ASSIGNMENT 3 SOLUTIONS
HPC1 Fall 2013

Due Date: ***Thursday, October 17***
(please submit your report electronically to the instructor via email, as one PDF file named *hw3-yourUBitname.pdf*)

Problem 1: Write a program to compute π by the summation:

$$\frac{\pi}{4} = \sum_{i=0}^{N\to\infty} \frac{(-1)^i}{2i+1},$$

and use **OpenMP** to parallelize the code. Determine the performance of your code as a function of N terms in the sum, and N_p processors. Note that, depending on the granularity of your timer, it may well be necessary to repeat the calculation (say j times, such that jN is a convenient timing interval) and time the total to get reliable average times, especially for smaller values of N. Plot the execution time, parallel speedup, and parallel efficiency as a function of N_p (note the utility of logarithmic scales!). Make careful note of what machine type you are using to perform this study, as for best comparative results versus **MPI** you will want to be consistent in your choice for the second problem.
Solution:

In this problem we are asked to compute the value of π in parallel using partial sums and the summation:

$$\frac{\pi}{4} = \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{(2n-1)}, \tag{1}$$

which we can easily break up in parallel - let i be the processor index, N the total number of terms in the full sum, then the partial sum for processor i becomes:

$$\frac{\pi^{(i)}}{4} \simeq \sum_{j=i(N/N_p)+1}^{(i+1)(N/N_p)} \frac{(-1)^{j+1}}{(2j-1)}, \qquad i \in [0, N_p - 1] \tag{2}$$

where N_p is the number of parallel processes, and we will enforce the rule that N can be divided exactly by N_p, i.e. $\mathrm{mod}(N, N_p) = 0$. Using **OpenMP** you can simply parallelize the summation with a *reduction* clause on the `parallel do` construct, as in the following code.

N	Sequential Time [sec]
128	0.70389E-06
1024	0.50578E-05
1024×10^1	0.50069E-04
1024×10^2	0.49685E-03
1024×10^3	0.49889E-02
1024×10^4	0.49770E-01
1024×10^5	0.49793E+00
1024×10^6	0.49792E+01

Table 1: Optimal sequential times on the 8-core 2.26GHz nodes, Intel compiler 12.1 used with OpenMP turned off, `-O3 -vec_report3 -fpp` options

Table 1 gives the optimal sequential times for various term counts that we will use for measuring parallel speedup (recall that the numerator in the expression for speedup is the optimal sequential time, **not** the parallel time on a single thread or single process). For OpenMP there is little overhead, so the sequential time is only about 0.5% faster on the smaller number of terms. It will be more significant in the MPI case.

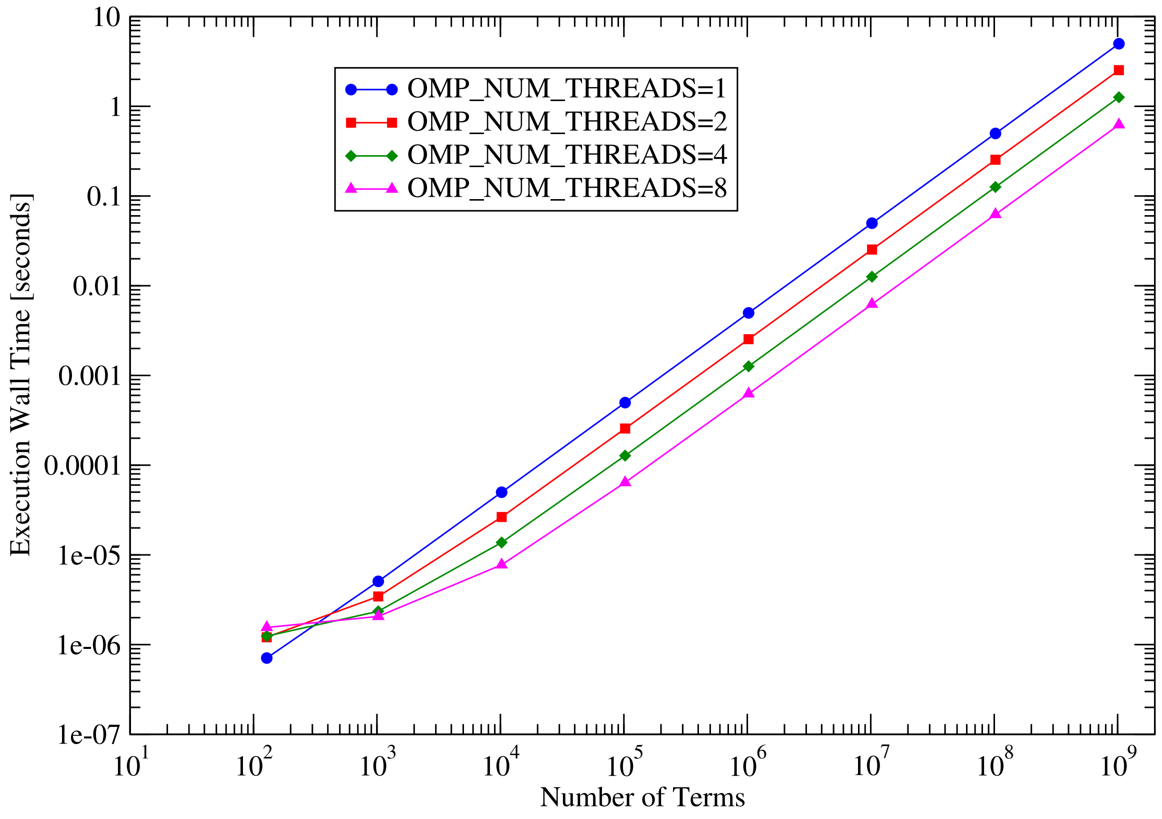

Figure 1: OpenMP execution time as a function of N and `OMP_NUM_THREADS` on a dedicated 8-core L5520 2.26GHz compute node in **UB/CCR**.

Figure 1 shows the execution times as a function of N and `OMP_NUM_THREADS` on a dedicated 8-core L5520 2.26GHz compute node in the **UB/CCR** cluster, while Figure 3 shows the parallel speedup and efficiency.

Note that in the execution times there is only a relatively substantial overhead for OpenMP when dealing with smaller numbers of terms in the sum. Overall the parallel speedup for the same values of N) is excellent, gaining near 100% efficiency for $N > 102400$. We will have an opportunity to compare the overhead with a message- passing implementation in the next problem.

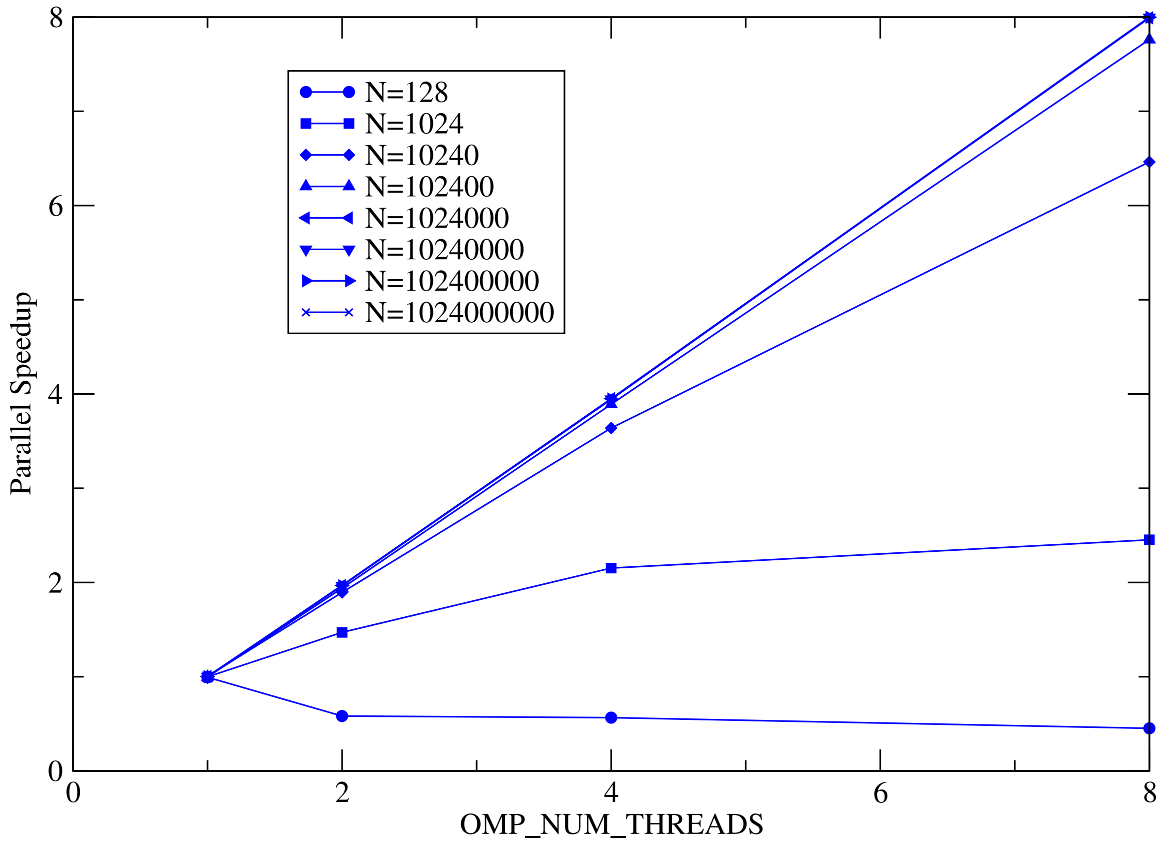

Figure 2: OpenMP parallel speedup as a function of N and `OMP_NUM_THREADS` on a dedicated 8-core 2.26GHz compute node in **UB/CCR**.

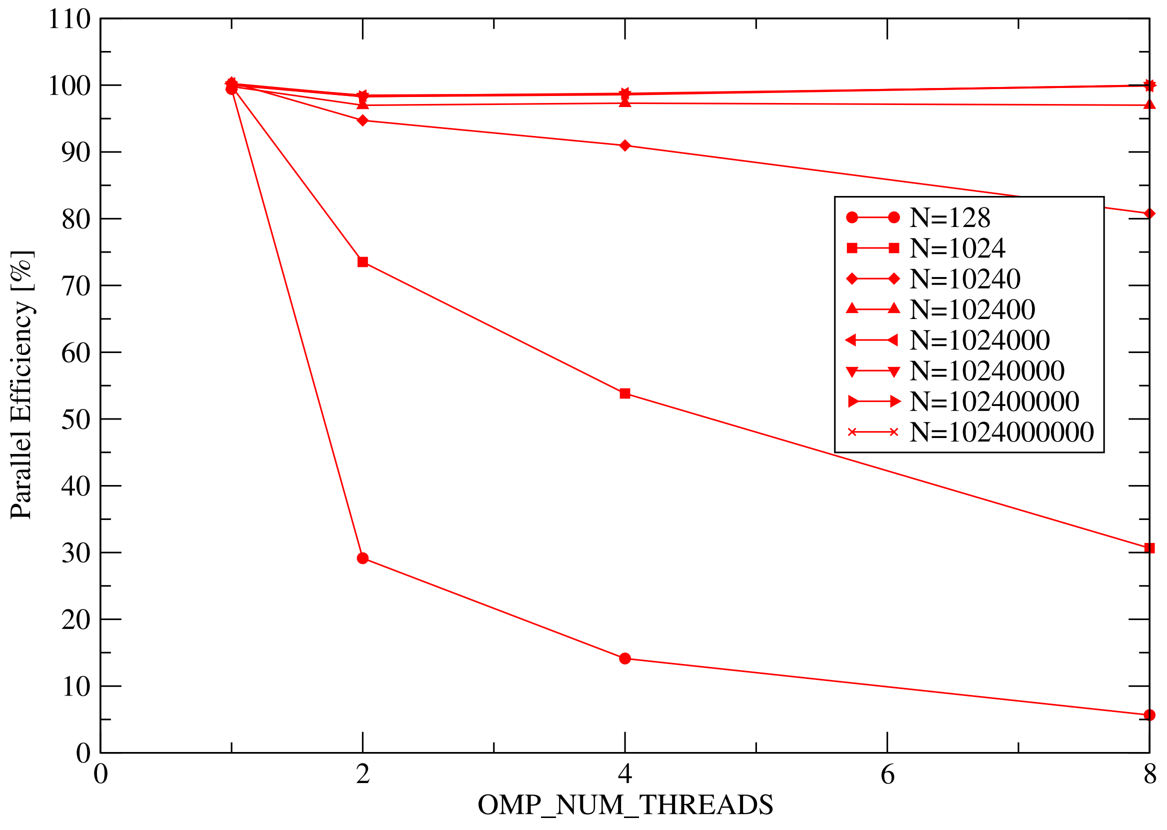

Figure 3: OpenMP parallel speedup and efficiency as a function of N and `OMP_NUM_THREADS` on a dedicated 8-core L5520 2.26GHz compute node in **UB/CCR**.

```fortran
MODULE myconst
  integer,parameter :: sp=KIND(1.0),dp=SELECTED_REAL_KIND(2*PRECISION(1.0_sp)), &
       si=KIND(1),di=SELECTED_INT_KIND(2*RANGE(1_si))
  real(kind=dp),parameter :: pi=3.14159265358979323846264З_dp,pi_over_4 = pi/4.0_dp
END MODULE myconst

PROGRAM PIomp
  !
  ! program to use partial sums to compute pi using OpenMP
  !
!$  USE omp_lib
  USE myconst
  implicit none
  !
  integer (kind=di) :: Nterms   ! number of terms in full sum
  integer (kind=di) :: Nruns
  !
  integer (kind=di) :: i,my_low,my_high,my_sign,Nreps,Nperproc
  integer :: irep,irun
  real(kind=dp) :: partial_sum_p,partial_sum_m,sum,time_delta,t0,t1
  !
  ! MPI
  !
  integer myid,Nprocs

  real(kind=dp) :: dsecnd
  external dsecnd

  print*, 'range of si,di = ',RANGE(myid),RANGE(Nterms)
  !
  ! Do a whole series for Nprocs determined by 10^{i+2}, i=1,Nruns
  !
  myid = 0
  Nprocs = 1
  Nruns = 8
  Nterms=128
  if (myid == 0) then
     write(*,'(2a12,a9,2a13)') "Nterms","Nperproc","Nreps","error","time/rep"
  end if

  do irun =1,Nruns
     !
     ! make sure that Nterms%Nprocs == 0
     !
     ! Nreps - if summing each term takes, say, 10 flop/s, we need
     !          on the order of 10^9 terms to get a few seconds of
     !          computation
     Nreps = 100000000/Nterms
     if (Nreps == 0) Nreps = Nreps+1
     !
     !
     !
     time_delta = 0.0
     do irep = 1,Nreps
        partial_sum_p = 0.0_dp
        partial_sum_m = 0.0_dp
        !
        !$OMP PARALLEL DEFAULT(SHARED) PRIVATE(i,myid,Nprocs) &
        !$OMP REDUCTION(+:partial_sum_p,partial_sum_m)
        !
!$      myid = OMP_GET_THREAD_NUM()
!$      Nprocs = OMP_GET_NUM_THREADS()
        Nperproc = Nterms/Nprocs
        Nterms = Nterms + MOD(Nterms,Nprocs)
#ifdef _OPENMP
        t0 = OMP_GET_Wtime()
#else
        t0 = dsecnd()
#endif
        !
        ! Note that I sum alternating terms separately to avoid
        !  roundoff errors from adjoining terms and to facilitate
        !  vectorization of this loop
        !
        !$OMP DO
        do i=1,Nterms,2
           partial_sum_p = partial_sum_p + 1.0_dp/(2.0_dp*i-1.0_dp)
           partial_sum_m = partial_sum_m - 1.0_dp/(2.0_dp*i+1.0_dp)
        end do
#ifdef _OPENMP
        t1 = OMP_GET_Wtime()
#else
        t1 = dsecnd()
#endif
        !
        !$OMP END PARALLEL
        !
        sum = partial_sum_p + partial_sum_m
        time_delta = time_delta + (t1-t0)
     end do
     time_delta = time_delta/Nreps
     if (myid == 0) then
        write(*,'(2i12,i9,2e13.5)') Nterms,Nperproc,Nreps,sum-pi_over_4,time_delta
     end if
     if (irun > 1) then
        Nterms = Nterms*10
     else
        Nterms = Nterms*8
     endif
  end do
END PROGRAM PIomp
```

Problem 2: Repeat problem 1 using **MPI** instead of **OpenMP**.

Hint: simple pseudo-code for splitting the sum into P partial sums might look something like:

```
myID = MyProcNumber()
Np = TotalProcNumber()
mySum = 0
do i=myID*(NsumTerms/Np)+1,(myID+1)*(NsumTerms/Np),2
    mySum = mySum + 1.0/(2*i-1)
    mySum = mySum - 1.0/(2*i+1)
end do
CollectPartialSums(S)
```

Solution:

We are asked to perform the same computation of π as in the previous problem, but now using message passing (MPI). The final sum is carried out by a single call to `MPI_REDUCE`, otherwise the work can be divided nearly perfectly into the number of processes. So a simple performance model would look like:

$$\tau(N, N_p) = \tau_{red}(8B, N_p) + \tau_{fpdiv}\frac{N}{N_p} \tag{3}$$

where τ_{fpdiv} is the cost of the floating point addition and division in the computation of the partial sums, and $\tau_{red}(8B, N_p)$ is the cost of the reduction, which we can approximate as

$$\tau_{red}(8B, N_p) \simeq \tau_{lat}\log_2(N_p). \tag{4}$$

So in this case we have a speedup,

$$S(N, N_p) = N_p\left[1 + \left(\frac{\tau_{lat}}{N\tau_{fpdiv}}\right)N_p\log_2 N_p\right]^{-1} \tag{5}$$

and the efficiency is then just

$$\mathcal{E}(N, N_p) = \left[1 + \left(\frac{\tau_{lat}}{N\tau_{fpdiv}}\right)N_p\log_2 N_p\right]^{-1} \tag{6}$$

Running the code for N in powers of ten from 10^3 to 10^{10} results in Figure 4.

Note that the MPI execution times are similar to the times we saw for the OpenMP code, but with a little more overhead on the smaller sums. Plotting the speedup will give us a little more insight into any subtleties in the behavior.

From the figure, we can easily discern the limits of our performance model - for small N, the communication costs dominate, and we see little to no speedup. On the opposite side, for large N, the communication costs are relatively minimized, and we observe a near-linear speedup. We could use our performance model to justify this assertion more quantitatively, and this behavior is well illustrated in Figures 4-6.

Note that MPI communication overhead dominates for the lowest range in the number of terms, and grows steadily worse (as you would expect) as the number of processes increases. Compare with the **OpenMP** implementation in the previous problem, which is able to achieve performance gains on smaller amounts of computation, thanks to lower communication overhead.

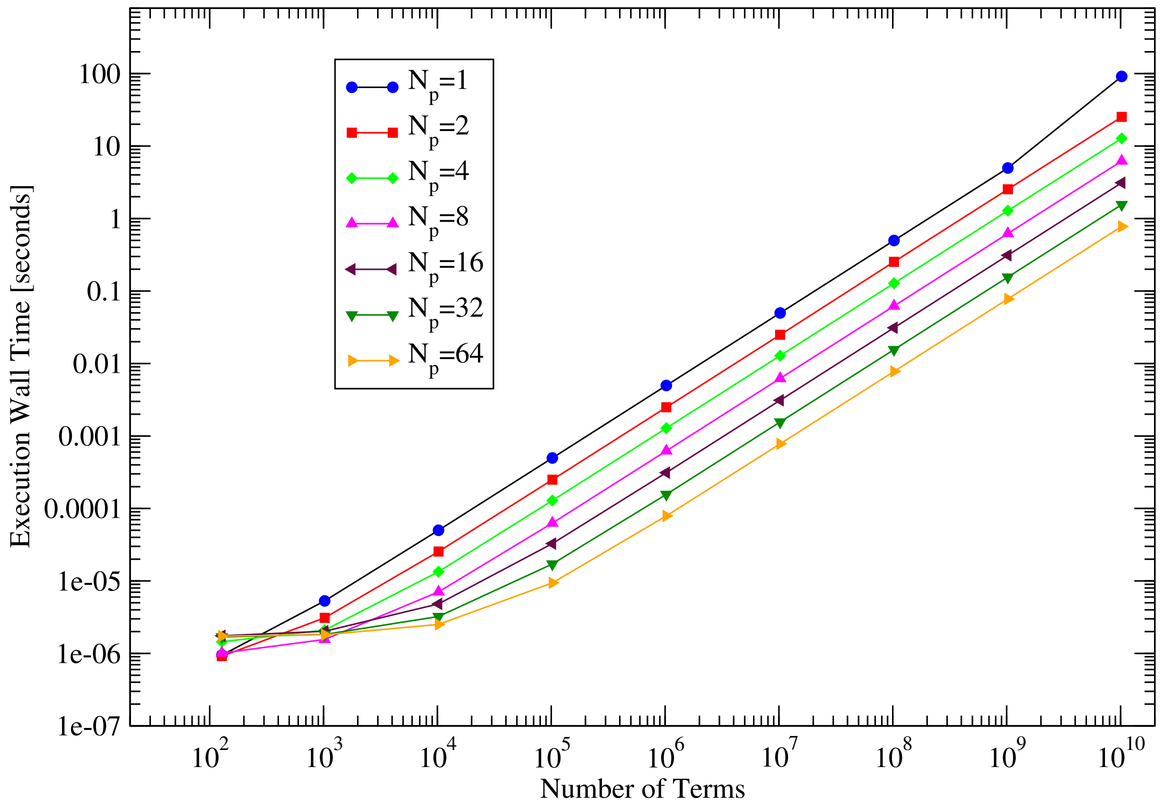

Figure 4: Execution time for calculation of π using MPI on the 8-core L5520 2.26GHz nodes in **UB/CCR**.

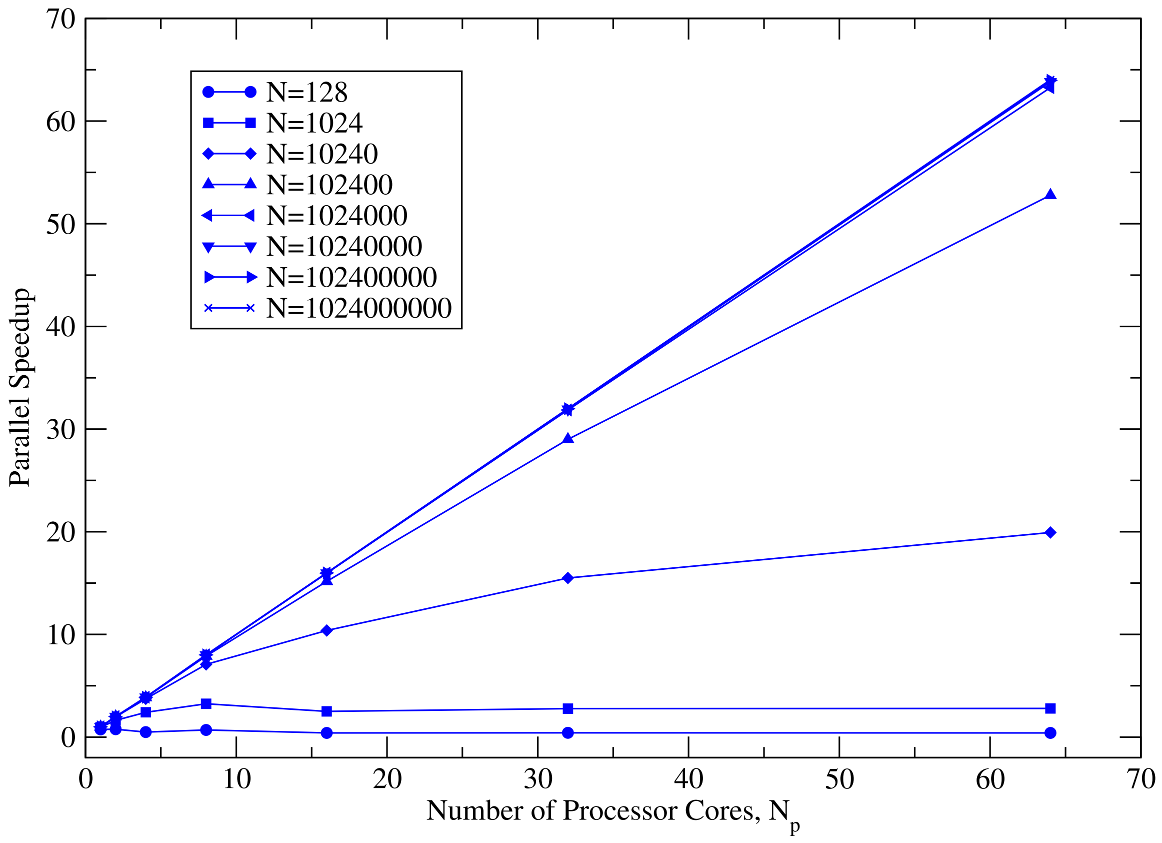

Figure 5: Parallel speedup for calculation of π using MPI on the 8-core L5520 2.26GHz nodes in **UB/CCR**.

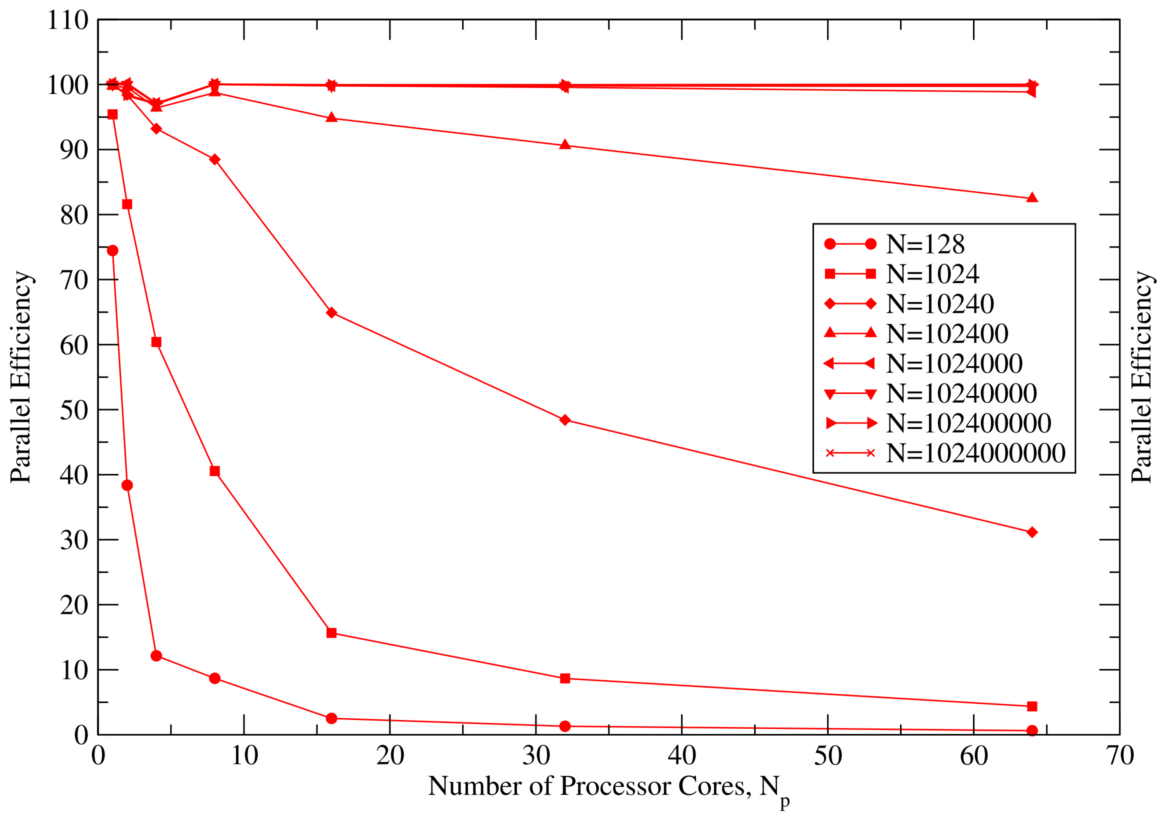

Figure 6: Parallel efficiency for calculation of π using MPI on the 8-core L5520 2.26GHz nodes in **UB/CCR**. Note that the efficiency is significantly impacted on smaller sums, even on a single MPI process.

```
MODULE myconst
  integer,parameter :: sp=KIND(1.0),dp=SELECTED_REAL_KIND(2*PRECISION(1.0_sp)), &
       si=KIND(1),di=SELECTED_INT_KIND(2*RANGE(1_si))
  real(kind=dp),parameter :: pi=3.141592653589793238462643_dp,pi_over_4 = pi/4.0_dp
END MODULE myconst

PROGRAM PImpi
  !
  ! program to use partial sums to compute pi
  !
  USE MPI
  USE myconst
  implicit none
  !
  integer (kind=di) :: Nterms   ! number of terms in full sum
  integer (kind=di) :: Nruns
  !
  integer (kind=di) :: i,my_low,my_high,my_sign,Nreps,Nperproc
  integer :: irep,irun
  real(kind=dp) :: sum,partial_sum,partial_sum_p,partial_sum_m,time_delta,t0,t1
  !
  ! MPI
  !
  character(len=MPI_MAX_PROCESSOR_NAME) :: procname
  integer myid,Nprocs,length_procname,ierr

  real(kind=dp) :: dsecnd
  external dsecnd

  CALL MPI_INIT(ierr)
  if (ierr /= 0) then
     print*, 'Unable to MPI_Init'
     STOP
  end if
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,Nprocs,ierr)
  CALL MPI_GET_PROCESSOR_NAME(procname,length_procname,ierr)
  write(*,'(a24,i4,a4,i4,1x,a32)') 'Greetings from proc ',myid,' of ',Nprocs,procname

  !print*, 'range of si,di = ',RANGE(myid),RANGE(Nterms)
  !
  ! Do a whole series for Nprocs determined by 10^{i+2}, i=1,Nruns
  !
  Nruns = 9
  Nterms=128
  if (myid == 0) then
     write(*,'(2a12,a9,2a13)') "Nterms","Nperproc","Nreps","error","time/rep"
  end if

  do irun =1,Nruns
     !Nterms = Nterms*10
     !
     ! make sure that Nterms%Nprocs == 0
     !
     !
     ! Nreps - if summing each term takes, say, 10 flop/s, we need
     !          on the order of 10^7 terms to get a few seconds of
     !          computation
     Nreps = 100000000/Nterms
     if (Nreps == 0) Nreps = Nreps+1
     !
     !
     !
     time_delta = 0.0
     Nperproc = Nterms/Nprocs
     my_low=myid*Nperproc+1
     my_high=(myid+1)*Nperproc
     if (MOD(Nterms,Nprocs).ne.0) then
        if (myid.eq.0) write(*,*) 'Error, leftovers!'
        CALL MPI_Abort(MPI_COMM_WORLD,1,ierr)
     endif
     do irep = 1,Nreps
        t0 = MPI_Wtime()
        ! to take advanatge of vectorization, we split the sums
        !  to avoid vector dependence
        partial_sum_p = 0.0_dp
        partial_sum_m = 0.0_dp
        do i=my_low,my_high,2
           partial_sum_p = partial_sum_p + 1.0_dp/(2.0_dp*i-1.0_dp)
           partial_sum_m = partial_sum_m - 1.0_dp/(2.0_dp*i+1.0_dp)
        end do
        partial_sum = partial_sum_p + partial_sum_m
        CALL MPI_REDUCE(partial_sum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, &
             MPI_COMM_WORLD,ierr)
        t1 = MPI_Wtime()
        time_delta = time_delta + (t1-t0)
     end do
     time_delta = time_delta/Nreps
     if (myid == 0) then
        write(*,'(2i12,i9,2e13.5)') Nterms,Nperproc,Nreps,sum-pi_over_4,time_delta
     end if
     if (irun>1) then
       Nterms = Nterms*10
     else
       Nterms = Nterms*8
     endif
  end do
  CALL MPI_FINALIZE(ierr)
END PROGRAM PImpi
```