Message-Passing and MPI Programming

More on Collectives

N.M. Maclaren

Computing Service

nmm1@cam.ac.uk
ext. 34761

July 2010

5.1 Introduction

There are two important facilities we have not covered yet; they are less commonly used, but are fairly often needed. In particular, one of them can help a lot with I/O.

There is also some information on how to use collectives efficiently, including one potentially useful MPI-2 feature.

5.2 Searching

You can use global reductions for searching. The bad news is that it needs MPI's derived datatypes, but the good news is that there are some useful built-in ones. All we need to do is a reduction with a composite datatype `(<value>,<index>)`. As with summation, we build up the reduction from a binary operator, and there are two built-in operators, `MPI_MINLOC` and `MPI_MAXLOC`. We shall use finding the minimum as an example.

We start with two values, `(value_1,index_1)` and `(value_2,index_2)`, and produce a result `(value_x,index_x)`. The algorithm for the binary operator is simply:

If `value_1` $\leq$ `value_2` then
    `value_x` $\Leftarrow$ `value_1`
    `index_x` $\Leftarrow$ `index_1`
Else
    `value_x` $\Leftarrow$ `value_2`
    `index_x` $\Leftarrow$ `index_2`

Equality is a bit cleverer, but it rarely matters; if it does matter to your program, see the MPI standard for the detailed specification. Operator `MPI_MINLOC` does precisely what we have described; operator `MPI_MAXLOC` searches for the maximum, and is the same with $\leq$ replaced by $\geq$.

Note that you create the `(value,index)` pairs first, and the the index can be anything, so set it to whatever is useful for your program. Generally, an index should usually be globally unique (i.e. not just the index into a local array). For example, you can combine the processor number and a local index. So how do we set up the data?

5.3 Fortran Searching

Fortran 77 does not have structures (though Fortran 90 does), so the datatypes are arrays of length two. The ones to use are MPI_2INTEGER and MPI_2DOUBLE_PRECISION. Note that DOUBLE PRECISION can hold any INTEGER value, on any current MPI system; in fact, it is almost required by the Fortran standard, and so is a safe assumption in almost all programs. MPI_2REAL is not recommended, except on *Cray* vector systems.

Using these for searching is very simple, as in the following example:

```
INTEGER :: sendbuf ( 2 , 100 ) ,    &
    recvbuf ( 2 , 100 ) , myrank , error , i
INTEGER, PARAMETER :: root = 3

CALL MPI_Comm_rank ( MPI_COMM_WORLD , myrank , error )
DO i = 1 , 100
sendbuf ( 1 , i ) = <value>
sendbuf ( 2 , i ) = 1000 * myrank + i
END DO

CALL MPI_Reduce ( sendbuf , recvbuf , 100, MPI_2INTEGER ,    &
    MPI_MINLOC , root , MPI_COMM_WORLD , error )
```

5.4 C and C++ Searching

These do have structures, so the datatypes are "struct {<value type>; int;}". Unfortunately, C/C++ structure layout is a can of worms, so certain compiler options may cause trouble. What is described here will usually work; if it does not, you will need to ask for help from an expert on both C and your compiler. Most probably, you will never encounter the problems if you follow these rules.

The recommended datatypes are MPI_2INT, MPI_LONG_INT and MPI_DOUBLE_INT, corresponding to <value type>s of int, long and double. You can also use MPI_LONG_DOUBLE_INT for "long double", if you use that. Do not use MPI_FLOAT_INT or SHORT_INT for C/C++ reasons you do **not** want to know!

C:

```
struct { double value ; int index ; }
    sendbuf [100] , recvbuf [100] ;
int root = 3, myrank , error , i;

for ( i = 1 ; i < 100 ; ++i ) {
    sendbuf [ i ] . value = <value> ;
    sendbuf [ i ] . index = 1000 * myrank + i ;
}

error = MPI_Reduce ( sendbuf , recvbuf ,
    100, MPI_DOUBLE_INT , MPI_MINLOC ,
    root , MPI_COMM_WORLD )
```

C++:

```
struct { double value ; int index ; }
    sendbuf [100] , recvbuf [100] ;
int root = 3, myrank , error , i;

for ( i = 1 ; i < 100 ; ++i ) {
    sendbuf [ i ] . value = <value> ;
    sendbuf [ i ] . index = 1000 * myrank + i ;
}

MPI::COMM_WORLD . Reduce ( sendbuf , recvbuf ,
    100, MPI::DOUBLE_INT , MPI::MINLOC , root )
```

5.5 Data Distribution

It can be inconvenient to make all counts the same, such as with a 100×100 matrix on 16 CPUs. One approach is to pad the short vectors, and that is usually more efficient than it looks. There are also extended MPI collectives for handling non-identical counts (`MPI_Gatherv`, `MPI_Scatterv`, `MPI_Allgatherv` and `MPI_Alltoallv`) but, obviously, their interface is more complicated. You should use whichever approach is easiest for you.

The count argument is now a vector of counts instead of a single count, with one count for each process, but only where there are multiple buffers. That is the receive counts for `MPI_Gatherv` and `MPI_Allgatherv`, the send counts for `MPI_Scatterv` and both counts for `MPI_Alltoallv`. It is used only on the root process for `MPI_Gatherv` and `MPI_Scatterv`. For `MPI_Allgatherv`, the count vectors must match on all processes. `MPI_Alltoallv` is described in a moment. The most reliable approach is always to make them match, for sanity; if you then change the root process, nothing will break.

The scalar counts may all be different, of course, because they must match for each pairwise send and receive. For example:

- For `MPI_Gatherv`, the send count on process N matches the Nth receive count element on the root. `MPI_Scatterv` just the converse of `MPI_Gatherv`.

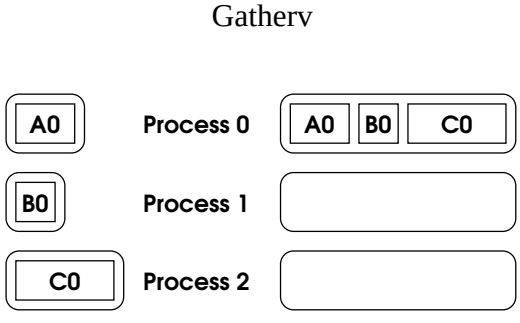

Figure 5.1

- For `MPI_Allgatherv`, the send count on process N matches Nth receive count element on **all** processes.

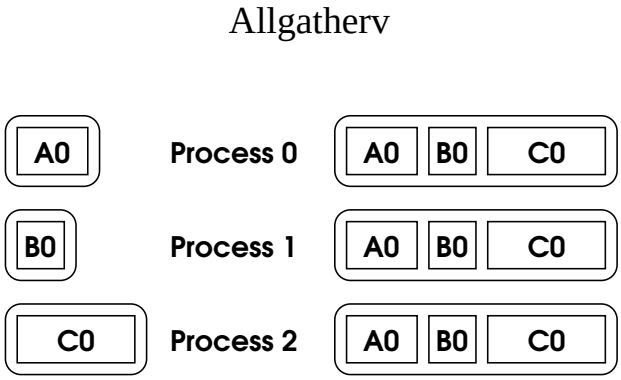

Figure 5.2

The most complicated one is `MPI_Alltoallv`, though it is not hard to use, if you keep a clear head. You should use a pencil and paper if you get confused, which is a good general rule for programming. Consider processes M and N:

- For `MPI_Alltoallv`, the Nth send count on process M matches the Mth receive count on process N.

As was said earlier, think of it as a matrix transpose with the data vectors as its elements.

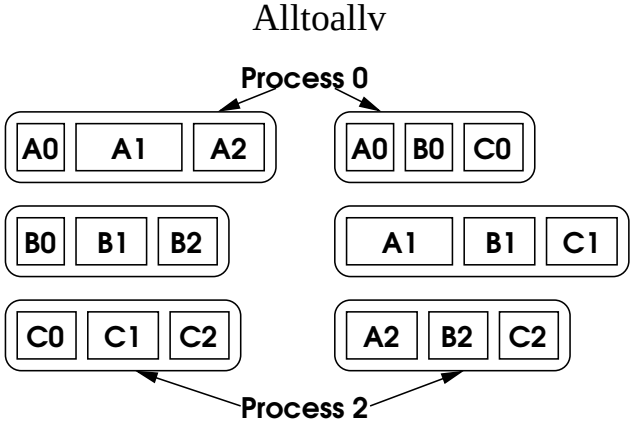

Figure 5.3

Where they have a vector of counts, they also have a vector of offsets. This is the offset of the sub-vector of data corresponding to each process, and **not** the offset of the basic elements. This allows for discontiguous buffers, but each pairwise transfer must still be contiguous. Normally, the first offset will be zero, But here is a picture of when it is not:

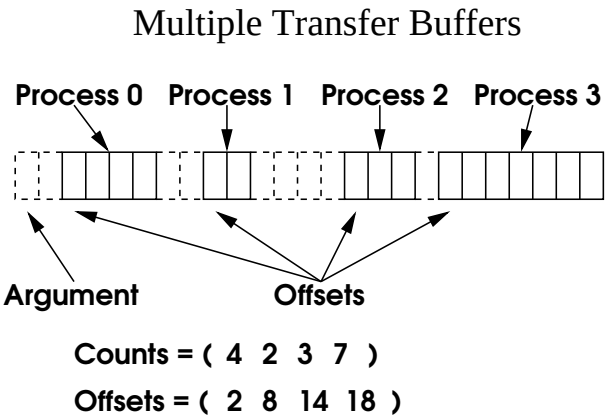

Figure 5.4

Unlike the counts, the offsets are purely local, and they need not match on all processes; even in the case of `MPI_Alltoallv`, the offset vectors need not match in any way. Each one is used just as a mapping for the local layout of its associated buffer.

Keep your use of these collectives simple; MPI will not get confused, but you and I will, and remember that any overlap is undefined behaviour. The following picture of a fairly general `MPI_Alltoallv` is just to see what can be done, and not to recommend such practice:

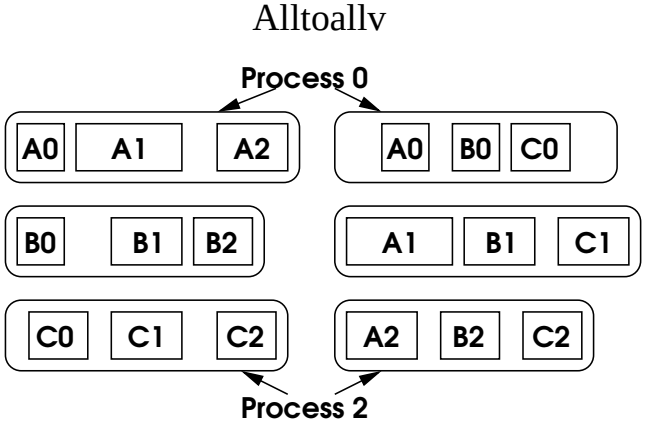

Figure 5.5

The examples are simple ones of using `MPI_Gatherv`; you should start with either this or `MPI_Scatterv` when testing.

Fortran:

```
INTEGER , DIMENSION ( 0 : * ) :: counts
REAL(KIND=KIND(0.0D0)) ::    &
    sendbuf ( 100 ) , recvbuf ( 100 , 30 )
INTEGER :: myrank , error, i
INTEGER , PARAMETER :: root = 3 ,    &
    offsets ( * ) = (/ ( 100 * i , i = 0 , 30 - 1 ) /)

CALL MPI_Gatherv (    &
    sendbuf , counts ( myrank ) , MPI_DOUBLE_PRECISION ,    &
    recvbuf , counts , offsets , MPI_DOUBLE_PRECISION ,    &
    root , MPI_COMM_WORLD , error )
```

C:

```
int counts [ ] ;
double sendbuf[100] , recvbuf[30][100] ;
int root = 3 , offsets[30] , error, i ;
for ( i = 0 ; i < 30 ; ++ i )
    offsets [ i ] = 100 * i ;

error = MPI_Gatherv (
    sendbuf , counts [ myrank ] , MPI_DOUBLE ,
    recvbuf , counts , offsets , MPI_DOUBLE ,
    root , MPI_COMM_WORLD ) ;
```

C++:

```
int counts [ ] ;
double sendbuf[100], recvbuf[30][100];
int root = 3 , offsets[30] , error, i ;
for ( i = 0 ; i < 30 ; ++ i )
    offsets [ i ] = 100 * i ;

MPI::COMM_WORLD . Gatherv (
    sendbuf , counts [ myrank ] , MPI::DOUBLE ,
    recvbuf , counts , offsets ,  MPI::DOUBLE ,
   root ) ;
```

Using these functions is fairly easy when the counts are predictable, but a lot of the time, they will not be. The easiest way to solve that problem is:

- For scatter, calculate the counts on the root process, use `MPI_Scatter` to distribute the count values, and then do the full `MPI_Scatterv` on the data.
- For gather, calculate a count on each process, use `MPI_Gather` to collect the count values, and then do the full `MPI_Gatherv` on the data.

5.6 Efficiency

Generally, you should use collectives whereever possible, because they provide most opportunity for the implementation to tune the calls. Similarly, you should use the composite ones (`MPI_Allgather`, `MPI_Allreduce` and `MPI_Alltoall`) where that makes sense, just as you should use the level 3 *BLAS* (`DGEMM` etc.) in preference to the level 2 ones (`DGEMV` etc.)

You should also do as much as possible in one collective – fewer, larger transfers are always better than a lot of small ones. Consider packing scalars into arrays for copying, even converting integers to reals (double precision, please!) to do so. It is not worth packing multiple large arrays into one, and a good rule of thumb is that the boundary between 'small' and 'large' is a single, pairwise transfer buffer of about 4 KB. But remember that it is only a rule of thumb, and do not waste time packing data unless you actually need to. If the code is not critical to performance, do not worry about it.

5.7 Using Barriers

The actual execution of collectives is not synchronised at all – except for `MPI_Barrier`, that is. Up to three successive collectives on the same communicator can overlap – in theory, this allows for improved efficiency. Note that this is taking the global viewpoint; from the viewpoint of a single process, collectives are executed sequentially. Programmers do not notice this in practice, except that it makes it hard to measure times. To synchronise, call `MPI_Barrier`; the first process leaves only after the last process enters, so all processes will run in lock-step.

Implementations usually tune for gang scheduling, and collectives often run faster when synchronised. Consider adding a barrier before every collective – it is an absolutely trivial change, after all. The best approach is to run 3–10 times with barriers, and 3–10 without; either one will be consistently faster than the other, or it will not matter which you choose. If you have major, non-systematic differences, then you have a nasty problem andmay need help from either or both of an MPI or operating tuning expert.

You can overlap collectives and point-to-point; MPI requires implementations to make that work, but it is strongly recommended to avoid doing that when performance is important. A correct program will definitely not hang or crash, but it may run horribly slowly. Remember that three collectives can overlap – point-to-point can interleave with those, and the scheduling can get very confused. It is much better to alternate the modes of use, and makes the program a lot easier to validate and debug, as well. For example:

> Start here ...
>
> [Consider calling `MPI_Barrier` here]
>
> Any number of collective calls
>
> [Consider calling `MPI_Barrier` here]
>
> Any number of point-to-point calls, but remember to wait for **all** of those calls to finish.
>
> And repeat from the beginning ...

5.8 In-Place Collectives

In MPI-1, using the same array twice will **usually** work, but it is a breach of the Fortran standard, and is not clearly permitted in C++ or even C (the latter two standards can be read either way). If you are seriously masochistic, you can ask for the reasons offline, but I recommend not bothering. It is much better to avoid doing this if at all possible. It will rarely cause trouble or get diagnosed – but, if it does, the bug will be almost unfindable. There have been systems on which it would fail, for good reasons – the Hitachi SR2201 was one, for example.

MPI-2 defines a `MPI_IN_PLACE` pseudo-buffer, which specifies the result overwrites the input (i.e. the real buffer is both source and target). You should read the MPI standard for its full specification; it is probably most useful for `MPI_allgather[v]`, `MPI_alltoall[v]` and `MPI_allreduce[v]`. In those cases, you use it for the **send** buffer on **all** processes, and the send counts, datatype and offsets are ignored. Here is a C example; the Fortran and C++ are very similar.

```
        double buffer [ 30 ] [ 100 ] ;
        int error ;
        error = MPI_Alltoall (
            MPI_IN_PLACE , 100 , MPI_DOUBLE ,
/* Or even 'MPI_IN_PLACE , 0 , MPI_INT ,'
            buffer , 100 , MPI_DOUBLE ,
            MPI_COMM_WORLD )
```

5.9 Epilogue

That is essentially **all** you need to know about collectives! We have covered everything that seems to be used; MPI collectives look very complicated, but are really quite simple. There are a few more features, which are rarely used, and were mainly introduced by MPI-2 for advanced uses. They are mentioned in the extra lectures, but only in passing. There are two exercises on searching and scatterv.