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 09 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，如二进制树(binary tree)、二项式树(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,
8   &ompi_mpi_instance_default,
9                               argc, argv);
10
11      // 3. 初始化通信子子系统
12      if (OMPI_SUCCESS != (ret = ompi_comm_init_mpi3())) {
13          error = "ompi_mpi_init: ompi_comm_init_mpi3 failed";
14          goto error;
15      }
16
17      // 4. 为预定义通信子选择集合通信组件
18      if (OMPI_SUCCESS != (ret = mca_coll_base_comm_select(MPI_COMM_WORLD))) {
19          error = "mca_coll_base_comm_select(MPI_COMM_WORLD) failed";
20          goto error;
21      }
22
23      if (OMPI_SUCCESS != (ret = mca_coll_base_comm_select(MPI_COMM_SELF))) {
24          error = "mca_coll_base_comm_select(MPI_COMM_SELF) failed";
25          goto error;
26      }
27
28      // 5. 标记初始化完成
29      opal_atomic_swap_32(&ompi_mpi_state, OMPI_MPI_STATE_INIT_COMPLETED);
30  }
```

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：

```
1   int mca_coll_base_comm_select(ompi_communicator_t *comm)
2   {
3       // 1. 初始化通信子的集合通信结构
4       comm->c_coll = (mca_coll_base_comm_coll_t*)calloc(1,
    sizeof(mca_coll_base_comm_coll_t));
5
6       // 2. 查询所有可用的集合通信组件（basic, tuned, han, xhc等）
7       selectable =
    check_components(&ompi_coll_base_framework.framework_components, comm);
8
9       // 3. 按优先级排序并启用最高优先级组件
10      for (item = opal_list_remove_first(selectable);
11           NULL != item;
12           item = opal_list_remove_first(selectable)) {
13
14          mca_coll_base_avail_coll_t *avail = (mca_coll_base_avail_coll_t *)
    item;
15          ret = avail->ac_module->coll_module_enable(avail->ac_module, comm);
16
17          if (OMPI_SUCCESS == ret) {
18              // 4. 设置函数指针到具体实现
19              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：

```c
// 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：

```c
// 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

```c
// 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进制树、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

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