Compiler-Controlled Extraction of Computation-Communication Overlap in MPI Applications

Dibyendu Das
IBM India Systems Technology Lab
dibyendu.das@in.ibm.com

Manish Gupta
IBM India Systems Technology Lab
mgupta7@in.ibm.com

Rajan Ravindran
IBM India Systems Technology Lab
rrajan@in.ibm.com

W Shivani
NIT Warangal
shivani_w2002@yahoo.co.in

P Sivakeshava
NIT Warangal
sivakeshava@gmail.com

Rishabh Uppal
IIT Kanpur
rishabh@cse.iitk.ac.in

Abstract

Exploiting Computation-Communication Overlap is a well-known requirement to speed up distributed applications. However, efforts till now use programmer expertise, rather than any automatic tool to do this. In our work we propose the use of an aggressive optimizing compiler (IBM's xl series) to automatically extract opportunities for computation communication overlap. We depend on aggressive inlining, dominator trees and SSA based use-def analyses provided by the compiler framework for exploiting such overlap. Our target is MPI applications. In such applications, we try to automatically move mpi_waits as well as split blocking mpi_send/recv to create more opportunities for overlap. Our objective is two-fold: firstly, our tool should relieve the programmer from the burden of hunting for overlap manually as much as possible, and secondly, it should aid in converging on parallel applications which benefit from such overlap quickly. These are necessary as MPI applications are quickly becoming complex and huge and manual overlap extraction is becoming cumbersome. Our early experience shows that it is not necessary that exploiting an overlap always leads to performance improvement. This corroborates with the fact that if we have an automatic tool, then, we can quickly discard such applications (or certain configurations of such applications) without spending person-hours to manually rewrite MPI applications for introducing non-blocking calls. Our initial experiments with the industry-standard NAS Parallel benchmarks show that we can get small-to-moderate improvements by utilizing overlap even in such highly tuned benchmarks. This augurs well for real-world applications that do not exploit overlap optimally.

Categories and Subject Descriptors [**MPI, Optimizing Compilers**]

General Terms: Parallel Applications, MPI, Algorithms, Performance, Compiler

Keywords *Computation-Communication Overlap, Compiler Optimization*

1. Introduction

For better computation-communication overlap, MPI [7] provides non-blocking versions of mpi_send and mpi_recv, called mpi_isend and mpi_irecv. The intention is that the programmer will intelligently invoke computation in between an mpi_isend or mpi_irecv and its matching mpi_wait. However, such a job is easier said than done. While programmers may (or do) try to exploit this overlap, it may not be possible to do this in many circumstances as this is non-trivial. This results in missed chances or opportunities resulting in poor computation-communication overlap (CCO) and higher run time for the programs executed in many cases. In our work we propose compiler-driven techniques that target sub-optimally positioned mpi_wait(s). These are mpi_wait(s) which can (probably) be moved further ahead (temporally) in the computation resulting in better CCO. We term this mpi_wait sinking. We also target blocking mpi_send/mpi_recv which can be converted to a pair of mpi_isend/mpi_wait (or mpi_irecv and mpi_wait). Subsequently, techniques that are applicable to mpi_wait sinking can be applied to the newly created mpi_wait part of the non-blocking calls.

The compiler framework of the IBM xl series of compilers [11] is used in our work. The compiler provides advanced analyses and optimization, including aggressive and cross-file inlining (which is necessary for our work), SSA-based use-def analyses and region-based optimizations. Using the compiler framework, we devise two algorithms in our paper, one that conservatively sinks waits within an interval (or a loop), while the more aggressive analysis sinks waits across intervals. We show how these techniques can be used even for highly-tuned NAS parallel benchmarks (NPB) [10]. Our initial experiments using NAS benchmarks reveal that we can get potential benefit by automatically exploiting CCO in several cases. However, this is not true across the board. For the same benchmark, under differing configurations (NPB classes A, B, C), we may see differing results. While for some we may gain in performance, for others there may not be any perceptible improvement, while some may even show small losses.

Our paper is organized as follows. Section 2 deals with several motivating examples from the NAS parallel benchmark suite for MPI. Section 3 discusses our algorithms for wait sinking. Section 4 is about our early experiences with some of the NAS parallel benchmarks according to the algorithms of Section 3. In Section 5 we provide related work. We end with conclusions and future work in Section 6.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

```
do stage = 1, ncells
   c = slice(1,stage)
    isize =  …  jsize =   ksize =

    if ( stage .eq. ncells ) then last = 1 else last = 0 endif
    if ( stage .eq. 1 ) then
       first = 1
       call lhsx( c )                              // lhs = …
       call x_solve_cell ( first, last, c )        // lhs = … rhs = …
    else
        first = 0
        // calls mpi_irecv ( out_buffer…)
        call x_receive_solve_info(recv_id,c )
        call lhsx(  c )                            // lhs = …

        // wait for the  prev iterations' mpi_isend
        (1) call mpi_wait ( send_id, … )
        call mpi_wait ( recv_id, … )

        // lhs = out_buffer, rhs = out_buffer
        call x_unpack_solve_info ( c )
        call x_solve_cell ( first, last, c )
    endif
    (2)  move wait here
     if ( last .eq. 0 )
        // calls mpi_isend(in_buffer…)
        call x_send_solve_info( send_id, c )
 end do
```

Fig 1 : An example from BT

2. Motivating Examples

In this section, we look at two cases from the NAS Parallel Benchmarks that show the necessity of mpi_wait sinking for improved CCO.

In the example from BT/SP (Fig. 1) the mpi_wait for an mpi_isend (marked as 1 in Fig. 1) crosses an iteration i.e. the mpi_wait in a particular iteration corresponds to mpi_isend issued in the previous iteration(marked as 2 in Fig. 1) through the call to x_send_solve_info(). The code can be found in x/y/z_solve.f.

It can be observed that the mpi_wait in question at point (1) has been placed suboptimally much ahead of the subsequent mpi_isend call in x_send_solve_info which overwrites the in_buffer. The mpi_wait can be pushed all the way down to the point before x_send_solve_info is invoked (but it has to be invoked under the proper condition). The final position is shown in Fig. 1 (using an arrow). Such movement allows the mpi_wait to be delayed as much as possible leading to x_unpack_solve_info and x_solve_cell to be computed in between. The new call to wait at point (2) is semantically safe when invoked conditionally. We will see how this is done in Sec 3.2. using boolean flags that capture the condition under which the original send is invoked.

In MG/mg.f (Fig. 2) there are cases of blocking send. In mg.f the function give3 updates a buffer named buff and invokes several blocking mpi_sends. In another function comm3 there is a call to give3. The call is followed by two instances of take3 which read buff but do not modify it. In such a scenario the latter mpi_send in give3 can be converted to mpi_isend and mpi_wait pairs and the mpi_wait from the mpi_isend call in give3 can be sunk below the take3 pairs as shown in Fig. 2. The wait movement in this case allows two calls to take3 to be spliced in between the isend and wait leading to higher overlap.

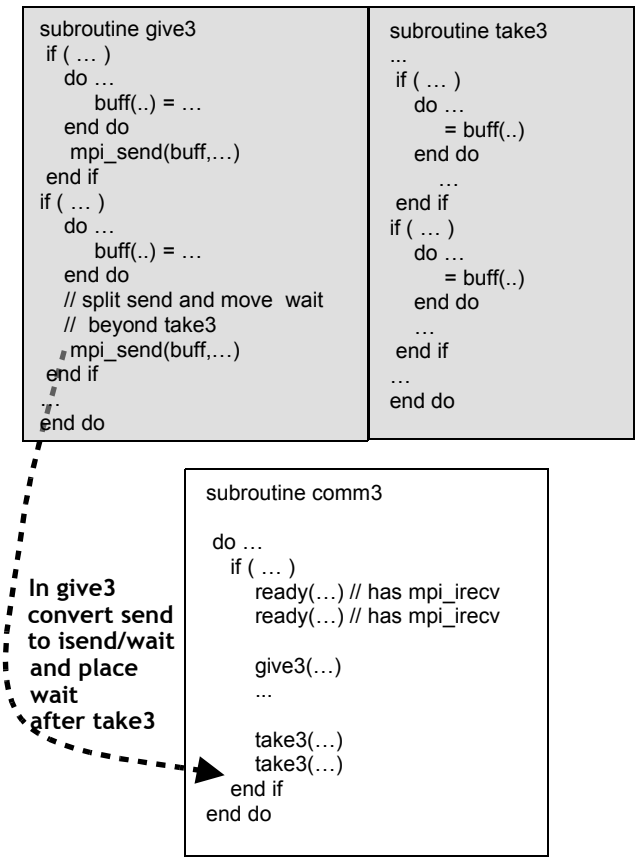

Fig 2: An example from MG

3. Algorithm for mpi_wait Sinking

The algorithm for mpi-wait sinking is divided into three sub-parts.

The first sub-part of the algorithm is known as mpi_wait matching. The algorithm pairs an mpi_wait/waitall with an mpi_isend/irecv. Several mpi_isends or irecvs can pair up with the same mpi_wait/waitall. We handle mpi_waitany or mpi_waitsome calls in a similar manner in which we pair up all possible isends/irecvs with these calls. The mpi_wait matching algorithm works in both the intra-procedural as well as the inter-procedural sense. In Fig. 1, the mpi_waits pair up with an mpi_isend or irecv embedded in x_send_solve_info and x_receive_solve_info calls. However, in our implementation, by applying the wait sinking algorithm after aggressive inlining has been applied, we avoid inter-procedural analyses. The matching algorithm finds the buffers that are associated with a particular mpi_wait and inserts these matching nodes in a graph termed the WaitGraph (WG). The second sub-part of the algorithm is the construction of the mpi_wait dependence web. This involves capturing the dependences between the matched mpi-waits and the following (temporally) instructions that use/define the buffer that has been used in the non-blocking isend/irecv. The final sub-part of the algorithm involves mpi_wait sinking (ConservativeWaitSinkGenerate). It moves the mpi_wait call to a suitable position beyond (temporally) its current position, honouring the dependences captured in the previous step.

3.1 The WaitGraph

The WaitGraph(WG) is a directed graph G=(V,E) whose vertices consist of either mpi call nodes (ex: mpi_wait/isend/...) or other expression nodes that read/write buffers that are used in the matching mpi calls. The edges represent dependences between the nodes. An edge between an mpi_isend node and an mpi_wait node represents a data dependence(technically) between the two while an edge between an mpi_wait and an expression node (that reads/writes the buffer), represents a real WAR or RAW dependence. To give a simple example consider this code:

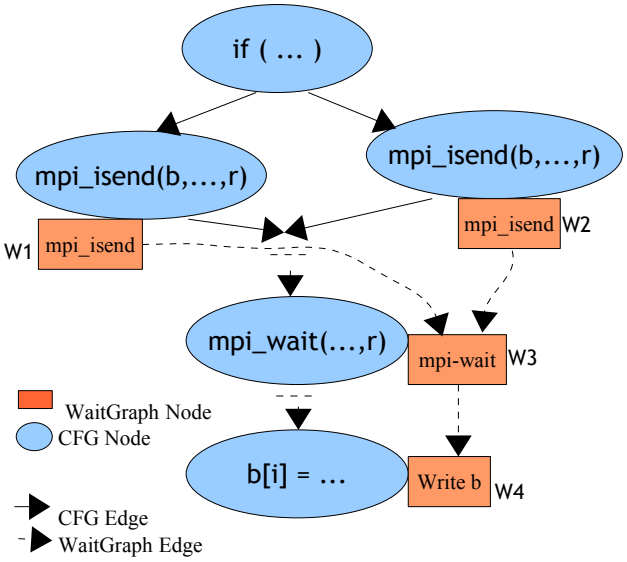

Fig 3: CFG and its WaitGraph

In this example, two calls to mpi_isend use buffer b and the request identifier denoted by r. The mpi_wait call waits on the request identifier r. The buffer b is updated subsequent to the mpi_wait call. The WaitGraph(WG) we construct for this snippet consists of four vertices. These are W1,W2,W3 and W4 – three for the mpi_isend and mpi_wait calls and one for the write to buffer b. There are directed edges from W1 and W2 to W3. These two edges are technical dependence edges representing the isends that match the wait in W3. There is also a data dependence edge (WAR) from W3 to W4 representing the fact that the buffer b is updated at W4 after W3.

3.2 Conservative Algorithm for mpi_wait sinking (Algorithm 1)

Our algorithm works on a per-function basis. All the steps outlined earlier, matching, dependence computation and sinking are carried out intra-procedurally and marked as (1), (2) and (3) respectively in the ConservativeMpiWaitSinking function of Algorithm 1. To make the method effective we invoke this pass after the compiler has carried out inlining. This enables cross-dependences and other complexities to bubble up to a single function level. All cross-interval dependences are satisfied because mpi_waits never move from the original interval where the mpi_sends/recvs are invoked. We will relax this constraint in a later aggressive algorithm (Algorithm 2). For blocking calls like mpi_send/mpi_recv, the matching part is trivial as there is no mpi_wait for such calls. We can view a blocking call as a combination of a non-blocking call immediately followed by a wait call. Our algorithm can then proceed to work on the matched isend/irecv and wait pairs.

The first section of Algorithm 1 captures all the mpi_isends/irecvs/waits/waitalls that appear in a particular function and stores them in a set for later use. We need several analyses to be used before Algorithm 1 can be applied. These include the computation of the dominator tree, SSA use-def information as well as the interval information.

Part 1 of the algorithm deals with mpi_wait matching. It involves extracting an mpi_wait/waitall from the set created earlier(**TS** in Algorithm 1) and then using the request identifier of the mpi_wait call to find out the set of isend/irecv calls that update the same request identifier. The matching pairs up all the mpi_waits with their corresponding dependent mpi_isend/mpi_irecvs. This happens using the SSA use-def information. Once the matched set of calls are known, the wait graph WG can be populated with the nodes that represent the wait and its matched calls (note that a single wait may have several matched calls depending on how the application is written). We consider the matched calls in the same interval as the wait/waitall call as well as those that may occur outside the interval.

Part 2 of the algorithm is used to extract the dependences on the buffer for which the wait is blocked. From the wait call and its matched set we are able to find the "buffer" that the wait call "waits" for. In general the the wait may be dependent on multiple buffers, which we term the BUFFSET in Algorithm 1. It is also necessary to find out where each buffer belonging to BUFFSET is read/written again following the wait. It allows us to find the points in the code where the wait call can be moved for increased overlap and hence better performance. Using SSA use-def we find any statement following the wait (lying in the same interval) that accesses each buffer in the BUFFSET (or its aliases). All such dependence nodes are inserted into the WG with a dependence edge connecting the wait call and the statement that accesses the set of buffers on which wait is dependent. In case of the original statement being an isend we consider only statements that update/write into one of the designated buffers, while in the case of the original statement being an irecv, we track those expression statements where the buffer is being read.

Part 3 of the algorithm carries out the actual code generation, where the wait call is moved around (sunk) to expose higher CCO. This is outlined in the ConservativeWaitSinkGenerate. In this part, each wait/waitall call is visited in the WG graph. If no statement is found in the same interval that has a dependence (due to the buffer read or write) then, the wait for the interval can be sunk to the last lexicographic block of the interval as seen in the SinkCodeGenerate(LastBlock(L),…)**.** LastBlock(L) is the last lexicographic block of a loop, before it jumps back to the start block. For bottom-tested loops, it is just the block which tests for the loop condition. Loop-back dependences are ignored as waits can be sunk to the bottom of the loop in the best case. SinkCodeGenerate emits the new "wait" call and deletes the old wait call. In case where dependences exist within the interval, the wait can be sunk only to a "safe" point such that all the dependences can be satisfied. We use dominator/post-dominator information for ensuring safety.

```
ConservativeMpiWaitSinking(func f)
{
    Capture all isend/irecv/wait/waitall  and store
    all these expression nodes in the set TS

    Compute Dominator Tree, DT, of function f
    WaitGraph = Φ

      (1)   // wait matching phase

    for ( every T1= wait/waitall in TS)  do {
        Find using SSA use-def chain,
        the set of all mpi_isend/irecv(T2set) that
        define the requests which wait consumes

        Create WaitGraph node for T1

        for ( every T2 in T2set )
            Create WaitGraph node for T2 and add a
            technical dependence edge between T1-T2
    }

      (2)   // wait dependence web building

    for ( every wait/waitall node W in WaitGraph ) do {

       Let BUFFSET be the set of buffers waited for
       for ( every buffer in BUFFSET or its aliases ) do {

         Let T be an expression node that reads/writes
         to buffer
         Add  a dependence edge between T and W
        }
    }
      (3)   ConservativeWaitSinkGenerate( //actual sinking
              WaitGraph, IntervalTree(f))
}

ConservativeWaitSinkGenerate(
     WaitGraph WG, IntervalTree ITree)
{
   while ( more wait/waitall are to be processed in WG ) do {

      W = wait/waitall node in the WaitGraph
      Let L be the interval in which W appears

      if ( OutDegree(W) == 0 ) // no dependence

         SinkCode(LastBlock(L), f, W) // SinkCode not shown

      else {
          DBSet = Set of nodes that are children of W in WG
          sinkBlock = lca(DBSet)

          (1) while ( ! done ) {
                if ( sinkBlock postdominates W ) {

                   SinkCode(sinkBlock, f, W )
                   done = true;
                }
                else
                  sinkBlock = Immediate Dominator of sinkBlock
              }
      }
   }
}
```

Algorithm 1

ConservativeWaitSinkGenerate function collects all the dependences in a set termed the DBSet, and computes the "least common ancestor(lca**)**" [13] of all the elements in the DBSet**.** The lca computation uses the dominator tree. If the wait statement is moved to the least common ancestor block (from its original position) it is guaranteed to be "safe" as far as dependences are concerned, as it dominates all the blocks containing the dependent statements (that read or write the buffer). However, when the wait is moved to the least common ancestor block it is not guaranteed to be safe with respect to the original position of wait. The reason for this is the existence of paths leading from isend/irecvs that do not pass through the least common ancestor block. In order to guarantee this, we must find a block that not only dominates the set of blocks in DBSet, but also post-dominates the block containing the original "wait" call. This is done in the while loop of the ConservativeWaitSinkGenerate function (marked as 1), where we move up from the least common ancestor block in the dominator tree, checking to see whether the current block post-dominates the block containing the original wait call. Once we locate such a block it becomes the designated "sink" block i.e. it is the new block where the wait call can be moved from its original position.

When ConservativeWaitSinkGenerate terminates it would have generated new wait calls at sink points via the SinkCode calls. It also deletes the occurrences of the old wait calls. There is a point to note for the new wait calls that are emitted. It is possible that the original waits are called conditionally. In such cases, the new wait calls must also be emitted such that they are invoked only under the original condition(s). This is true for the case in Fig. 1 where the original wait is called conditionally. In order to honour conditional execution, we use the concept of a boolean flag *f* per wait/waitall call. This boolean flag *f* is set to true at all those program points where the original wait/waitall is called. At the new position where the wait call is to be invoked, code is generated to test for the value of *f*. The new wait call is invoked conditionally only for a true value of *f*. Subsequently, code is generated to reset the value of *f* to false. This can be seen in the code generated for MG shown later in Fig. 4.

When implemented on the MG code (Fig. 2), the final code generated after mpi_wait sinking looks as shown in Fig. 4. Inserted code is shown in bold. The ready/take3/give3 functions have been inlined in the code. b_<1>, b_<2> ... are the boolean flags described earlier. In this code the wait matching is trivial as the calls are blocking mpi_sends. As mentioned earlier, we interpret a blocking mpi_send as a couple of adjacent calls comprising of an mpi_isend and an mpi_wait following immediately. The dependence graph computation phase finds one dependence of the buffer used in mpi_send. The two split mpi_wait calls from those created in (1) and (3) are moved to the points (2) and (4). The sink generation phase, creates the if (...) mpi_wait sequence at (2) and (4).The mpi_wait sequences are conditionally executed as the original split mpi_wait calls are also conditionally executed. We may have incorrect results if the matching semantics is not maintained. This semantic matching is achieved through the use of a couple of boolean flags that are set and reset at the appropriate points. The movement of the wait(s) to the positions shown in Fig. 4, can be derived easily if we construct the WaitGraph of the code.

```
subroutine comm3
{
   b_<1> = b_<2> = FALSE;
   Loop:
      // inlined ready
      ...
       // start of inlined give3
       if ( … )
              …
              buff(..) = …
              (1) mpi_isend(buff,…)  ; b_<1> = TRUE;
              ...                          Move  wait of (1) to (2)
       endif
       (2) if ( b_<1> ) mpi_wait(…); b_<1> = FALSE
       if ( ... )
              …
              buff(..) = …
              (3) mpi_isend(buff,…) ; b_<2> = TRUE;
              ...
       end if
      // end of inlined give3              Move  wait of (3) to (4)

      // start of inlined calls to take3
       ...  = buff( ...) // uses of buff
       ...
       ...  = buff(...) // use of buff

      // end of inlined calls to take3
      // mpi_wait has been moved across the inlined calls to
      // take3 as there are only uses of buff but no definitions

      (4) if ( b_<2> ) mpi_wait(…); b_<2> = FALSE
   End of Loop:
}
```

Fig 4: MG example after wait movement

The WaitGraph of Fig. 4 along with some of the CFG nodes and edges are shown in Fig. 5. The figure is simplified to show the important and interesting details. There are three WaitGraph nodes created for the code snippet in Fig. 4. Two of these nodes correspond to the two mpi_send which have been split into two adjacent calls of mpi_isend/mpi_wait. We have shown the WaitGraph nodes corresponding to the two mpi_isend and mpi_wait nodes merged for simplicity. The third WaitGraph node comes into existence during the creation of the dependence web. This node corresponds to the write of the buff.

There is only one true dependence edge between the nodes of the WaitGraph. This is shown by the dotted edge between nodes W1 and W2. The dependence edge signifies that the wait call in the W1 cannot be sunk beyond W2 as there is an update of buff at W2. But for the wait call in W3 there is no data dependence that hinders its movement. Hence, this wait call can be sunk all the way to the end of the loop (shown by the bold edge) which is the enclosing interval of the wait call.

This is also highlighted in Fig. 4 where the wait is moved from point (3) to (4) just before the end of the loop. For the wait call in W1, we find that C4 is not safe as far as sinking the call is concerned because C4 does not post-dominate C2. Using the post-dominator tree we find that the safe point for sinking the wait call in W1 is C3. This is shown by the bold line in Fig. 5 and also highlighted in Fig. 4 where the wait call is seen to move from (1) to (2). Both of the wait calls are conditionally executed based on the values of b_<1> and b_<2> which are set to true/false at the original and new wait invocation points.

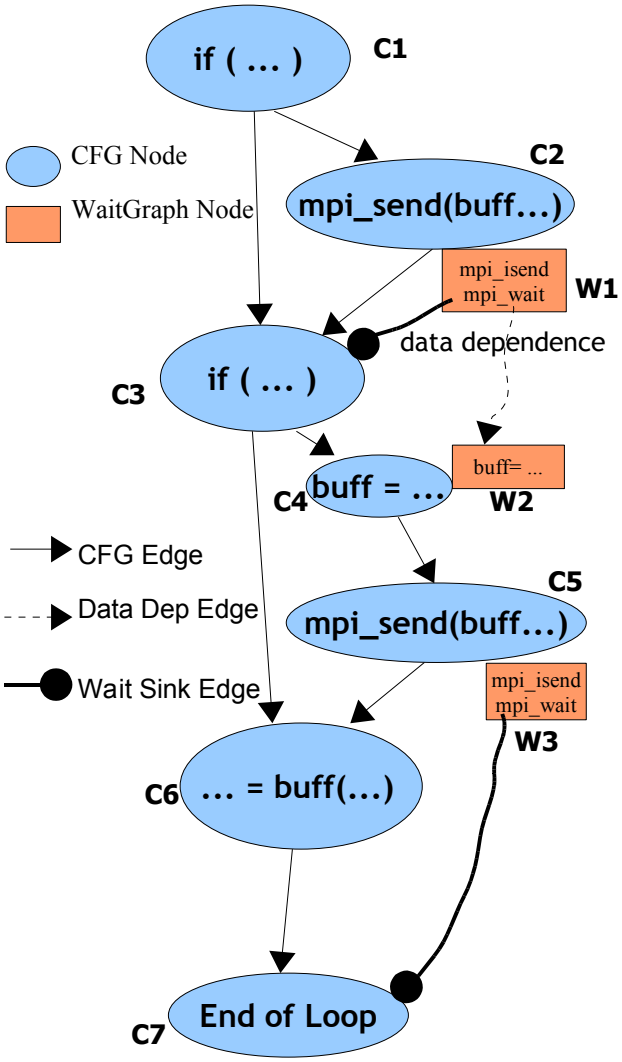

Fig 5: CFG and WaitGraph of Fig 4

3.3 Aggressive Algorithm for mpi_wait sinking using an Inter-Interval(Loop) Approach (Algorithm 2)

The aggressive algorithm for mpi-wait sinking sinks mpi_waits to the points just before the buffers involved are used or defined - instead of sinking to a node which dominates all the use or define points. This allows for sinking waits across intervals and loops in an aggressive manner leading to greater CCO exploitation.

The crux of the algorithm is to locate all the intervals outside of L (assuming wait is in interval L) where the buffers for which mpi_wait/waitall is stalled is being used or defined. Then we find a suitable node where the wait code (called in interval L) can be sunk. To find a node that dominates all use/defs in other intervals for the buffer in question, we first extract the set of all the intervals where the de-

pendences exist (LDSet). Next we compute L'= lca(L,LDset) and find the ancestor A of L in the interval tree that is a direct child of L' that contains L. The exit node of this ancestor A, is the point where the sink code is generated. For the example shown below (Fig. 6) if there is a wait call in interval L1 and there is an usage of the buffer in interval L1', denoted by the dotted arrow dependence edge, then the sink code is generated at the exit point of interval L2. This follows from the algorithm as LDSet={L1'}, lca(L1,LDSet) = L3 and the predecessor of L1 that is a direct child of L3 is L2 and L2 contains L1.

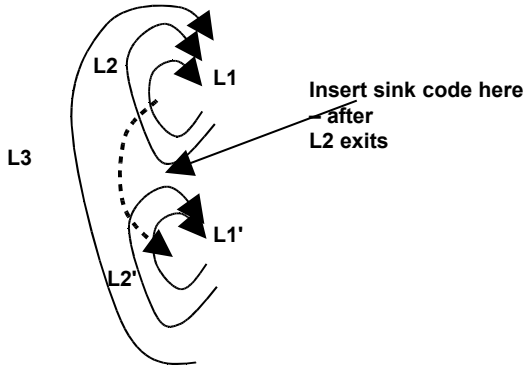

Fig 6: Inter-Interval Wait Sinking Approach

The aggressive algorithm - AggressiveWaitSinkGenerate is used when some of the dependences lie outside the interval L. Algorithm 1 would push the wait(s) to the bottom of the interval (corresponding to L) in such cases conservatively. The aggressive version can sink the wait(s) beyond the interval(s). In the aggressive algorithm, all the dependence generating intervals (that are not the same as L) are collected in LDSet. The block where the wait/waitall can finally be moved is determined by the process described above and denoted as SinkNode.

AggressiveSinkCodeGenerate is invoked to generate the wait(s) at the SinkNode. This sinking may involve a group of waits rather than a single wait we have assumed till now. As we plan to move a wait called in a loop(L), outside it we may need to compensate for all the possible wait(s) that could have been called in the loop. AggressiveSinkCodeGenerate creates an array of resource ids, Rid[ ] and two additional variables denoted as w_cnt and i_cnt for each wait. These variables are used to keep track of the next resource id to be used and the resource id already consumed. Code needs to be inserted both at the original wait call point, at the non-blocking call points as well as the SinkNode. At the isend/irecv calls we need to increment i_cnt to keep track of outstanding requests. At each of the dependence point we need to insert an if-check to find out whether at this point we really need to wait or not. This is done by checking the difference of the i_cnt and w_cnt values. If the result of the comparison is positive it implies that there is/are outstanding request(s) for which a wait needs to be called. w_cnt is also incremented at each dependence point. At the SinkNode, we need to generate (i_cnt–w_cnt+1) number of wait calls (or preferably a single waitall), as these are all outstanding requests for which waits have not been emitted yet.

```
AggressiveWaitSinkGenerate(
        WaitGraph WG, IntervalTree ITree)
{
   while ( more mpi_wait/waitall to be processed in WG ) do {

      Let W = mpi_wait/waitall node in the WaitGraph WG

      if ( some dependences of W are outside L ) {
         Let LDSet = { set of all intervals ,different from L,
                       where dependences exist  }

         // Find least common ancestor of the dependent
         // set as well as the interval where wait appears

         Find SinkInterval = lca(LDSet,L) in ITree

         // compute the sinking interval
         Lpred = pred(L) // predecessor of L in the ITree

         while ( Lpred != SinkInterval )
            Lpred = pred(Lpred) ;

         Let SinkNode = Exit Node of Lpred

         AggressiveSinkCodeGenerate( SinkNode, W, WG)
      }
   }
}

AggressiveSinkCodeGenerate(
       Node SinkNode, Node WaitNode, WaitGraph WG)

{
   L = Interval in which Waitnode is present

   Create three variables :
   w_cnt_<num>, i_cnt_<num>, Rid_<num>[ ]
   // num is a compiler generated global that is incremented
   // every time this function is called

   Let M = matched mpi_isend/irecv node

   Insert (after M) the code :
      i_cnt_<num>++

   Replace the request id of mpi_isend/irecv to create:
      mpi_isend(buff,...,Rid_<num>[i_cnt_<num>],...) ;

   for ( every dependence point in L) do {

      Insert  an if-conditional to check:
         if ( w_cnt_<num>  >  i_cnt_<num> ) {
            mpi_wait(...,Rid_<num>[w_cnt_<num>])
            w_cnt_<num>++
         }
   }
   At SinkNode do the following:
   {
       Insert code to create a waitall:

       mpi_waitall(...,Rid_<num>,...,
                       i_cnt_<num> - w_cnt_<num> + 1);
   }
   num++;
}
```

Algorithm 2

Fig. 7 employs the inter-interval aggressive wait sinking algorithm for the MG example to arrive at the new code. As there are two mpi_wait points in the loop we employ two w_cn**t,** i_cnt and Rid arrays. w_cnt keeps track of how many waits have been issued and matched, while i_cnt keeps track of how many isends/irecvs have been issued. This is done for each matched pair of isend and wait. For any intra-loop dependence that exists, conditional code is employed to generate waits to check whether they are really required at runtime.

```
subroutine comm3
{
   integer  w_cnt_<1> = w_cnt_<2> = 0
   integer  i_cnt_<1> = i_cnt_<2> = 0;
   integer  Rid_<1>[loop_cnt], Rid_<2>[loop_cnt];

   Loop:
      ...
      if ( … )
         ...
            if (i_cnt_<1> - w_cnt_<1>  .gt. 0 )
               mpi_wait(…,Rid_<1>[w_cnt_<1>]);
               w_cnt_<1>++;
            endif

            if (i_cnt_<2> - w_cnt_<2> .gt. 0 )
               mpi_wait(…, Rid_<2>[w_cnt_<2>]);
               w_cnt_<2>++;
            endif
            …
            buff(..) = …
            mpi_isend(buff,…,Rid_<1>[i_cnt_<1>],…) ;
            i_cnt_<1>++;
            ...
      endif
      if ( … )
            …
            if (i_cnt_<1> - w_cnt_<1>  .gt. 0 )
               mpi_wait(…,Rid_<1>[w_cnt_<1>]);
               w_cnt_<1>++;
            endif

            if (i_cnt_<2> - w_cnt_<2> .gt. 0 )
               mpi_wait(…, Rid_<2>[w_cnt_<2>]);
               w_cnt_<2>++;
            endif

            buff(..) = …
            mpi_isend(buff,…,Rid_<2>[i_cnt_<2>],…) ;
            i_cnt_<2>++;
         ...
      end if
      …
   // End of Loop
   // Groups of waits for isends that did not get
   // consumed in the Loop

   mpi_waitall(…,Rid_<1>,...,i_cnt_<1>-w_cnt_<1>+1);
   mpi_waitall(…,Rid_<2>,...,i_cnt_<2>-w_cnt_<2>+1);
   ...
}
```

Fig 7: MG example after aggressive wait movement

4. Current Status and Experiments

We have currently implemented only the conservative algorithm in the IBM's xlc/c++/fortran compiler series [11] as a proof-of-concept. The compiler identifies all the mpi calls that need to be moved/split and the final positions where the mpi_waits need to be placed. The actual splitting and placement is done by hand from the report. For our experiments we have taken the NAS Parallel Benchmarks. Here, we mainly concentrate on three benchmarks : MG, LU and SP.

The hardware configuration used consists of IBM Power5+ CPUs, connected by a High Performance Federation Switch. IBM's Parallel Operating Environment (POE) [8] is used to launch the MPI applications on these systems. The POE runs were carried out by setting the environment variable **MP_CSS_INTERRUPT=ON** which allows for independent progress. Without this variable being set, asynchronous and independent progress of communication and computation does not happen and we may actually lose significant performance for rendezvous messages. The runtime reported here are for an average of five runs.

4.1 MG

In this subsection we show the results obtained from MG according to the conservative wait-sinking algorithm. The MG benchmark can only be run for a number of processors whose power is 2. We have run MG for classes B and C and for two configurations of processors, 32 and 64. We observed that class B had a greater impact (up to 20% improvement) when our algorithm was applied, compared to class C, where the improvement is very small (Table 1)

4.2 LU

LU has blocking calls which can be converted to non-blocking calls (as in MG) and then moving the corresponding waits. The LU benchmark can only be run for a number of processors whose power is 2. We have run LU for classes B and C for 32 processors only. This is because we saw high runtime both for the unoptimized as well as optimized codes when 64 processors are used. We are investigating the reasons for it. When 32 processors are used, we saw a modest speedup for class C while for class B the speedup is really high(26%) (Table 2).

4.3 SP

This NAS benchmark already uses some form of overlap. Overlap is exploited by using isend/irecv and waits. Hence, the only optimization that could be applied was to sink the waits so that we could create a higher computation-communication overlap. There was no scope for replacing blocking sends/recvs. SP can be run only for a square number of processors. We ran SP using 16, 25 and 36 processors for classes B and C. We saw a significant improvement for class B when 25 processors are used. For others, the improvement is marginal or flat. We saw a small slowdown with a B class run using 36 processors. The slowdown was higher at 4% for class C with 36 processors(Table 3). This is probably due to the overhead of thread switching (a special thread is required for independent progress when interrupt mode is on) overshadowing the benefit of overlap when such benefit is small.

Table 1: MG

Class	#processors	Unoptimized run time (in secs)	Optimized (split sends and move waits) runtime	%improvement
B	32	1.53	1.22	20.3
B	64	1.12	1.02	9.0
C	32	7.85	7.82	0.4
C	64	4.32	4.31	0.25

Table 2: LU

Class	#processors	Unoptimized run time (in secs)	Optimized (split sends and move waits) runtime	%improvement
B	32	37.9	27.8	26
C	32	122.1	114.9	5

Table 3: SP

Class	#processors	Unoptimized run time (in secs)	Optimized (split sends and move waits) runtime	%improvement
B	16	59.5	58.8	1.1
B	25	69.8	60.2	13.8
B	36	61.7	62.2	-0.8
C	16	152.5	151.3	0.8
C	25	106.6	106.3	0.3
C	36	77.3	80.4	-4.0

5. Related Work

Previous work in the area of CCO also termed as split-phase communication, have targeted various languages like UPC, Parallel C and HPF [1,5,6]. However none of them have have tackled explicit message passing applications like MPI. In [1] Chakrabarti et al. deal with HPF and the main purpose of the work is message coalescing rather than CCO. The work [5] by Iancu et al for UPC comes closest to ours. They try to build an automated tool for UPC [9] that tries to exploit overlap as well as message coalescing. However, it uses a very simplistic algorithm to expose CCO for UPC code. Their technique is inadequate for non-trivial applications. We have devised both conservative as well as aggressive algorithms for CCO exploitation with which we can tackle complicated MPI applications and non-trivial send/recv sequences. This is specifically true when such sequences are invoked in loops and have intervening function calls, for which [5] does not do anything. The aggressive algorithm is able to introduce groups of mpi_waits at the end of loops for mpi_isends/mpi_irecvs which have not been waited for inside a loop, as part of wait sinking.

6. Conclusion and Future Work

Our work outlines how mpi_waits can be moved/sunk to allow better opportunities for overlap of computation with communication. We have developed a conservative and an aggressive strategy to effect this movement. In the first case, the mpi_waits can move only within the interval where they are originally invoked. In the aggressive algorithm, we loosen this restriction, thereby allowing mpi_waits originating in one interval to be finally sunk to a parent interval. On experimenting with some of the NAS benchmarks according to the algorithms we devised, we noted moderate to good speedups for certain classes and configurations. We also saw small drops in performance for some benchmarks. For other configurations, the performance was flat. From these we can conclude that exploiting CCO may lead to good-to-moderate performance improvements in some cases while for others it may not yield anything. However, we are also exploring the reasons of lower/negative speedups for the bigger classes of NAS benchmarks as well as for those which use a larger number of processors. We would also like to apply our analyses to real-world applications as well as the specmpi2007[12] benchmark suite, where gains may be much higher, as many of them may not be optimized for overlap. Future work also involves supporting the aggressive algorithm via the compiler.

Acknowledgements

We would like to thank Raul Silvera and Kit Barton of the IBM compiler TPO team and the anonymous referees for their useful suggestions.

7. Bibliographic References

[1] Global Communication Analysis and Optimization**,** Soumen Chakrabarti, Manish Gupta and J. D. Choi, *SIGPLAN Programming Language Design and Implementation(PLDI) 1996, pp 68-78*.

[2] Performance Analysis Of Distributed Applications using Automatic Classification of Communication Inefficiencies, Jeffrey Vetter. *Intn'l Conf on Supercomputing, 2000*, *pp 245-254*.

[3] Automated Approach to Improve Communication-Computation Overlap in Clusters**,** L. Fishgold, A. Danalis, L. Pollock, M. Swany, *Parallel Computing 2005*, *pp 481-488.*

[4] Using Overdecomposition to Overlap Communication Latencies with Computation and Take Advantage of SMT Processors, Lars Ailo Bongo, Brian Vinter, Otto J Anshus, Tore Larsen, John Markus Bjorndalen, *Intl. Conf. Workshops on Parallel Processing, 2006*, *pp 239-247*.

[5] Communication Optimizations for Fine-grained UPC applications, Wei-Yu Chen, Costin Iancu and Katherine Yelick, *Parallel Architectures and Compiling Techniques 2005*, *pp 267-278*.

[6] Communication Optimizations for Parallel C programs, Y. Zhu and L. Hendren, *SIGPLAN Programming Language Design and Implementation (PLDI) 1996, pp 199-211.*

[7] MPI: A Message Passing Interface Standard, MPI Forum.

[8] IBM Parallel Operating Environment Manual, Vol I and II.

[9] Unified Parallel C, http://upc.gwu.edu

[10] NAS Parallel Benchmarks,http://www.nas.nasa.gov/Software/NPB

[11] Overview of the IBM XL C/C++ and XL Fortran compiler, http://www.ibm1.com/support

[12] specmpi2007 benchmark suite in http://www.spec.org

[13] On loops, dominators and dominance frontiers**,** G Ramalingam, ACM Transactions on Programming Language and Systems, Vol 24(5), pp 455-490.