ASSIGNMENT 2 SOLUTIONS

HPC1 Fall 2013

Due Date: ***Tuesday, October 1***

(please submit your report electronically to the instructor via email, as one PDF file named *hw2-yourUBitname.pdf*)

Problem 1: Write your own vector dot product benchmark code. For simplicity, just consider **double** (double precision).

a) Write a small benchmark code for the L1 BLAS routine for vector dot products. Note that for small vector lengths, depending on the granularity of your timer, you may need to average over a number of dot product evaluations.

b) Link with a reference copy of the BLAS (*e.g.*, -lblas on U2, or you can download a copy of the source from netlib if you prefer), and plot your results in terms of MFlop/s as a function of vector length. Do you see any dependence on compiler flags?

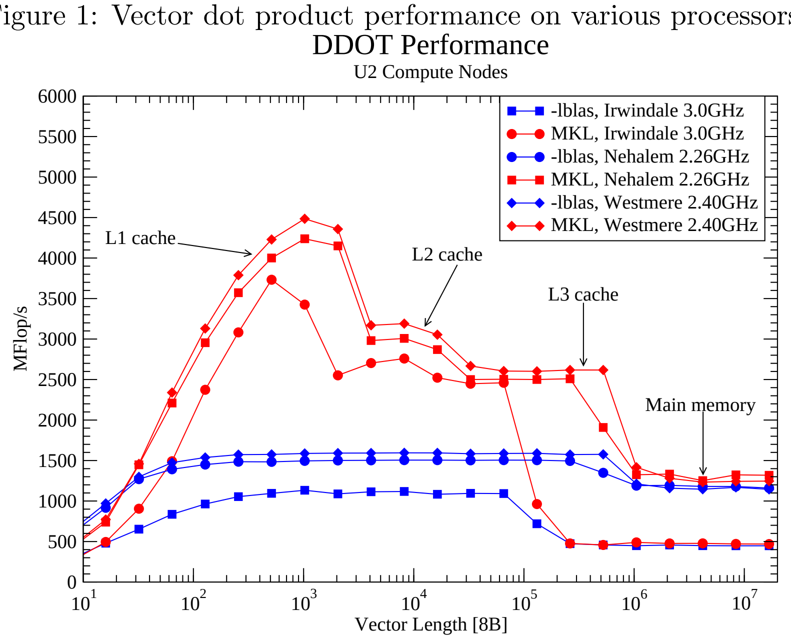
c) Now use an optimized BLAS library (*e.g.*, Intel's MKL), and repeat the above, plotting both results together. You should see significantly different behavior than part **b**. Interpret your results. Note that Intel has a web tool to facilitate linking applications against the MKL,

`http://software.intel.com/sites/products/mkl/MKL_Link_Line_Advisor.html`

Solution:

The plot below (and included source code) shows the `DDOT` performance as a function of vector length using the reference and optimized BLAS.

Figure 1: Vector dot product performance on various processors.



Note the influence of the processor's cache memory hierarchy (U2's oldest "Irwindale" processors have two levels, L1 (32KB) and L2 (2MB), while the newer Nehalem and Westmere-based nodes have L1 (32KB), L2 (256KB per-core) and a shared per-socket L3 (8MB for Nehalem, 12MB for Westmere)), but most significantly on the *optimized* BLAS. That is due to the optimization making maximum use of the appropriate *blocking* for the processor, namely it can feed the CPU perfectly sized "chunks" of the vectors to maintain the optimal floating-point operation count. The reference BLAS ignores these features - you can mimic them yourself by *unrolling* the loops, such that each iteration actually executes multiple element products and accumulates the sum.

```fortran
program bench_ddot
  implicit none

  integer,parameter :: sp=KIND(1.0),  &
       & dp=SELECTED_REAL_KIND(2*PRECISION(1.0_sp))

  integer,parameter :: Nmax=2000
  real(kind=dp),allocatable :: A(:),B(:)
  real(kind=dp),parameter :: one=1.0_sp,zero=0.0_sp
  !real(kind=sp),allocatable :: Adp(:,:),Bdp(:,:),Cdp(:,:)
  real(kind=dp) :: sum

  double precision :: myddot
  external myddot

  integer :: astat,n_start,n_stop,n_incr,n_iter,nreps,N
  integer :: i,j,k,l

  real :: t_start,t_end,t_ave

  n_start = 16
  n_stop = 1024*1024*4
  n_incr = 2
  N = n_start
  !
  ! want each timing interval to be ~1s
  ! use 2GFlop/s to guess number of repetitions
  !
  write(*,'(a8,2x,a12,a10,a12)') "N","time","Nreps","MFlop/s"
  do while (n <= n_stop)
     ALLOCATE(A(N),B(N),stat=astat)
     if (astat.ne.0) then
        print*,'Unable to allocate arrays of order ',N
        STOP 'memory allocation error'
     end if
     CALL RANDOM_NUMBER(A)
     CALL RANDOM_NUMBER(B)
     !
     ! Number of repetitions for accurate timing
     !
     nreps = 1.0/(n*1.e-9)
     nreps = MAX(nreps,10)
     call cpu_time(t_start)
     sum = 0.0
     do i=1,nreps
        sum=myddot(N,A,B)
     end do
     call cpu_time(t_end)
     t_ave = (t_end-t_start)/nreps
     write(*,'(i8,2x,e12.4,i10,f12.2)') N,t_ave,nreps, &
          & (2.0*N*1.e-6)/t_ave
     N = N*n_incr
     DEALLOCATE(A,B)
  end do
end program bench_ddot

double precision function myddot(N,A,B)
  implicit none
  double precision A(*),B(*)
  integer :: N
#ifdef _USEBLAS
  double precision :: ddot
  external ddot
#endif

  integer :: i
  double precision :: sum
#ifdef _USEBLAS
  sum=DDOT(N,A,1,B,1)
#else
  sum = 0.d0
  do i=1,N
     sum = sum + A(i)*B(i)
  end do
#endif
  myddot = sum
  return
end function myddot
```

Problem 2: Use a simple MPI code to perform the "Ping-Pong" benchmark using blocking sends and receives. Time your results for buffer sizes ranging from, say, 8 Bytes to 8 MBytes, and plot the resulting message times and bandwidth (also identify the approximate latency) for:

a. U2 nodes (inter and intra-node) using Infiniband.

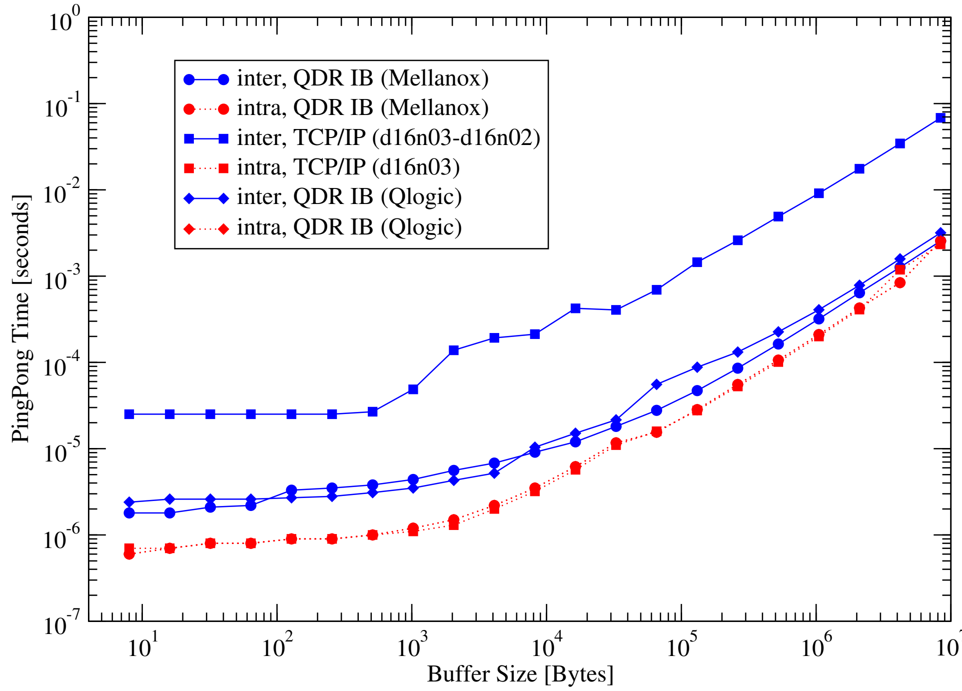
b. U2 nodes (inter and intra-node) using gigabit ethernet.

Note that the conventional way of implementing ping-pong uses two MPI processes, each of which does a send/recv pair, and then the single transit time is half the measured elapsed time. Also note that there are lots of ping-pong benchmark codes available in various places - if you prefer to use one of them rather than develop your own, just make sure that you provide an appropriate citation.

Solution:

The ping-pong benchmark code is given below. Note that I have added some extra statistics to the time of flight data, so we could actually plot some error bars if we wanted to (they tend not to show up very well on the scales that we are looking at). The plots for the various combinations look like the following.

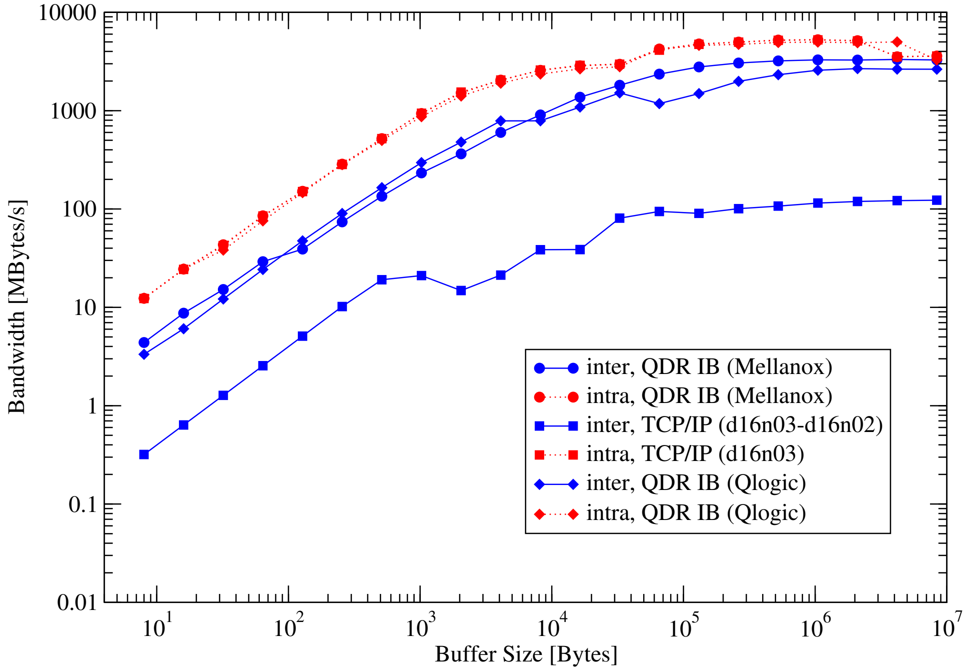
Figure 2: Message times for ping-pong using both flavors of QDR Infiniband and TCP/IP, both inter-node and intra-node.



In terms of the message times in Fig. 2, we see that the intra-node performance (which gets implemented using some form of shared memory) is generally quite superior to that of the inter-node (external network), and it naturally ignores which network protocol we are using (since it does not need to use the network). The performance characteristics of both flavors of QDR Infiniband (Mellanox IB cards tend to have a slightly better latency), however, are excellent compared to TCP/IP (in this case over gigabit Ethernet). Note that, using Intel MPI, you can force use of TCP/IP simply by setting `I_MPI_FABRICS_LIST=tcp` in your environment (otherwise, by default it has a list of protocols ranked in order of the best performing).

The bandwidth plot in Fig. 3 shows a similar tendency to that of the message times (no great surprise given the relation between the two).

Figure 3: Bandwidth for ping-pong on various nodes and interconnects



The zero-size message time (latency) is easy enough to read from the data (no need for a fancy extrapolation), and is summarized in the following table.

Table 1: MPI Latencies

Platform	Network	Intra-node	Inter-node
U2	QDR IB (Mellanox)	0.6μs	1.8μs
U2	QDR IB (Qlogic)	0.6μs	2.4μs
U2	QDR IB (TCP/IP)	0.6μs	25μs

```fortran
MODULE myconst
  integer,parameter :: sp=KIND(1.0),  &
       dp=SELECTED_REAL_KIND(2*PRECISION(1.0_sp)),    &
       sc=KIND((1.0,1.0)),                            &
       dc=SELECTED_REAL_KIND(2*PRECISION(1.0_sc))
END MODULE myconst

PROGRAM PP
  USE MPI
  USE myconst
  implicit none
!  include "mpif.h"

  ! max number of 8B reals in buffer
  integer,parameter :: MAX_BUFFER_LENGTH=1048576,Nensemble=2
  real(kind=dp) :: d8_buffer(MAX_BUFFER_LENGTH)

  integer :: Nrepeats,length_buffer
  integer :: i_repeat,i_ensemble
  real(kind=dp) :: guess
  real(kind=dp) :: time_start,time_end,time_delta,time_max,time_min,time_ave,&
       time_sigma,sum_time,sum_time2
  real(kind=dp) :: throughput_ave,throughPut_sigma,throughput_min, &
       throughput_max

  integer :: gdbWait=0
  integer myid,Nprocs,ierr,mpi_procname_length
  integer :: status(MPI_STATUS_SIZE)
  character(len=MPI_MAX_PROCESSOR_NAME) :: mpi_procname

  !
  ! Initialize communicator, check that we are using only 2p
  !
  CALL MPI_INIT(ierr)
  if (ierr /= 0) then
     print*, 'Unable to intialize MPI.'
     STOP
  end if
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,Nprocs,ierr)
  ! dummy pause point for gdb insertion
  !do while (gdbWait /=1)
  !end do
  if (Nprocs /= 2) then
     print*, 'This is PingPong between 2 procs, not ',Nprocs
     STOP
  end if
  CALL MPI_GET_PROCESSOR_NAME(mpi_procname,mpi_procname_length,ierr)
  write(*,'("Hello from proc ",i2," of ",i2,a32)') myid,Nprocs,&
       mpi_procname(1:mpi_procname_length)
  length_buffer = 1
  CALL MPI_BARRIER(MPI_COMM_WORLD,ierr) ! consistent stdout
  CALL FLUSH() ! force stdout to flush buffers - requires fortran 2003!
  if (myid.eq.0) then
     write(*,'("Number Averaged for Sigmas: ",i6)') Nensemble
     write(*,'(a7,1x,4a10,2x,a9,3a9,1x,a8)') "Len(B)","AVE(s)","SIGMA(s)",&
          "MIN(s)","MAX(s)","AVE(MB/s)","SIGMA","MIN","MAX","Nrepeats"
  end if
  do ! loop over buffer length
     !
     ! Try to measure something on the order of at least
     !  1-2 seconds for each run in the ensemble
     ! guess is 10 microsends + length/25 MB/s
     !
     guess = 0.00001_dp + DBLE(8*length_buffer)/(25.0_dp*1048576.0_dp)
     Nrepeats = 2.0_dp/guess
     time_min = 1.e8
     time_max = -1.e8
     sum_time = 0.0_dp
     sum_time2 = 0.0_dp

     if(length_buffer > MAX_BUFFER_LENGTH) exit
     do i_ensemble=1,Nensemble
        time_start = MPI_WTIME()
        if (myid == 0) then
           do i_repeat=1,Nrepeats
              CALL MPI_SEND(d8_buffer,length_buffer,MPI_DOUBLE_PRECISION, &
                   1,i_ensemble,MPI_COMM_WORLD,ierr)
              CALL MPI_RECV(d8_buffer,length_buffer,MPI_DOUBLE_PRECISION, &
                   1,i_ensemble,MPI_COMM_WORLD,status,ierr)
           end do
           time_end = MPI_WTIME()
           time_delta = (time_end-time_start)/DBLE(Nrepeats)
           time_min = MIN(time_min,time_delta)
           time_max = MAX(time_max,time_delta)
           sum_time = sum_time + time_delta
           sum_time2 = sum_time2 + time_delta**2
        else
           do i_repeat=1,Nrepeats
              CALL MPI_RECV(d8_buffer,length_buffer,MPI_DOUBLE_PRECISION, &
                   0,i_ensemble,MPI_COMM_WORLD,status,ierr)
              CALL MPI_SEND(d8_buffer,length_buffer,MPI_DOUBLE_PRECISION, &
                   0,i_ensemble,MPI_COMM_WORLD,ierr)
           end do
        end if
     end do
```

```fortran

      ! note factor of two for half the round-trip time
      time_min = 0.5_dp*time_min
      time_max = 0.5*time_max
      time_ave = 0.5_dp*sum_time/DBLE(Nensemble)
      time_sigma = SQRT( (0.25_dp*sum_time2/DBLE(Nensemble) - time_ave**2)/ &
           DBLE(Nensemble-1) )
      !
      ! Output results
      if (myid == 0) then
         throughput_ave = length_buffer*8*0.000001_dp/time_ave
         throughput_min = length_buffer*8*0.000001_dp/time_max
         throughput_max = length_buffer*8*0.000001_dp/time_min
         throughput_sigma = throughput_ave*time_sigma/time_ave
         write(*,'(i7,1x,4f10.7,2x,f9.3,3f9.3,1x,i8)') 8*length_buffer, &
              time_ave,time_sigma, time_min, time_max, throughput_ave, &
              throughput_sigma, throughput_min, throughput_max, Nrepeats
      end if
      length_buffer = length_buffer*2
   end do ! loop over buffer length
   CALL MPI_FINALIZE(ierr)
END PROGRAM PP
```