuf +
66 (ptrdiff_t)segindex * (ptrdiff_t)segment_increment,
                                            count_by_segment, datatype,
67 tree->tree_prev,
                                            MCA_COLL_BASE_TAG_REDUCE,
68 MCA_PML_BASE_SEND_STANDARD, comm) );
69                 segindex++;
70                 original_count -= count_by_segment;
71             }
72         } else {
73             // 使用流控制的非阻塞发送
```

```
            sreq = ompi_coll_base_comm_get_reqs(module->base_data,
74  max_outstanding_reqs);
75
76              // 发送前max_outstanding_reqs个分段
                for (segindex = 0; segindex < max_outstanding_reqs; segindex+
77  +) {
                    ret = MCA_PML_CALL( isend((char*)sendbuf +
78  (ptrdiff_t)segindex * (ptrdiff_t)segment_increment,
                                              count_by_segment, datatype,
79  tree->tree_prev,
                                              MCA_COLL_BASE_TAG_REDUCE,
80  MCA_PML_BASE_SEND_SYNCHRONOUS,
81                                            comm, &sreq[segindex]) );
82                  original_count -= count_by_segment;
83              }
84
85              // 流水线处理剩余分段
86              creq = 0;
87              while ( original_count > 0 ) {
88                  ret = ompi_request_wait(&sreq[creq], MPI_STATUS_IGNORE);
                    ret = MCA_PML_CALL( isend((char*)sendbuf +
89  (ptrdiff_t)segindex * (ptrdiff_t)segment_increment,
                                              count_by_segment, datatype,
90  tree->tree_prev,
                                              MCA_COLL_BASE_TAG_REDUCE,
91  MCA_PML_BASE_SEND_SYNCHRONOUS,
92                                            comm, &sreq[creq]) );
93                  creq = (creq + 1) % max_outstanding_reqs;
94                  segindex++;
95                  original_count -= count_by_segment;
96              }
97
                ret = ompi_request_wait_all( max_outstanding_reqs, sreq,
98  MPI_STATUSES_IGNORE );
99          }
100     }
101
102     return OMPI_SUCCESS;
103 }
```

代码 3.29 通用树形 Reduce 算法核心代码

算法复杂度分析：通用树形归约算法的时间复杂度取决于具体的树结构，通常为$O(\alpha \log(p) + \beta m)$，其中$m$为总数据量。该算法通过分段处理支持大消息归约，通过流水线技术实现通信-计算重叠。非交换操作需要特殊处理以保证运算顺序的正确性。

适用场景包括需要自定义树形拓扑的应用、大消息归约、需要流水线优化的高性能计算场景。该算法作为 Open MPI 中其他具体树形算法的基础框架，提供了灵活的参数配置和优化选项。

#### 3.4.5.2. 二项式树算法（Binomial Tree Algorithm）

函数：`ompi_coll_base_reduce_intra_binomial()`

源码文件路径：ompi/mca/coll/base/coll_base_reduce.c

其主要原理是：使用二项式树结构进行归约，通过调用通用树形算法实现。相比 Gather 的单纯数据收集，该算法在每个节点执行归约运算，减少了网络传输的数据量。

```
int ompi_coll_base_reduce_intra_binomial( const void *sendbuf, void *recvbuf,
                                          size_t count,
ompi_datatype_t* datatype,
                                          ompi_op_t* op, int root,
                                          ompi_communicator_t* comm,
                                          mca_coll_base_module_t *module,
                                          uint32_t segsize,
int max_outstanding_reqs  )
{
    size_t segcount = count;
    size_t typelng;
    mca_coll_base_module_t *base_module = (mca_coll_base_module_t*) module;
    mca_coll_base_comm_t *data = base_module->base_data;

    COLL_BASE_UPDATE_IN_ORDER_BMTREE( comm, base_module, root );

    // 计算分段参数
    ompi_datatype_type_size( datatype, &typelng );
    COLL_BASE_COMPUTED_SEGCOUNT( segsize, typelng, segcount );

    // 调用通用树形算法
    return ompi_coll_base_reduce_generic( sendbuf, recvbuf, count, datatype,
                                          op, root, comm, module,
                                          data->cached_in_order_bmtree,
                                          segcount, max_outstanding_reqs );
}
```

代码 3.30　二项式树 Reduce 算法核心代码

算法复杂度分析：二项式树归约算法的时间复杂度为$O(\alpha \log(p) + \beta m)$。相比二项式树 Gather 的$O(\alpha \log(p) + \beta m(p-1))$，归约操作通过在中间节点执行运算显著减少了数据传输量。延迟复杂度为$O(\log p)$，适合大规模通信子。

适用场景包括大规模通信子、大消息归约、延迟敏感的归约操作。该算法通过二项式树的平衡结构和归约运算的数据压缩特性，在大多数场景下提供优秀的性能。

#### 3.4.5.3. K 项树算法（K-nomial Tree Algorithm）

函数：`ompi_coll_base_reduce_intra_knomial()`

源码文件路径：ompi/mca/coll/base/coll_base_reduce.c

其主要原理是：使用可配置基数的 k 项树结构进行归约，每个节点可以有多个子节点。通过调节 radix 参数在延迟和并发度之间权衡，支持非阻塞接收优化。

```
int ompi_coll_base_reduce_intra_knomial( const void *sendbuf, void *recvbuf,
```

```
                                                size_t count,
2
ompi_datatype_t* datatype,
3
                                                ompi_op_t* op, int root,
4
                                                ompi_communicator_t* comm,
5
                                                mca_coll_base_module_t *module,
                                                uint32_t segsize, int
6
max_outstanding_reqs, int radix)
7
{
8
    // 创建k项树
9
    COLL_BASE_UPDATE_KMTREE(comm, base_module, root, radix);
10
    tree = data->cached_kmtree;
11
    num_children = tree->tree_nextsize;
12
13
    // 分配子节点数据缓冲区
14
    if(!is_leaf) {
        buf_size = opal_datatype_span(&datatype->super, (int64_t)count *
15
num_children, &gap);
16
        child_buf = (char *)malloc(buf_size);
17
        child_buf_start = child_buf - gap;
18
        reqs = ompi_coll_base_comm_get_reqs(data, max_reqs);
19
    }
20
21
    // 非阻塞接收所有子节点数据
22
    for (int i = 0; i < num_children; i++) {
23
        int child = tree->tree_next[i];
        err = MCA_PML_CALL(irecv(child_buf_start + (ptrdiff_t)i * count
24
* extent,
                                 count, datatype, child,
25
MCA_COLL_BASE_TAG_REDUCE,
26
                                 comm, &reqs[num_reqs++]));
27
    }
28
29
    // 等待所有接收完成
30
    if (num_reqs > 0) {
31
        err = ompi_request_wait_all(num_reqs, reqs, MPI_STATUS_IGNORE);
32
    }
33
34
    // 执行归约操作
35
    for (int i = 0; i < num_children; i++) {
36
        ompi_op_reduce(op, child_buf_start + (ptrdiff_t)i * count * extent,
37
                       reduce_buf, count, datatype);
38
    }
39
40
    // 向父节点发送结果
41
    if (rank != root) {
        err = MCA_PML_CALL(send(reduce_buf_start, count, datatype, tree-
42
>tree_prev,
                                MCA_COLL_BASE_TAG_REDUCE,
43
MCA_PML_BASE_SEND_STANDARD, comm));
44
    }
45
46
    // 根节点复制最终结果
47
    if (rank == root) {
```

```
48          err = ompi_datatype_copy_content_same_ddt(datatype, count,
                                                      (char*)recvbuf,
49  (char*)reduce_buf_start);
50      }
51
52      return MPI_SUCCESS;
53  }
```

代码 3.31 K 项树 Reduce 算法核心代码

算法复杂度分析：K 项树归约算法的时间复杂度为$O\left(\alpha \log_{k(p)} + \beta m\right)$。通过调节 radix 参数可以在通信轮数和单轮并发度之间权衡：较大的 k 减少延迟但增加单轮复杂度。该算法支持任意进程数且具有良好的扩展性。

适用场景包括需要调节延迟-并发度权衡的应用、具有多端口网络的系统、中大规模任意进程数的通信子。相比二项式树的固定结构，K 项树提供了更灵活的性能调优选项。

#### 3.4.5.4. 有序二叉树算法（In-order Binary Tree Algorithm）

函数：`ompi_coll_base_reduce_intra_in_order_binary()`

源码文件路径：ompi/mca/coll/base/coll_base_reduce.c

其主要原理是：专门为非交换归约操作设计的算法，使用有序二叉树确保运算顺序的正确性。必须使用进程号(size-1)作为内部根节点，最后将结果传输给实际根进程。

```
1   int ompi_coll_base_reduce_intra_in_order_binary( const void *sendbuf, void
    *recvbuf,
                                                     size_t count,
2   ompi_datatype_t* datatype,
3                                                    ompi_op_t* op, int root,
4                                                    ompi_communicator_t* comm,
                                                     mca_coll_base_module_t
5   *module,
                                                     uint32_t segsize,
6   int max_outstanding_reqs  )
7   {
8       // 有序二叉树必须使用(size-1)作为内部根节点以保证运算顺序
9       io_root = size - 1;
10      use_this_sendbuf = (void *)sendbuf;
11      use_this_recvbuf = recvbuf;
12
13      if (io_root != root) {
14          dsize = opal_datatype_span(&datatype->super, count, &gap);
15
16          if ((root == rank) && (MPI_IN_PLACE == sendbuf)) {
17              // 实际根进程使用IN_PLACE时的特殊处理
18              tmpbuf_free = (char *) malloc(dsize);
19              tmpbuf = tmpbuf_free - gap;
20              ompi_datatype_copy_content_same_ddt(datatype, count,
```

```
                                                    (char*)tmpbuf,
21 (char*)recvbuf);
22             use_this_sendbuf = tmpbuf;
23         } else if (io_root == rank) {
24             // 内部根进程分配临时接收缓冲区
25             tmpbuf_free = (char *) malloc(dsize);
26             tmpbuf = tmpbuf_free - gap;
27             use_this_recvbuf = tmpbuf;
28         }
29     }
30
31     // 使用有序二叉树执行归约
32     ret = ompi_coll_base_reduce_generic( use_this_sendbuf, use_this_recvbuf,
   count, datatype,
33                                          op, io_root, comm, module,
34                                          data->cached_in_order_bintree,
35                                          segcount, max_outstanding_reqs );
36
37     // 处理内部根与实际根不同的情况
38     if (io_root != root) {
39         if (root == rank) {
40             // 实际根进程从内部根接收最终结果
41             ret = MCA_PML_CALL(recv(recvbuf, count, datatype, io_root,
                                       MCA_COLL_BASE_TAG_REDUCE,
42 comm, MPI_STATUS_IGNORE));
43         } else if (io_root == rank) {
44             // 内部根进程向实际根发送最终结果
45             ret = MCA_PML_CALL(send(use_this_recvbuf, count, datatype, root,
                                       MCA_COLL_BASE_TAG_REDUCE,
46 MCA_PML_BASE_SEND_STANDARD, comm));
47         }
48     }
49
50     return MPI_SUCCESS;
51 }
```

代码 3.32 有序二叉树 Reduce 算法核心代码

算法复杂度分析：有序二叉树归约算法的时间复杂度为$O(\alpha \log(p) + \beta m)$，与标准二叉树相同，但增加了根节点间数据传输的开销。该算法确保了非交换操作的运算顺序正确性，这是处理诸如矩阵乘法、字符串连接等非交换操作的关键。

适用场景包括非交换归约操作、需要严格运算顺序的数值计算、字符串处理等应用。该算法是 Open MPI 中专门处理非交换操作的重要实现，确保了数学运算的正确性。

#### 3.4.5.5. 分散-聚集算法（Reduce-scatter-gather Algorithm）

函数：`ompi_coll_base_reduce_intra_redscat_gather()`

源码文件路径：ompi/mca/coll/base/coll_base_reduce.c

其主要原理是：实现 Rabenseifner 算法，先执行 reduce-scatter 将数据分散到各进程并执行部分归约，再通过二项式树 gather 收集最终结果。适合大规模归约操作，特别是当数据量大于进程数时。

```
int ompi_coll_base_reduce_intra_redscat_gather(
    const void *sbuf, void *rbuf, size_t count, struct ompi_datatype_t *dtype,
    struct ompi_op_t *op, int root, struct ompi_communicator_t *comm,
    mca_coll_base_module_t *module)
{
    // 第一步：处理非2的幂次进程数
    int nprocs_rem = comm_size - nprocs_pof2;

    if (rank < 2 * nprocs_rem) {
        int count_lhalf = count / 2;
        int count_rhalf = count - count_lhalf;

        if (rank % 2 != 0) {
            // 奇数进程：与左邻居交换并归约右半部分
            err = ompi_coll_base_sendrecv(rbuf, count_lhalf, dtype, rank - 1,
                                          MCA_COLL_BASE_TAG_REDUCE,
                                          (char *)tmp_buf + (ptrdiff_t)count_lhalf * extent,
                                          count_rhalf, dtype, rank - 1,
                                          MCA_COLL_BASE_TAG_REDUCE, comm, MPI_STATUS_IGNORE, rank);

            ompi_op_reduce(op, (char *)tmp_buf + (ptrdiff_t)count_lhalf * extent,
                           (char *)rbuf + count_lhalf * extent, count_rhalf, dtype);

            err = MCA_PML_CALL(send((char *)rbuf + (ptrdiff_t)count_lhalf * extent,
                                    count_rhalf, dtype, rank - 1,
                                    MCA_COLL_BASE_TAG_REDUCE, MCA_PML_BASE_SEND_STANDARD, comm));
            vrank = -1;  // 不参与后续阶段
        } else {
            // 偶数进程：与右邻居交换并归约左半部分
            err = ompi_coll_base_sendrecv((char *)rbuf + (ptrdiff_t)count_lhalf * extent,
                                          count_rhalf, dtype, rank + 1, MCA_COLL_BASE_TAG_REDUCE,
                                          tmp_buf, count_lhalf, dtype, rank + 1,
                                          MCA_COLL_BASE_TAG_REDUCE, comm, MPI_STATUS_IGNORE, rank);

            ompi_op_reduce(op, tmp_buf, rbuf, count_lhalf, dtype);
```

```
36
37              err = MCA_PML_CALL(recv((char *)rbuf + (ptrdiff_t)count_lhalf
* extent,
38                                      count_rhalf, dtype, rank + 1,
39                                      MCA_COLL_BASE_TAG_REDUCE,
comm, MPI_STATUS_IGNORE));
40              vrank = rank / 2;
41          }
42      } else {
43          vrank = rank - nprocs_rem;
44      }
45
46      // 第二步：递归减半的reduce-scatter
47      if (vrank != -1) {
48          step = 0;
49          wsize = count;
50
51          for (int mask = 1; mask < nprocs_pof2; mask <<= 1) {
52              int vdest = vrank ^ mask;
53              int dest = (vdest < nprocs_rem) ? vdest * 2 : vdest +
nprocs_rem;
54
55              // 确定发送和接收的数据范围
56              if (rank < dest) {
57                  rcount[step] = wsize / 2;
58                  scount[step] = wsize - rcount[step];
59                  sindex[step] = rindex[step] + rcount[step];
60              } else {
61                  scount[step] = wsize / 2;
62                  rcount[step] = wsize - scount[step];
63                  rindex[step] = sindex[step] + scount[step];
64              }
65
66              // 交换数据并执行归约
67              err = ompi_coll_base_sendrecv((char *)rbuf +
(ptrdiff_t)sindex[step] * extent,
68                                            scount[step], dtype,
dest, MCA_COLL_BASE_TAG_REDUCE,
69                                            (char *)tmp_buf +
(ptrdiff_t)rindex[step] * extent,
70                                            rcount[step], dtype,
dest, MCA_COLL_BASE_TAG_REDUCE,
71                                            comm, MPI_STATUS_IGNORE, rank);
72
73              ompi_op_reduce(op, (char *)tmp_buf + (ptrdiff_t)rindex[step]
* extent,
74                             (char *)rbuf + (ptrdiff_t)rindex[step] * extent,
75                             rcount[step], dtype);
76
77              // 更新下一轮的窗口
78              if (step + 1 < nsteps) {
79                  rindex[step + 1] = rindex[step];
80                  sindex[step + 1] = rindex[step];
```

```
                wsize = rcount[step];
                step++;
            }
        }
    }

    // 第三步：二项式树gather收集最终结果
    if (vrank != -1) {
        step = nsteps - 1;

        for (int mask = nprocs_pof2 >> 1; mask > 0; mask >>= 1) {
            int vdest = vrank ^ mask;
            int dest = (vdest < nprocs_rem) ? vdest * 2 : vdest +
nprocs_rem;

            // 确定是发送还是接收
            vdest_tree = vdest >> step;
            vdest_tree <<= step;
            vroot_tree = vroot >> step;
            vroot_tree <<= step;

            if (vdest_tree == vroot_tree) {
                err = MCA_PML_CALL(send((char *)rbuf +
(ptrdiff_t)rindex[step] * extent,
                                        rcount[step], dtype,
dest, MCA_COLL_BASE_TAG_REDUCE,
                                        MCA_PML_BASE_SEND_STANDARD, comm));
                break;
            } else {
                err = MCA_PML_CALL(recv((char *)rbuf +
(ptrdiff_t)sindex[step] * extent,
                                        scount[step], dtype,
dest, MCA_COLL_BASE_TAG_REDUCE,
                                        comm, MPI_STATUS_IGNORE));
            }
            step--;
        }
    }

    return err;
}
```

代码 3.33 分散-聚集 Reduce 算法核心代码

算法复杂度分析：分散-聚集归约算法的时间复杂度为$O(\alpha \log(p) + \beta m)$，但具有更好的可扩展性。该算法特别适合 count >= p 的场景，通过 reduce-scatter 阶段的并行处理和 gather 阶段的结果收集，在大规模系统上表现优异。算法要求操作必须是交换的。

适用场景包括大规模并行系统、大数据量归约操作、高带宽网络环境。该算法是基于 Rabenseifner 论文的经典实现，在 HPC 领域广泛应用于大规模数值计算。

#### 3.4.5.6. 线性算法（Linear Algorithm）

函数: `ompi_coll_base_reduce_intra_basic_linear()`

源码文件路径: ompi/mca/coll/base/coll_base_reduce.c

其主要原理是:所有非根进程将数据发送给根进程,根进程按相反顺序接收数据并执行归约操作。实现简单但根进程会成为瓶颈。

```
int ompi_coll_base_reduce_intra_basic_linear(const void *sbuf, void *rbuf,
size_t count,
                                             struct ompi_datatype_t *dtype,
struct ompi_op_t *op,
                                             int root, struct
ompi_communicator_t *comm,
                                             mca_coll_base_module_t *module)
{
    // 非根进程: 发送数据并返回
    if (rank != root) {
        err = MCA_PML_CALL(send(sbuf, count, dtype, root,
                                MCA_COLL_BASE_TAG_REDUCE,
MCA_PML_BASE_SEND_STANDARD, comm));
        return err;
    }

    // 根进程: 处理MPI_IN_PLACE情况
    if (MPI_IN_PLACE == sbuf) {
        sbuf = rbuf;
        inplace_temp_free = (char*)malloc(dsize);
        rbuf = inplace_temp_free - gap;
    }

    // 初始化接收缓冲区: 从最高进程号开始
    if (rank == (size - 1)) {
        err = ompi_datatype_copy_content_same_ddt(dtype, count, (char*)rbuf,
(char*)sbuf);
    } else {
        err = MCA_PML_CALL(recv(rbuf, count, dtype, size - 1,
                                MCA_COLL_BASE_TAG_REDUCE, comm,
MPI_STATUS_IGNORE));
    }

    // 按降序接收数据并执行归约
    for (i = size - 2; i >= 0; --i) {
        if (rank == i) {
            inbuf = (char*)sbuf;
        } else {
            err = MCA_PML_CALL(recv(pml_buffer, count, dtype, i,
                                    MCA_COLL_BASE_TAG_REDUCE,
comm, MPI_STATUS_IGNORE));
            inbuf = pml_buffer;
        }

        // 执行归约操作
        ompi_op_reduce(op, inbuf, rbuf, count, dtype);
```

```
40      }
41
42      // 处理MPI_IN_PLACE的最终复制
43      if (NULL != inplace_temp_free) {
44          err = ompi_datatype_copy_content_same_ddt(dtype, count, (char*)sbuf,
    rbuf);
45          free(inplace_temp_free);
46      }
47
48      return MPI_SUCCESS;
49  }
```

代码 3.34 线性 Reduce 算法核心代码

算法复杂度分析：线性归约算法的时间复杂度为$O((p-1)\alpha+(p-1)\beta m)$。延迟复杂度为$O(p)$，带宽复杂度为$O(pm)$，根进程成为明显瓶颈。该算法通过按降序接收数据确保了非交换操作的正确性。

适用场景包括小规模通信子、小消息归约、作为复杂算法的回退选择，以及调试和验证目的。虽然性能较差，但实现简单可靠，在某些特定场景下仍有价值。

#### 3.4.5.7. 其它 Reduce 算法

除了上述核心算法外，Open MPI 还实现了以下专用算法：

- 链式归约算法（`ompi_coll_base_reduce_intra_chain`）
  形成一个或多个通信链，数据沿链向根进程归约，支持通过 fanout 参数控制并行链数。
- 流水线归约算法（`ompi_coll_base_reduce_intra_pipeline`）
  将大消息分段，采用流水线方式在链式结构上进行归约，提升大消息处理效率。
- 二进制树归约算法（`ompi_coll_base_reduce_intra_binary`）
  使用完全二叉树结构进行归约，通过调用通用算法框架实现。

#### 3.4.5.8. 总结

基于上述对`MPI_Reduce`的算法的讨论，整理得如下表格：

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
| --- | --- | --- | --- | --- |
| 通用树形算法 | `ompi_coll_base_reduce_generic` | `tree segcount max_reqs` | 取决于树结构 | 通用框架<br>任意树形拓扑 |
| 二项式树算法 | `ompi_coll_base_reduce_intra_binomial` | `segsize max_reqs` | $O(\alpha\log(p)+\beta m)$ | 大规模通信子<br>平衡性能需求 |
| K 项树算法 | `ompi_coll_base_reduce_intra_knomial` | `segsize max_reqs radix` | $O\left(\alpha\log_{k(p)}+\beta m\right)$ | 延迟-并发权衡<br>多端口网络 |

| 算法名称 | 函数名称 | 可选参数 | 时间复杂度 | 适用场景 |
|---|---|---|---|---|
| 有序二叉树算法 | `ompi_coll_base_reduce_intra_in_order_binary` | `segsize` `max_reqs` | $O(\alpha \log(p) + \beta m)$ | 非交换操作<br>严格运算顺序 |
| 分散-聚集算法 | `ompi_coll_base_reduce_intra_redscat_gather` | 无 | $O(\alpha \log(p) + \beta m)$ | 大规模系统<br>大数据量归约 |
| 线性算法 | `ompi_coll_base_reduce_intra_basic_linear` | 无 | $O((p-1)\alpha + (p-1)\beta m)$ | 小规模通信子<br>回退选择 |
| 链式归约算法 | `ompi_coll_base_reduce_intra_chain` | `segsize` `fanout` `max_reqs` | $O(\frac{p}{\text{fanout}} \cdot \alpha + \beta m)$ | 特定网络拓扑<br>多链并行 |
| 流水线算法 | `ompi_coll_base_reduce_intra_pipeline` | `segsize` `max_reqs` | $O(\alpha \log(p) + \beta m)$ | 大消息归约<br>流水线重叠 |
| 二进制树算法 | `ompi_coll_base_reduce_intra_binary` | `segsize` `max_reqs` | $O(\alpha \log(p) + \beta m)$ | 完全二叉树结构<br>负载平衡 |

表 3.4：Open MPI Reduce 算法总结

参数说明：

- $\alpha$: 通信延迟参数，$\beta$: 带宽倒数参数
- $m$: 消息大小，$p$: 进程数量
- `segsize`：控制消息分段大小的参数
- `max_reqs`：最大未完成请求数，用于流控制
- `radix`：K 项树的基数参数（k 值）
- `fanout`：链式算法中的扇出参数
- `tree`：指定使用的树结构类型
- `segcount`：每段传输的元素数量

## 3.5. 小结

本章通过深入分析 Open MPI 源码，系统阐述了集合通信算法的架构设计与核心实现。研究发现，Open MPI 采用模块化组件架构（MCA），将集合通信实现分为 `base`、`basic`、`tuned` 等专门化组件，通过 `mca_coll_base_comm_select()` 机制根据运行时参数动态选择最优算法实现，为用户提供透明而高效的性能优化。

在算法实现层面，本章详细分析了五种核心集合通信操作的多种算法变体。Broadcast 操作提供了 10 种算法实现，从线性算法的$O(p\alpha + p\beta m)$复杂度到 K 项树算法的$O\left(\log_{k(p)} \alpha + \beta m\right)$最优带宽效率，涵盖了从小规模到大规模通信子的各种应用场景。Scatter 操作实现了 7 种算法，其中二项式树算法通过$O\left(\alpha \log(p) + \beta \frac{m(p-1)}{p}\right)$的复杂度在大规模场景下有效分担根进程负载，而非阻塞线性算法通过通信重叠技术提升了中等规模应用的性能。Gather 操作包含已实现的 3 种核心算法和 4 种规划中的算法，线性同步算法通过两阶段数据传输确保了不可靠网络环境下的数据完整性，体现了可靠性与性能的权衡设计。

Allgather 操作提供了最丰富的 8 种算法实现，从递归加倍的$O(\alpha \log_2(p) + \beta m'(p-1))$延迟优化到环形算法的完美负载均衡，满足了无根进程全收集的多样化性能需求。Reduce 操作通过 10 种

算法实现了数据收集与运算融合的优化，通用树形算法提供了灵活的框架支持，有序二叉树算法专门处理非交换操作的运算顺序正确性，分散-聚集算法在大规模系统中展现出优异的扩展性。

在性能优化策略方面，分析发现 Open MPI 在算法设计中普遍采用了延迟-带宽权衡机制，通过 radix、segsize 等参数实现算法的性能调优；运用非阻塞通信和流水线处理技术提升带宽利用率；通过树形结构避免根进程瓶颈，环形算法实现完美负载分布；并关注数据局部性优化，如 Sparbit 算法等新兴实现注重缓存友好的数据访问模式。这些优化策略的综合运用使得 Open MPI 能够在不同网络环境和应用场景下提供高效的集合通信服务。

通过源码分析，本章系统梳理了集合通信算法的复杂度特征、适用场景和参数影响，为后续的性能建模和算法选择优化提供了完整的理论基础。同时通过阅读源码深化了对 Open MPI 集合通信实现的理解。

# 章节 4. 集合通信参数配置分析

基于 Open MPI 4.1.2 版本[15] 的官方文档 Open MPI v4.1.x Documentation，本章较全面的分析集合通信操作的参数配置体系，重点关注影响算法选择和性能优化的关键参数。

## 4.1. MCA 参数体系概述

Open MPI 通过模块化组件架构（MCA）提供了丰富的运行时参数配置能力。集合通信相关的参数主要分布在以下组件中：

- `coll` 组件参数：控制集合通信算法选择和行为
- `btl` 组件参数：影响底层传输性能
- `pml` 组件参数：控制点对点消息传递层
- 通用 MPI 参数：影响整体通信行为

### 4.1.1. 参数查询与设置方法

Open MPI 4.1 版本提供多种参数配置方式：

```
# 查询所有coll组件参数
ompi_info --param coll all
# 查询特定参数详细信息
ompi_info --param coll tuned --parsable
# 运行时设置参数（环境变量方式）
export OMPI_MCA_coll_tuned_use_dynamic_rules=1
mpirun -np 8 ./my_program
# 运行时设置参数（命令行方式）
mpirun -np 8 --mca coll_tuned_use_dynamic_rules 1 ./my_program
# 通过配置文件设置
```

[15] 与后续实验环境保持一致

```
echo "coll_tuned_use_dynamic_rules = 1" >> ~/.openmpi/mca-params.conf
```

代码 4.35 MCA 参数查询与设置方法

## 4.2. 算法选择参数

### 4.2.1. 组件优先级参数

Open MPI 通过组件优先级控制算法选择策略：

| 参数名称 | 默认值 | 范围 | 功能说明 |
|---|---|---|---|
| coll_tuned_priority | 30 | 0-100 | tuned 组件优先级，提供优化算法 |
| coll_basic_priority | 10 | 0-100 | basic 组件优先级，基础算法实现 |
| coll_libnbc_priority | 10 | 0-100 | 非阻塞集合通信组件优先级 |
| coll_sync_priority | 50 | 0-100 | 同步集合通信组件优先级 |

表 4.1：集合通信组件优先级参数（v4.1）

### 4.2.2. 动态规则控制参数

Open MPI 4.1 的 tuned 组件支持基于消息大小和进程数的动态算法选择：

| 参数名称 | 默认值 | 功能说明 |
|---|---|---|
| coll_tuned_use_dynamic_rules | 1 | 启用动态规则，根据运行时条件选择算法 |
| coll_tuned_dynamic_rules_filename | 空 | 指定自定义决策规则文件路径 |
| coll_tuned_init_tree_fanout | 2 | 初始化时树形算法的默认扇出 |
| coll_tuned_init_chain_fanout | 4 | 初始化时链式算法的默认扇出 |

表 4.2：动态规则控制参数

### 4.2.3. 强制算法选择参数

可以强制选择特定算法：

| 参数名称 | 示例值 | 功能说明 |
|---|---|---|
| coll_tuned_bcast_algorithm | 0-6 | 强制 broadcast 使用特定算法 |
| coll_tuned_reduce_algorithm | 0-6 | 强制 reduce 使用特定算法 |
| coll_tuned_allreduce_algorithm | 0-5 | 强制 allreduce 使用特定算法 |
| coll_tuned_gather_algorithm | 0-3 | 强制 gather 使用特定算法 |
| coll_tuned_scatter_algorithm | 0-3 | 强制 scatter 使用特定算法 |
| coll_tuned_allgather_algorithm | 0-8 | 强制 allgather 使用特定算法 |

表 4.3：强制算法选择参数

## 4.3. Broadcast 算法参数配置

基于 4.1 版本文档，broadcast 操作的算法 ID 映射：

| 算法 ID | 算法名称 | 适用场景 |
|---|---|---|
| 0 | 决策函数自动选择 | 默认模式，根据消息大小和进程数自动优化 |
| 1 | basic_linear | 小规模通信子或回退选择 |
| 2 | bintree | 中等规模，平衡延迟和带宽 |
| 3 | binomial | 大规模通信子，延迟敏感 |
| 4 | pipeline | 大消息，需要流水线重叠 |
| 5 | split_bintree | 特定拓扑优化的分裂二叉树 |
| 6 | knomial | 可调节 fanout 的 k 进制树 |

表 4.4：Broadcast 算法 ID 映射（v4.1）

### 4.3.1. Broadcast 性能调优参数

| 参数名称 | 默认值 | 单位 | 功能说明 |
|---|---|---|---|
| `coll_tuned_bcast_tree_fanout` | 2 | 无 | 树形算法的分支因子 |
| `coll_tuned_bcast_chain_fanout` | 4 | 无 | 链式算法的并行链数 |
| `coll_tuned_bcast_segment_size` | 0 | 字节 | 消息分段大小，0 表示不分段 |
| `coll_tuned_bcast_max_requests` | 0 | 个数 | 最大未完成请求数 |

表 4.5：Broadcast 性能调优参数

## 4.4. Reduce 算法参数配置

### 4.4.1. Reduce 算法选择

| 算法 ID | 算法名称 | 适用场景 |
|---|---|---|
| 0 | 决策函数自动选择 | 默认模式，智能算法选择 |
| 1 | linear | 小规模通信子，实现简单 |
| 2 | binomial | 大规模通信子，树形归约 |
| 3 | in_order_binomial | 非交换操作，保证运算顺序 |
| 4 | rabenseifner | 大消息归约，分散-聚集策略 |
| 5 | knomial | 可调节的 k 进制树归约 |

表 4.6：Reduce 算法 ID 映射（v4.1）

### 4.4.2. Reduce 性能调优参数

| 参数名称 | 默认值 | 单位 | 功能说明 |
|---|---|---|---|
| coll_tuned_reduce_tree_fanout | 2 | 无 | 树形归约的分支因子 |
| coll_tuned_reduce_chain_fanout | 4 | 无 | 链式归约的扇出参数 |
| coll_tuned_reduce_segment_size | 0 | 字节 | 消息分段大小 |
| coll_tuned_reduce_crossover | 4096 | 字节 | 算法切换的消息大小阈值 |

表 4.7：Reduce 性能调优参数

## 4.5. Allgather 算法参数配置

### 4.5.1. Allgather 算法选择

Allgather 在 4.1 版本中提供了较丰富的算法选择：

| 算法 ID | 算法名称 | 适用场景 |
|---|---|---|
| 0 | 决策函数自动选择 | 默认智能选择 |
| 1 | basic_linear | 小规模，线性收集后广播 |
| 2 | bruck | 通用 Bruck 算法，支持任意进程数 |
| 3 | recursive_doubling | $2^n$ 进程数，延迟最优 |
| 4 | ring | 环形算法，完美负载均衡 |
| 5 | neighbor_exchange | 偶数进程，邻居交换 |
| 6 | two_proc | 两进程专用优化 |
| 7 | sparbit | 数据局部性感知算法 |
| 8 | k_bruck | 可调基数的扩展 Bruck 算法 |

表 4.8：Allgather 算法 ID 映射（v4.1）

### 4.5.2. Allgather 性能调优参数

| 参数名称 | 默认值 | 单位 | 功能说明 |
|---|---|---|---|
| coll_tuned_allgather_bruck_radix | 2 | 无 | Bruck 算法的基数 |
| coll_tuned_allgather_max_requests | 0 | 个数 | 最大并发请求数 |
| coll_tuned_allgather_short_msg_size | 81920 | 字节 | 短消息阈值 |
| coll_tuned_allgather_long_msg_size | 524288 | 字节 | 长消息阈值 |

表 4.9：Allgather 性能调优参数

## 4.6. 决策函数阈值参数

Open MPI 4.1 使用基于消息大小和通信子大小的决策函数：

| 参数名称 | 默认值 | 单位 | 功能说明 |
|---|---|---|---|
| coll_tuned_bcast_small_msg | 12288 | 字节 | Broadcast 小消息阈值 |
| coll_tuned_bcast_large_msg | 524288 | 字节 | Broadcast 大消息阈值 |
| coll_tuned_reduce_crossover_msg_size | 4096 | 字节 | Reduce 算法切换阈值 |
| coll_tuned_scatter_small_msg | 2048 | 字节 | Scatter 小消息阈值 |
| coll_tuned_gather_small_msg | 2048 | 字节 | Gather 小消息阈值 |

表 4.10：决策函数阈值参数

## 4.7. 调试与监控参数

### 4.7.1. 详细日志配置

| 参数名称 | 推荐值 | 输出信息 |
|---|---|---|
| coll_base_verbose | 100 | 显示基础组件选择和算法执行过程 |
| coll_tuned_verbose | 100 | 显示 tuned 组件的决策过程和算法选择 |
| mpi_show_mca_params | coll,btl | 显示集合通信和传输层相关参数 |
| coll_tuned_dynamic_rules_verbose | 1 | 显示动态规则的匹配和应用过程 |

表 4.11：调试与监控参数

### 4.7.2. 性能分析工具

Open MPI 4.1 集成了多种性能分析工具：

```
# 使用内置的决策规则分析工具
ompi_info --param coll tuned --level 9

# 生成性能决策报告
export OMPI_MCA_coll_tuned_dynamic_rules_verbose=1
export OMPI_MCA_coll_tuned_use_dynamic_rules=1
mpirun -np 8 ./benchmark 2>&1 | grep "coll:tuned"

# 使用MPI_T接口监控算法选择
export OMPI_MCA_mpi_show_mpi_alloc_mem_leaks=1
mpirun -np 8 ./mpi_t_monitoring_tool
```

代码 4.36 性能分析工具使用示例

## 4.8. 参数优化策略

### 4.8.1. 基于应用特征的优化

| 应用场景 | 关键参数 | 推荐配置（v4.1） |
| --- | --- | --- |
| 延迟敏感应用 | 算法选择 | coll_tuned_bcast_algorithm=3 |
| | 树形结构 | coll_tuned_bcast_tree_fanout=4 |
| | 消息阈值 | coll_tuned_bcast_small_msg=4096 |
| 带宽密集应用 | 算法选择 | coll_tuned_allgather_algorithm=3 |
| | 分段大小 | coll_tuned_bcast_segment_size=65536 |
| | 并发控制 | coll_tuned_allgather_max_requests=8 |
| 大规模并行 | 树形拓扑 | coll_tuned_bcast_tree_fanout=8 |
| | 动态规则 | coll_tuned_use_dynamic_rules=1 |
| | 组件优先级 | coll_tuned_priority=40 |
| 内存受限环境 | 分段控制 | coll_tuned_bcast_segment_size=16384 |
| | 请求限制 | coll_tuned_bcast_max_requests=4 |
| | 算法回退 | coll_basic_priority=30 |

表 4.12：基于应用场景的参数优化策略

### 4.8.2. 动态规则文件配置

Open MPI 4.1 支持通过自定义决策规则文件来精确控制不同场景下的算法选择，这是实现性能优化的重要机制。

规则文件采用固定的格式，每行定义一个优化规则：

```
operation comm_size msg_start msg_end algorithm_id fanout segment_size
max_requests
```

各字段含义如下：

- **operation**: 集合通信操作类型（bcast、reduce、allgather 等）
- **comm_size**: 通信器大小（进程数）
- **msg_start/msg_end**: 消息大小范围（字节）
- **algorithm_id**: 算法编号（0=auto, 1=linear, 2=bintree, 3=binomial, 4=pipeline, 5=split_bintree, 6=knomial）
- **fanout**: 树形算法的扇出度
- **segment_size**: 流水线算法的分段大小
- **max_requests**: 最大并发请求数

```
# 格式: operation comm_size msg_start msg_end algorithm fanout segment max_requests
# 4进程Broadcast优化规则
bcast 4 0 1024 3 2 0 0           # 小消息: binomial算法, fanout=2 (46.18μs)
bcast 4 1024 262144 4 0 32768 4   # 大消息: pipeline算法, 32KB分段 (48.32μs)
bcast 4 262144 999999999 4 0 32768 8  # 超大消息: pipeline优化配置
```

```
# 8进程Broadcast规则
bcast 8 0 2048 3 2 0 0           # 小消息：binomial，fanout=2
bcast 8 2048 131072 4 0 16384 4   # 中消息：pipeline，16KB分段
bcast 8 131072 999999999 6 4 65536 8  # 大消息：knomial，fanout=4
```

代码 4.37 自定义决策规则文件示例

可使用环境变量指定规则文件并运行程序或直接用 mca 参数指定：

```
# 设置动态规则文件
export OMPI_MCA_coll_tuned_dynamic_rules_filename=`pwd`/optimized_rules.conf

# 运行应用程序
mpirun -np 4 ./a.out

# 或者直接在命令行中指定
mpirun -np 4 --mca coll_tuned_dynamic_rules_filename ./optimized_rules.conf ./
a.out
```

## 4.9. 参数验证与性能测试

结合上述梳理，下面给出在实际应用中，选择合适的集合通信算法和参数的方法。Open MPI 提供了多种方式来指定和验证算法选择。

### 4.9.1. 算法指定方法

Open MPI 支持通过 MCA 参数直接指定特定的集合通信算法：

```
# 方法1：命令行参数指定
mpirun --mca coll_tuned_bcast_algorithm 3 -np 4 ./program

# 方法2：环境变量设置
export OMPI_MCA_coll_tuned_bcast_algorithm=3
mpirun -np 4 ./program

# 方法3：参数文件配置
echo "coll_tuned_bcast_algorithm = 3" > mpi_params.conf
mpirun --mca-param-file mpi_params.conf -np 4 ./program
```

其中常用的算法编号包括

- 0: 自动选择（默认）
- 1: linear（线性算法）
- 2: bintree（二叉树）
- 3: binomial（二项式树）
- 4: pipeline（流水线）
- 5: split_bintree（分割二叉树）
- 6: knomial（k 叉树）

#### 4.9.2. 性能验证示例

通过直接指定算法，可以测得不同算法在特定场景下的性能表现，此处以`MPI_Bcast`作为示例：

```
# 测试binomial算法（通常适合小消息）
mpirun --mca coll_tuned_bcast_algorithm 3 \
       --mca coll_tuned_bcast_algorithm_fanout 2 \
       -np 4 ./bcast_test
# 测试pipeline算法（通常适合大消息）
mpirun --mca coll_tuned_bcast_algorithm 4 \
       --mca coll_tuned_bcast_algorithm_segmentsize 32768 \
       -np 4 ./bcast_test
```

基于我们的测试结果，4 进程环境下的最优配置为：

- 小消息（≤1KB）：binomial 算法，fanout=2，延迟约 46μs
- 大消息（>1KB）：pipeline 算法，32KB 分段，延迟约 48μs

同时可以通过 verbose 模式验证配置的正确性[16]。

### 4.10. 小结

本章基于Open MPI 4.1.2版本简要分析了集合通信参数配置的方式，为后续的数据集构建和机器学习建模提供了完整的参数空间基础；并梳理了动态规则文件配置的标准格式和决策逻辑，为后续模型训练和标签生成提供技术支持。

## 章节 5. 数据集构建与应用场景设计

本章基于前述集合通信参数配置分析，进行建模框架设计和数据集构建。

### 5.1. 建模框架设计

#### 5.1.1. 问题抽象和建模

集合通信算法优化问题可抽象为一个多目标参数优化问题。给定通信环境特征，寻找最优算法配置以最小化通信延迟并最大化带宽利用率。

在本次研究进行了以下简化：

- 忽略了系统环境的影响，仅仅将通信操作、进程数、消息大小作为输入特征；
- 仅仅以最小化程序运行时间作为优化目标。

具体建模如下。

[16]实验过程中已通过性能表现对上述提及的算法进行初步验证，未对该部分进行更深入的探究，此处略过。

设通信环境特征向量为 $\boldsymbol{x} = (n, m, \text{op})$，其中：

- $n \in \mathcal{N} = \{2, 4, 8, 16, 32\}$ 表示进程数
- $m \in \mathcal{M} = \{2^6, 2^8, 2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$ 表示消息大小（字节）
- $\text{op} \in \mathcal{O} = \{\text{bcast}, \text{scatter}, \text{gather}, \text{allgather}, \text{reduce}\}$ 表示通信操作类型。

对于每种通信操作 $\text{op}_i$，定义其算法配置空间 $\Theta_{\text{op}_i}$：

$$\Theta_{\text{op}_i} = \mathcal{A}_{\text{op}_i} \times \mathcal{P}_{\text{op}_i} \tag{5.1}$$

其中 $\mathcal{A}_{\text{op}_i}$ 为算法选择空间，$\mathcal{P}_{\text{op}_i}$ 为参数配置空间。

基于章节 3. Open MPI 集合通信算法源码分析和章节 4. 集合通信参数配置分析，整理得下表中 Open MPI 4.1.2 各通信操作完整的算法空间及其对应的参数空间[17]：

| 操作 | 算法 ID | 算法名称 | 参数 1 | 参数 2 | 参数 3 |
|---|---|---|---|---|---|
| **Broadcast** | 1 | basic_linear | - | - | - |
| | 2 | bintree | fanout: [2,4,8] | - | - |
| | 3 | binomial | fanout: [2,4,8] | - | - |
| | 4 | pipeline | - | segment: [0,8K,32K,131K] | max_req: [0,4,8,16] |
| | 5 | split_bintree | fanout: [2,4,8] | segment: [0,8K,32K] | - |
| | 6 | knomial | fanout: [2,4,8] | - | - |
| **Reduce** | 1 | linear | - | - | - |
| | 2 | binomial | fanout: [2,4,8] | segment: [0,8K,32K] | chain_fanout: [2,4,8] |
| | 3 | in_order_binomial | fanout: [2,4,8] | segment: [0,8K,32K] | chain_fanout: [2,4,8] |
| | 4 | rabenseifner | fanout: [2,4,8] | segment: [0,8K,32K] | chain_fanout: [2,4,8] |
| | 5 | knomial | fanout: [2,4,8] | segment: [0,8K,32K] | chain_fanout: [2,4,8] |
| **Allgather** | 1 | basic_linear | - | - | - |
| | 2 | bruck | bruck_radix: [2,4,8] | - | - |
| | 3 | recursive_doubling | - | - | - |
| | 4 | ring | - | max_req: [0,4,8,16] | ring_seg: [0,1] |

[17] 由于 ID = 0 对应 auto 策略，无法明确所使用的算法，因此在下面的算法空间中去除 ID = 0 的选项。

| 操作 | 算法 ID | 算法名称 | 参数 1 | 参数 2 | 参数 3 |
|---|---|---|---|---|---|
| | 5 | neighbor_exchange | - | - | - |
| | 6 | two_proc | - | - | - |
| | 7 | sparbit | bruck_radix: [2,4,8] | - | - |
| | 8 | k_bruck | bruck_radix: [2,4,8] | - | - |
| Scatter | 1 | linear | - | - | - |
| | 2 | binomial | fanout: [2,4,8] | segment: [0,8K,32K] | - |
| | 3 | bintree | fanout: [2,4,8] | segment: [0,8K,32K] | - |
| Gather | 1 | linear | - | - | - |
| | 2 | binomial | fanout: [2,4,8] | segment: [0,8K,32K] | - |
| | 3 | bintree | fanout: [2,4,8] | segment: [0,8K,32K] | - |

表 4.13：集合通信算法空间与参数空间

上表中的参数缩写含义如下：

- **fanout**: `coll_tuned_[op]_tree_fanout` - 树形算法的分支因子
- **segment**: `coll_tuned_[op]_segment_size` - 消息分段大小（字节）
- **max_req**: `coll_tuned_[op]_max_requests` - 最大未完成请求数
- **chain_fanout**: `coll_tuned_reduce_chain_fanout` - 链式归约的扇出参数
- **bruck_radix**: `coll_tuned_allgather_bruck_radix` - Bruck 算法的基数
- **ring_seg**: `coll_tuned_allgather_ring_segmentation` - 环形算法分段开关

其中消息分段大小的具体数值为：

- 0: 不分段
- 8K: 8192 字节
- 32K: 32768 字节
- 131K: 131072 字节

### 5.1.2. 特征空间定义

基于上述建模框架，定义输入特征空间和输出标签空间如下。

#### 5.1.2.1. 输入特征向量

输入特征向量 $\boldsymbol{x} = (n, m, \text{op})$ 的具体编码方式如表 5.1 所示：

| 特征维度 | 取值范围 | 编码方式 | 说明 |
| --- | --- | --- | --- |
| 进程数 $n$ | $\mathcal{N}=\{2,4,8,16,32\}$ | 整数编码 | 直接使用数值表示通信器大小 |
| 消息大小 $m$ | $\mathcal{M}=\{2^6,2^8,\ldots,2^{20}\}$ | 对数编码 | 使用 $\log_2(m)$ 并归一化到 $[6,20]$ |
| 操作类型 op | $\mathcal{O}=\{\text{bcast},\text{scatter},\ldots\}$ | 独热编码 | 5 维二进制码表示操作类型 |

表 5.1 输入特征编码规范

具体而言，特征向量的编码规则为：

进程数特征：$n_{\text{encoded}}=n$，直接使用原始数值。

消息大小特征：$m_{\text{encoded}}=\frac{\log_2(m)-6}{14}$，将对数值归一化到 $[0,1]$ 区间。

操作类型特征：使用独热编码 $\text{op}_{\text{encoded}}=(o_1,o_2,o_3,o_4,o_5)$，其中：

- $o_1=1$ 表示 broadcast，否则为 0
- $o_2=1$ 表示 scatter，否则为 0
- $o_3=1$ 表示 gather，否则为 0
- $o_4=1$ 表示 allgather，否则为 0
- $o_5=1$ 表示 reduce，否则为 0

**5.1.2.2.** 输出标签空间

根据表 4.13 中的算法空间分析，每种通信操作的算法配置空间大小如表 5.2 所示：

| 操作 | 算法数 | 有参算法 | 配置总数 | 主要变化维度 |
| --- | --- | --- | --- | --- |
| Broadcast | 6 | 3 | 2×3 + 1×3×4×4 + 1×3×3 + 1×3 = 75 | fanout, segment_size, max_requests |
| Reduce | 5 | 4 | 4×3×3×3 = 108 | fanout, segment_size, chain_fanout |
| Allgather | 8 | 4 | 2×3 + 1×4×2 + 2×1 = 16 | bruck_radix, max_requests, ring_seg |
| Scatter | 3 | 2 | 2×3×3 = 18 | fanout, segment_size |
| Gather | 3 | 2 | 2×3×3 = 18 | fanout, segment_size |
| 总计 | 25 | 15 | **235** | - |

表 5.2 各操作算法配置空间规模

由于不同操作具有不同的算法配置空间，本研究采用分层标签体系：

第一层：算法选择标签 对于操作 $\text{op}_i$，算法选择标签为：

$$y_{\text{alg}}\in\left\{1,2,\ldots,|\mathcal{A}_{\text{op}_i}|\right\} \tag{5.2}$$

第二层：参数配置标签 对于选定算法，参数配置向量为：

$$\boldsymbol{y}_{\text{param}} = (p_1, p_2, p_3) \tag{5.3}$$

其中各参数的具体含义依据算法类型确定。

#### 5.1.2.3. 完整特征-标签映射

最终的机器学习问题可表述为：

给定输入特征 $\boldsymbol{x} = (n_{\text{encoded}}, m_{\text{encoded}}, \text{op}_{\text{encoded}})$，预测最优算法配置：

$$(y_{\text{alg}}, \boldsymbol{y}_{\text{param}}) = f_{\text{ML}}(\boldsymbol{x}) \tag{5.4}$$

其中 $f_{\text{ML}}$ 为待训练的机器学习模型，目标是最小化预测配置的执行时间。

## 章节 6. 机器学习建模与性能预测

## 章节 7. 模型验证与检验

## 章节 8. 结果分析与总结

## 章节 9. 结论与展望

# 参考文献