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 14 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。

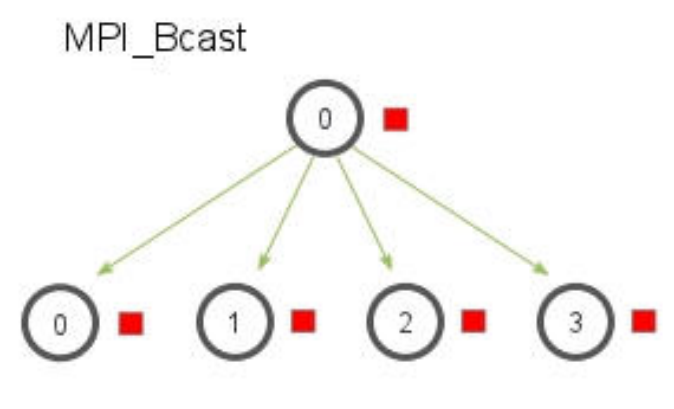


图 3.1 MPI_Bcast 通信模式图示

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：

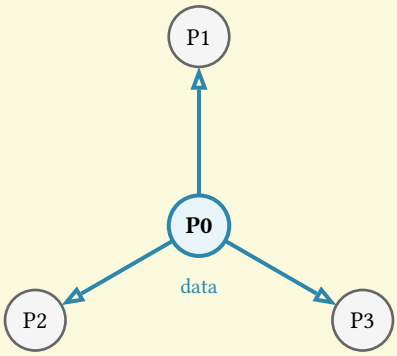


图 3.2 Broadcast 线性算法图示

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 α（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 β（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

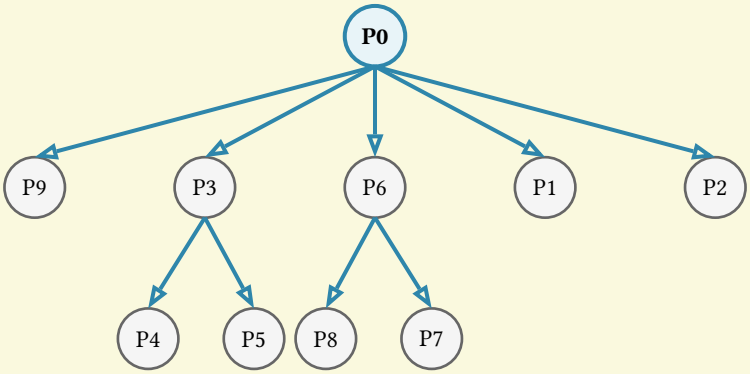


图 3.3 K 项树 Broadcast 算法的树形结构（radix=3）

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`

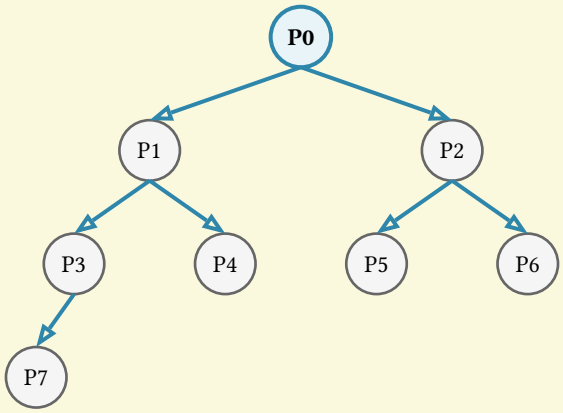
源码文件路径：ompi/mca/coll/base/coll_base_bcast.c

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：

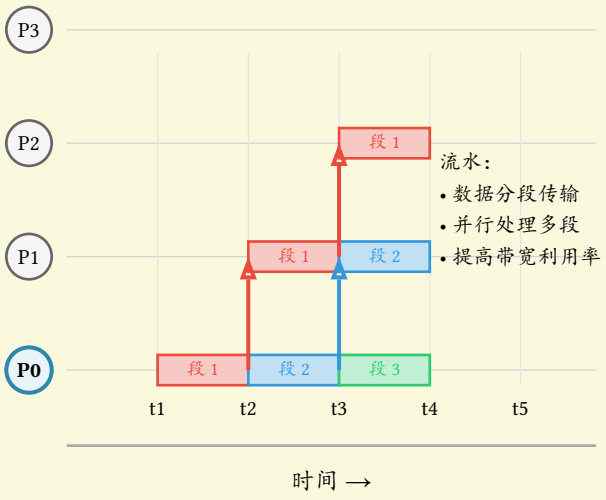


图 3.5 流水线 Broadcast 算法图示

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。

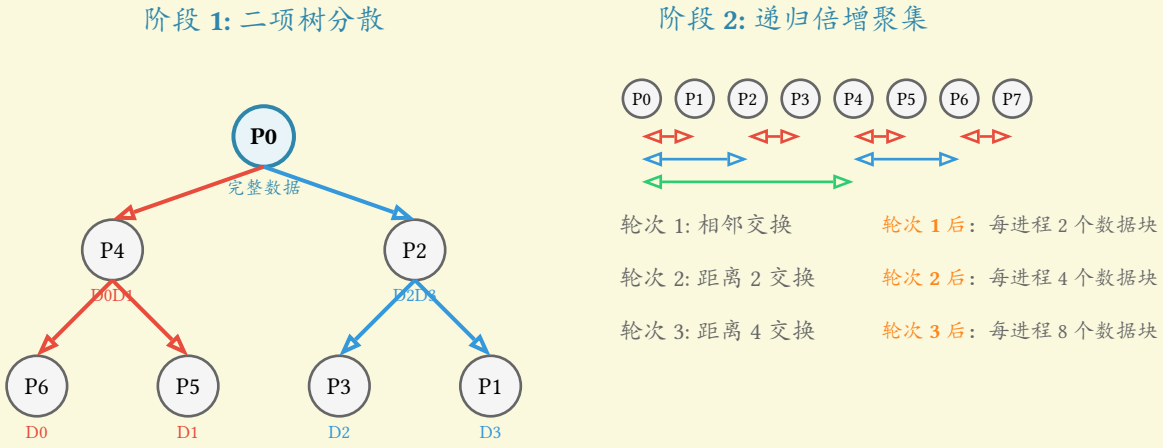
其主要原理是：采用两阶段策略，第一阶段使用二项树将数据分散到各进程（Scatter），第二阶段使用递归倍增算法进行全聚集（Allgather），重构完整数据。

```c
/*
 * 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 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 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 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 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 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
- α: 通信延迟参数，β: 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（≥ 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。

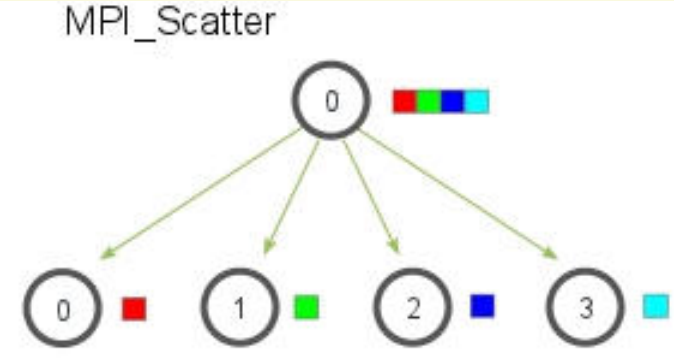


图 3.7 MPI_Scatter 通信模式图示

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。

```
int ompi_coll_base_scatter_intra_linear_nb(
    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 max_reqs)
{
    int i, rank, size, err, nreqs;
    ompi_request_t **reqs = NULL, **preq;

    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 MPI_SUCCESS;
    }

    // 计算请求数量和分配请求数组
    if (max_reqs <= 1) {
        nreqs = size - 1;  // 除自己外的所有进程
    } else {
        // 周期性使用阻塞发送，减少请求数量
        nreqs = size - (size / max_reqs);
    }

    reqs = ompi_coll_base_comm_get_reqs(module->base_data, nreqs);

    // 根进程：循环发送数据
    for (i = 0, ptmp = (char *)sbuf, preq = reqs; i < size; ++i, ptmp +=
incr) {
        if (i == rank) {
            // 本地复制
            if (MPI_IN_PLACE != rbuf) {
                err = ompi_datatype_sndrcv(ptmp, scount, sdtype,
                                           rbuf, rcount, rdtype);
            }
        } else {
            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 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 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 通信-计算重叠
链式散射算法	`ompi_coll_base_scatter_intra_chain`	`segsize` `chains`	$O\left(\frac{p}{\text{chains}} \cdot \alpha + \beta m\right)$	特定网络拓扑 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 流水线重叠
通用树形算法	`ompi_coll_base_scatter_intra_generic`	`tree` `segcount`	取决于树结构	通用框架 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 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
- α: 通信延迟参数，β: 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。

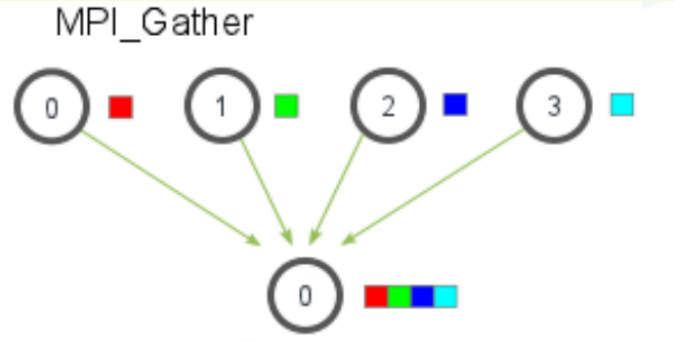


图 3.8　MPI_Gather 通信模式图示

Open MPI 为 Gather 操作提供了多种算法实现：

3.4.4. 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。每个进程都能获得完整的聚集结果。

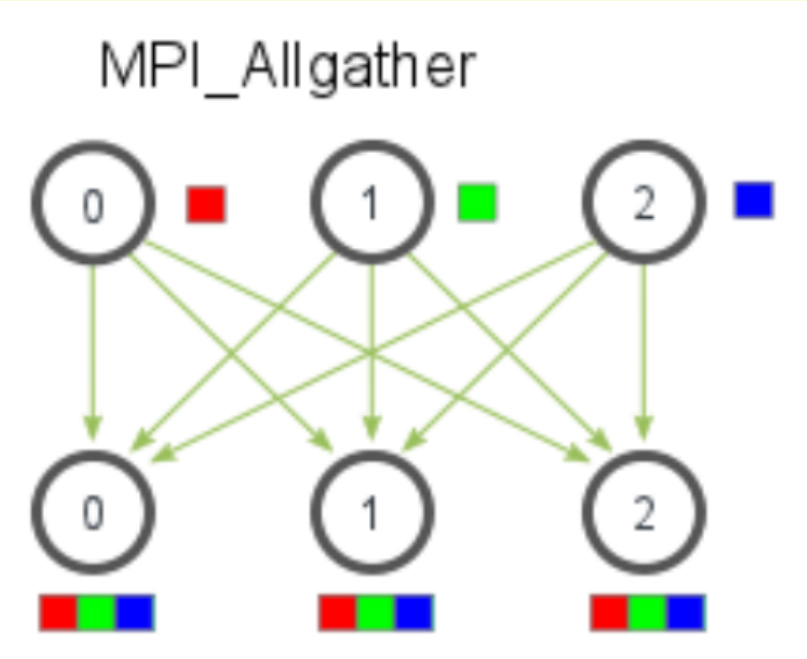


图 3.9 MPI_Allgather 通信模式图示

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` 参数是一组每个进程准备规约的数据，其类型为 `datatype`。`recv_data` 只与具有根 `rank` 的进程相关。`recv_data` 数组包含规约的结果并且其大小为 `sizeof(datatype) * count`。`op` 参数是对数据应用的操作。

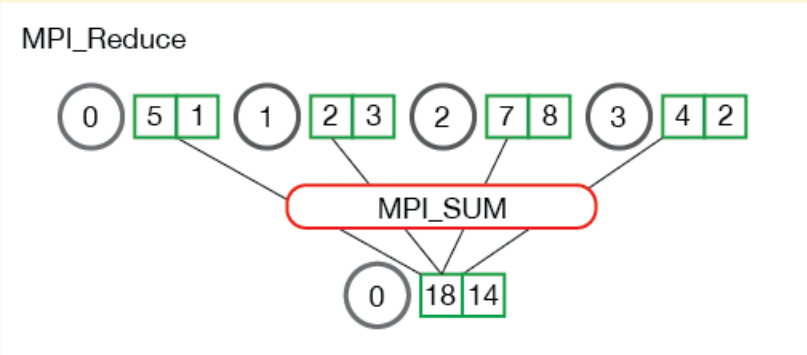


图 3.10 MPI_Reduce 通信模式图示

主要有如下算法实现。

3.4.5.1. 线性算法（Linear Algorithm）

函数：`ompi_coll_base_reduce_intra_basic_linear()`

源码文件路径：

其主要原理是：

图示如下：

算法复杂度分析：

3.4.5.2. 超立方体算法

章节 4. 集合通信参数配置分析

章节 5. 数据集构建与应用场景设计

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