Exploring Computation-Communication Overlap in some NAS Parallel Benchmarks using Program-level Software Pipelining

Rishabh Uppal(IIT Kanpur), Dibyendu Das, Rajan Ravindran (IBM India Systems &Technology Lab)
Contact:rishabh@cse.iitk.ac.in, {dibyendu.das,rrajan}@in.ibm.com

1. Introduction

Computation communication overlap (CCO) is an old problem as far as distributed computation is concerned. The objective of CCO is to pack in as much computation as possible between a non-blocking communication call (ex: MPI_Isend) and its corresponding blocking wait primitive (ex: MPI_Wait). This , in general, results in lower execution time. However, the problem of extracting CCO automatically from an application has not been tackled satisfactorily in the MPI (or similar) programming space. Several reasons exist for this. Principal among them is the non-trivial nature of transforming an MPI based program (or similar environments like UPC) for exploiting overlap. Previous efforts in the CCO space have either concentrated on fine-grained architectures or specifically used languages like HPF. Only recently have there been some studies to look at coarse-grain CCO from the perspective of a compiler or other automatic transformers [2,3]. In this work we will look at some MPI-based NAS benchmarks to see how we can use software pipelining (SWP) at the program level to achieve computation-communication overlap. Program-level software pipelining will imply application of software pipelining transformation on some of the loops of the high-level code itself, in order to extract as much CCO as possible. These loops are the ones which contain communication primitives like MPI_Send/Recv or their non-blocking counterparts like MPI_Isend/Irecv and the matching MPI_Waits. We make the distinction clear as software pipelining is usually carried out in the low-level-optimizer component of an optimizing compiler only after pseudo-machine instructions have been generated [4].

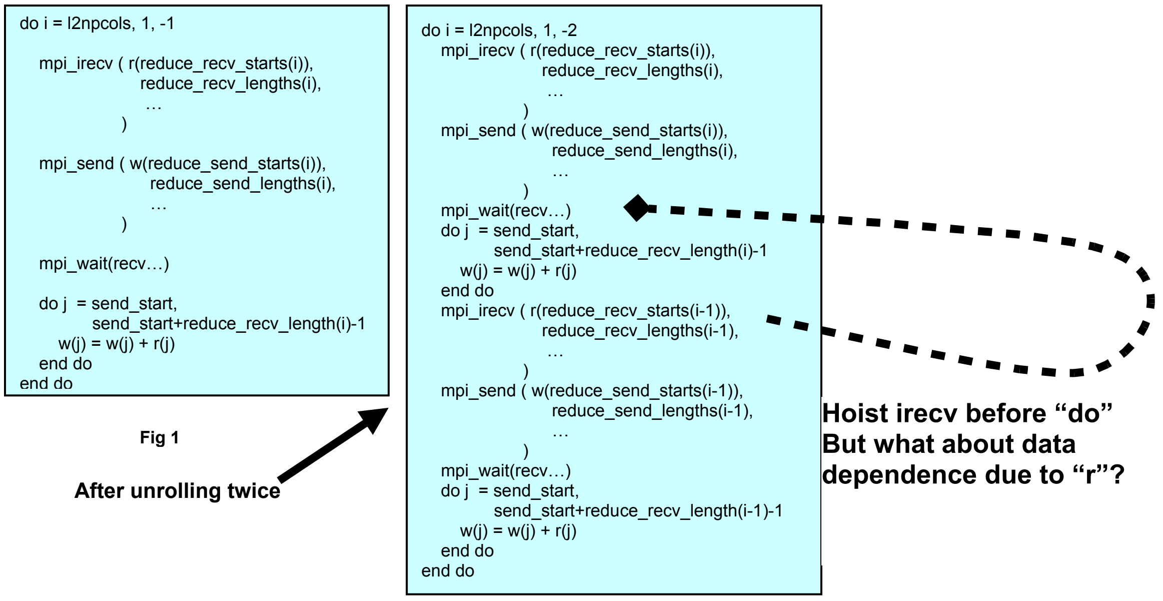
The main motivation of this work is to explore whether software pipelining at the program level can exploit opportunities in highly-tuned benchmarks like NAS. That can provide a good indicator for general HPC code or other benchmarks. Optimizing compilers and automatic transformers can then be re-tooled to utilize their analyses to convert MPI-based code such that software pipelining can be applied in a transparent manner as a high level transformation. This will help extract a greater level of overlap than what exists in an application before the pipelining transformation is applied - without programmer intervention.

2. Program-level Software Pipelining

In this work the goal is to identify hot loops that carry MPI communication primitives, unroll them by a small constant factor (in our experiments we have used a unroll factor or 2) and then reorder the communication primitives so that overlap can be achieved. We have devised a new method called **conditional software pipelining** that allows us to carry out software pipelining even when all the data dependences are not known fully at compile time.

Conditional Software Pipelining

We will illustrate it more vividly with this loop from the NAS benchmark, CG. This pattern can be found frequently in cg.f



```
do i = l2npcols, 1, -1

   mpi_irecv ( r(reduce_recv_starts(i)),
               reduce_recv_lengths(i),
               …
             )

   mpi_send ( w(reduce_send_starts(i)),
              reduce_send_lengths(i),
              …
            )

   mpi_wait(recv…)

   do j  = send_start,
         send_start+reduce_recv_length(i)-1
      w(j) = w(j) + r(j)
   end do
end do
```

```
do i = l2npcols, 1, -2
   mpi_irecv ( r(reduce_recv_starts(i)),
               reduce_recv_lengths(i),
               …
             )
   mpi_send ( w(reduce_send_starts(i)),
              reduce_send_lengths(i),
              …
            )
   mpi_wait(recv…)
   do j  = send_start,
         send_start+reduce_recv_length(i)-1
      w(j) = w(j) + r(j)
   end do
   mpi_irecv ( r(reduce_recv_starts(i-1)),
               reduce_recv_lengths(i-1),
               …
             )
   mpi_send ( w(reduce_send_starts(i-1)),
              reduce_send_lengths(i-1),
              …
            )
   mpi_wait(recv…)
   do j  = send_start,
         send_start+reduce_recv_length(i-1)-1
      w(j) = w(j) + r(j)
   end do
end do
```

Fig 1

After unrolling the loop shown on the left hand side of Fig 1 twice, we notice that we can hoist the irecv of the i-1th iteration(note that the loop moves backward) to the point above the computation of w and just after the wait of the irecv for the ith iteration. However, the second irecv may overwrite (partially or fully) the r buffer which may result in wrong computation of w, when it is hoisted to the position shown. To tackle the possible overwrite problem (due to the presence of a possible **anti** or **WAR** dependence) we guard the irecv code when hoisted. We also leave a guarded copy of irecv at its original place but with the guard condition being the complement of the condition used in the hoisted irecv. **DoesRangeOverlap**(r1,r2) checks whether r1 ∩r2 != Φ. Here r1 and r2 represent the array access regions of r.

```
do i = l2npcols, 1, -2

    mpi_irecv ( r(reduce_recv_starts(i)), …)
    mpi_send ( w(reduce_send_starts(i)),…)
    mpi_wait(recv…)

    R1 = {send_start, reduce_recv_length(i) +  send_start -1}
    R2 = {reduce_recv_starts(i-1), reduce_recv_lengths(i-1) + reduce_recv_starts(i-1)}
    // condition code added in case there is a dependence
    if  ( ! DoesRangeOverlap(R1,R2) )
        mpi_irecv ( r(reduce_recv_starts(i-1),…)

    do j  = send_start,
             send_start+reduce_recv_length(i)-1
        w(j) = w(j) + r(j)
    end do
    // condition code added in case there is a dependence
    if ( DoesRangeOverlap(R1,R2))
        mpi_irecv ( r(reduce_recv_starts(i-1),…)

    mpi_send ( w(reduce_send_starts(i-1),…)
    mpi_wait(recv…)
    do j  = send_start,
             send_start+reduce_recv_length(i-1)-1
        w(j) = w(j) + r(j)
    end do
end do
```

Fig 2

Now is the time to peel the first iteration of the loop and create a kind of prolog/epilog and the kernel (as in software pipelining). The peeled and "pipelined" code will look like this:

```
// prolog of Software-Pipelined Loop
mpi_irecv ( r(reduce_recv_starts(l2npcols)), …)
mpi_send ( w(reduce_send_starts(l2npcols)),…)

// start kernel of Software-Pipelined Loop
do i = l2npcols, 2, -1    // do one iteration less
    mpi_wait(recv…)

    R1 = {send_start, reduce_recv_length(i) +  send_start -1}
    R2 = {reduce_recv_starts(i-1), reduce_recv_lengths(i-1) + reduce_recv_starts(i-1)}

    if ( ! DoesRangeOverlap(R1,R2) )
        mpi_irecv ( r(reduce_recv_starts(i-1),…)

    do j  = send_start,
             send_start+reduce_recv_length(i)-1
        w(j) = w(j) + r(j)
    end do

    if ( DoesRangeOverlap(R1,R2))
        mpi_irecv ( r(reduce_recv_starts(i-1),…)

    mpi_send ( w(reduce_send_starts(i-1),…)
end do
// end kernel of Software-Pipelined Loop

// epilog  of Software-Pipelined Loop
mpi_wait(recv…)
do j  = send_start,
             send_start+reduce_recv_length(i)-1
        w(j) = w(j) + r(j)
end do
```

Fig 3

The unrolled and pipelined loop can now be seen as one where the irecv of iteration i-1 overlaps with the computation of w for iteration i. This also increases the distance between the **irecv** and its corresponding **wait**. This allows for more computation to be sandwiched between the irecv and wait leading to better overlap. If you notice the original program carefully we can see that the CCO is only between the **irecv** and its corresponding **wait** and the only thing that happens between these two is a **send**. We can visualize the original loop as a string where the following pattern repeats (where IR=mpi_irecv, S=mpi_send, W=mpi_wait and C=computation).

$$\begin{tikzcd}
\mathbf{IR_i} & \mathbf{S_i} & \mathbf{W_i} & \mathbf{C_i} \arrow[lll, bend left]
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbf{IR_n} & \mathbf{S_n} & \mathbf{Wi} & \mathbf{IR_{i-1}} & \mathbf{Ci} & \mathbf{S_i} & \mathbf{W_1} \arrow[llll, bend left] & \mathbf{C_1}
\end{tikzcd}$$

The original loop is on the left hand side(lhs), while the unrolled and pipelined loop is on the right hand side(rhs). The CCO has increased from the lhs where only S_i is active between the IR_i and W_i, to the rhs where CCO is due to C_i + S_i between IR_{i-1} and W_i of the next iteration. The original loop computation has been perturbed due to the introduction of the **DoesRangeOverlap** computation. We can optimize this further by hoisting the entire range computation as a separate loop invoked just prior to the main loop. This will cause the original loop to be perturbed very lightly. The new-look loop will now be as follows:

Fig 4

```
do i = l2npcols, 1, -1
   R1 = {send_start, reduce_recv_length(i) +  send_start -1}
   R2 = {reduce_recv_starts(i-1), reduce_recv_lengths(i-1) + reduce_recv_starts(i-1)}
   RANGETABLE(i) = DoesRangeOverlap(R1,R2)
end do
//prolog
mpi_irecv ( r(reduce_recv_starts(l2npcols)), …)
mpi_send ( w(reduce_send_starts(l2npcols)),…)
//kernel
do i = l2npcols, 2, -1
   mpi_wait(recv…)
   if  ( ! RANGETABLE(i) )
      mpi_irecv ( r(reduce_recv_starts(i-1),…)

   do j  = send_start,
           send_start+reduce_recv_length(i)-1
      w(j) = w(j) + r(j)
   end do
   if ( RANGETABLE(i))
      mpi_irecv ( r(reduce_recv_starts(i-1),…)

   mpi_send ( w(reduce_send_starts(i-1),…)
end do
//epilog
mpi_wait(recv…)
do j  = send_start,
           send_start+reduce_recv_length(i)-1
      w(j) = w(j) + r(j)
end do
```

Unconditional Software Pipelining

Now, we will look at a subroutine from the NAS benchmark BT/x-solve.f . This subroutine demonstrates a couple of complications, chief among them being: a) The isend/irecv codes are embedded within calls and b) There is a conditional inside the loop. Conditionals inside loops make software pipelining a more complex venture when attempted automatically.

Fig 5

```
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
        call x_receive_solve_info(recv_id,c ) // calls mpi_irecv ( out_buffer…)
        call lhsx(  c )                          // lhs = …

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

        call x_unpack_solve_info ( c )           // lhs = out_buffer, rhs = out_buffer
        call x_solve_cell ( first, last, c )     // lhs = …, rhs = …
    endif
     if ( last .eq. 0 ) call x_send_solve_info( send_id, c ) // in_buffer = …, mpi_isend(in_buffer…)
enddo
```

Notice that the conditionals are such that all of them can be removed by peeling some iterations either at the start of the loop or at the bottom to get a loop without any conditional. The loop is then unrolled by 2 to produce the code below.

Fig 6

```
do stage = 2, ncells-1,2
    c = slice(1,stage) isize =  …  jsize =   ksize =

    last = 0     first = 0
    call x_receive_solve_info(recv_id,c ) // calls mpi_irecv ( out_buffer…)
    call lhsx(  c )                              // lhs = …

    call mpi_wait ( recv_id, … )                 // wait for mpi_irecv

    call x_unpack_solve_info ( c )               // lhs = out_buffer, rhs = out_buffer
    call x_solve_cell ( first, last, c )         // lhs = …, rhs = …

    call mpi_wait ( send_id, … )                 // wait for the prev iterations' mpi_isend
    call x_send_solve_info( send_id, c ) // in_buffer = …, mpi_isend(in_buffer…)
    c = slice(1,stage+1) isize =  …  jsize =   ksize =

    last = 0     first = 0
    call x_receive_solve_info(recv_id,c ) // calls mpi_irecv ( out_buffer…)
    call lhsx(  c )                              // lhs = …

    call mpi_wait ( recv_id, … )                 // wait for mpi_irecv

    call x_unpack_solve_info ( c )               // lhs = out_buffer, rhs = out_buffer
    call x_solve_cell ( first, last, c )         // lhs = …, rhs = …

    call mpi_wait ( send_id, … )                 // wait for the last mpi_isend
    call x_send_solve_info( send_id, c ) // in_buffer = …, mpi_isend(in_buffer…)
end do
```

Can hoist irecv in x_receive_solve_info

In this case, the x_receive_solve_info(…) can be hoisted to be positioned after the call to x_unpack_solve(…). We cannot hoist it further without using a different buffer (and thus breaking the WAR dependence). The s/w pipelined loop will now look like this:

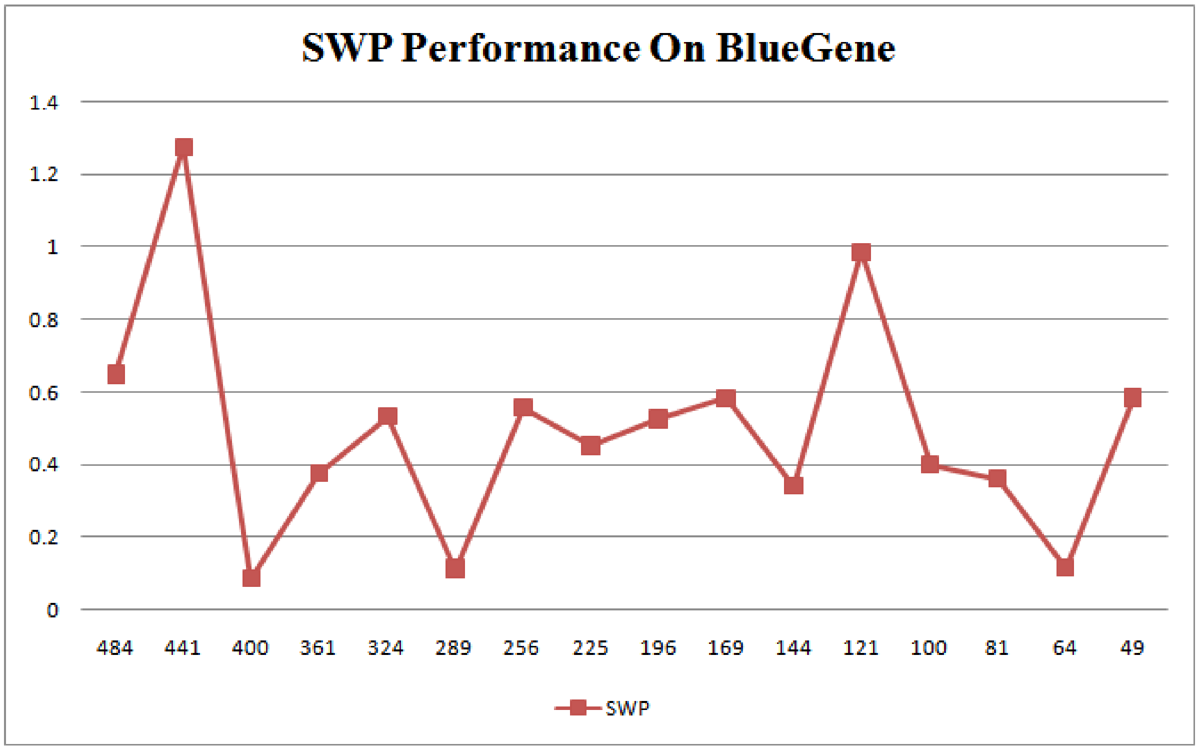
```
Pre Loop Computation
        call lhsx(c)
        call x_solve_cell(first,last,c)
        call x_send_solve_info(send_id,c)
        call x_receive_solve_info(recv_id,c)

Loop
     do stage = 2,ncells – 1                    // Loop iterations reduced by 2
// c used below is from previous iteration
          call lhsx(c)
          call mpi_wait(recv_id,r_status,error) // previous iteration mpi_ircv's wait
          call x_unpack_solve_info(c)
          c_temp = slice(1,stage + 1)          //  c calculated as c_temp for next iteration
// Below is receive for current iteration and has corresponding wait in next iteration
          call x_receive_solve_info(recv_id,c_temp)
//Below calculation is done on pervious iteration's receive
          call x_solve_cell(first,last,c)
          call mpi_wait(send_id,r_status,error)
          call x_send_solve_info(send_id,c)
          c = c_temp                           // c is substituted for next iteration
     enddo

Post Loop Computation
          c = slice(1,ncells)
          call lhsx(c)
          call mpi_wait(recv_id,r_status,error)
          call x_unpack_solve_info(c)
          call x_solve_cell(first,last,c)
          call mpi_wait(send_id,r_status,error)
```

Fig 7

3. Experimental Results and Future Work

Experiments were conducted using hand-changed NAS benchmarks on a 512 node BlueGene/L machine. The results shown below are for the benchmark BT class-C. We see small improvements in performance (up to 1.3%+) for all configurations even for such a highly tuned benchmark as BT. The x-axis shows the number of processing nodes used (BT requires a square number) while the y-axis provides the percentage improvement over non-SWP code.

Future work involves incorporating program level software pipelining in an optimizing compiler and see the effects of the transformations on a larger set of benchmarks and applications. This will provide a better understanding of whether automatic program-level software pipelining can be used to extract higher computation-communication overlap from MPI applications in a sustained manner.



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. *Automatic MPI application transformation with ASPhALT*, Anthony Danalis, Lori Pollock, and Martin Swany. Intl Parallel and Distributed Processing Symposium, 2007. pp 1-8.
3. *Communication Optimizations for Fine-grained UPC applications*, Wei-Yu Chen, Costin Iancu and Katherine Yelick, Parallel Architectures and Compiling Techniques 2005, pp 267-278.
4. *Iterative Modulo Scheduling*, B R Rau, HP Labs Tech Report, 1995.