(HPSC **5576** ELIZABETH JESSUP)

HIGH PERFORMANCE SCIENTIFIC COMPUTING

:: Homework / 8

:: Student / Florian Rappl

1 problem / **10** points

Problem 1

Task:

Write a short program demonstrating the use of MPE's profiling interface, *10 pts*

Use MPE directives to instrument your implementation of *MPI_Allreduce* from the Week 3 homework. Create custom regions for the following phases of your program, if applicable:

- Memory (malloc calls, if used)
- Communication (send/receive regions)
- Computation (loops summing vectors)

Compile and link using MPE including the library that logs the transmission of all MPI messages. (In your resulting MPE plots, you should see your colored regions, colored regions for MPI calls, and arrows for messages.)

In the week 3 homework, you were asked to test low-to-high and high-to-low bit traversals. You may just choose one here. If you run on a remote system and scp the .clog* file to your laptop, you will need to find the visualization toolkit of the same version to be able to open the log file for analysis.

The SLOG SDK releases are at: ftp://ftp.mcs.anl.gov/pub/mpi/slog2/

The version on Frost uses an intermediate build. If you run on Frost, you'll need the full MPE source containing the matching SLOG SDK: ftp://ftp.mcs.anl.gov/pub/mpi/mpe/mpe2-1.0.4.tar.gz

Solution:

I did use Frost again in order to simplify the comparison process with my former implementation. I used the [**L->H**] (Low to High) bit traversal. To include MPE in my project I had to do the following:

include flags	-I /contrib/bgl/mpe2/include
link flags	-L /contrib/bgl/mpe2/lib -llmpe –lmpe

Therefore I compiled it using the following two commands:

```
mpixlc -g -I /contrib/bgl/mpe2/include -c -o allmpe.o all_mpe.c
mpixlc -o all_mpe allmpe.o -g -L /contrib/bgl/mpe2/lib -llmpe -lmpe
```

In order to distinguish between with and without MPE I built in another command line argument, which sets a bit flag to 1 if detected. The MPE commands will only be executed if the flag is at 0. For evaluating the *.clog2 files I used Jumpshot-4.

Question:

Run your *Allreduce()* routine for two data sizes -- 8 doubles, and 1MB worth of doubles -- on 8 or 16 cores. Produce MPE plots showing the behavior of the program for both reduction operations. Can you identify the butterfly structure of the communication in the MPE plots?

Answer:

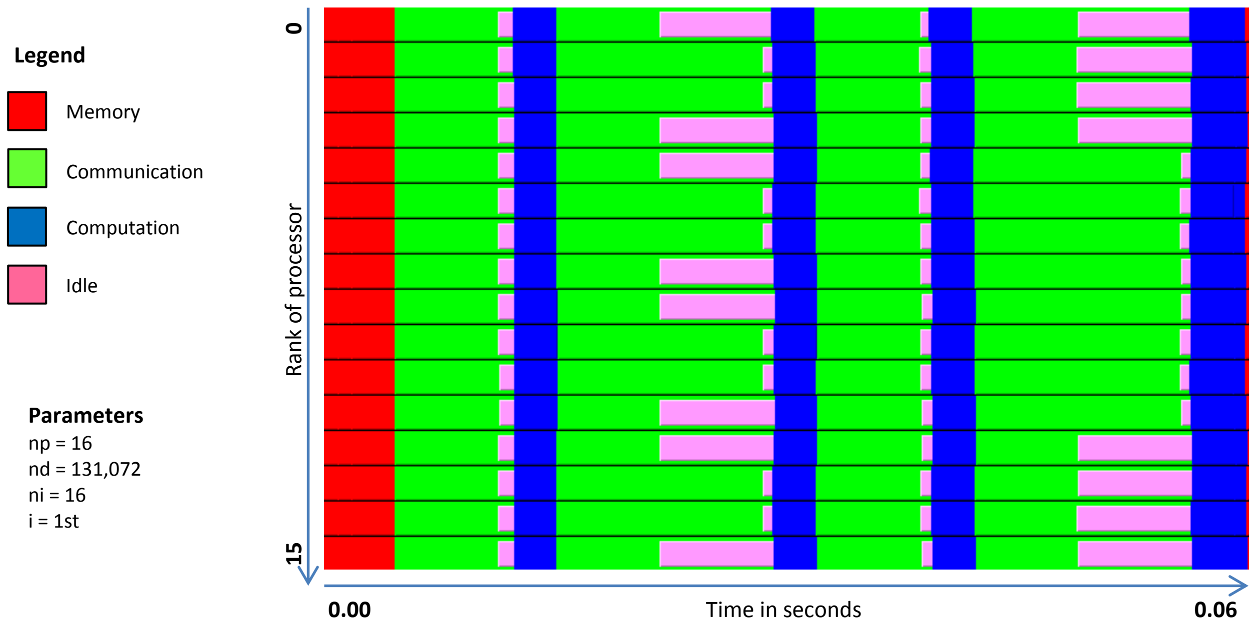

Fig. 1 Measurement with 16 processors and 131072 doubles (1 MB) running one iteration

This plot shows 16 processors performing the *Allreduce()* function (one time). The Butterfly topology is here perfectly visible (and the disadvantage of the Butterfly on the BlueGene). We see that first only next neighbor communication is performed, therefore all arriving at the same time. After that first blue line we see that communication takes longer for 1-3, 4-6 etc. whereas the communication for the others like 0-2, 5-7 etc. is as short as in the previous phase. This is obviously one of the features of BlueGene's Torus topology, with is not optimized for Butterfly (better said Butterfly is not optimized for the Torus) and therefore produces contention on the wires in this phase. The 3rd phase offers again quite short times. Therefore we cannot do any conclusions here only that this phase is better optimized for the Butterfly topology than the previous and the next one. But in the fourth phase we see that obviously the ends are communicating, i.e. those are being slower due to contention on the wires.

Note 1: I used the first iteration for this plot. The other plots have a smaller memory time-block, which is nearly as small as a computation time-block.

Note 2: With arrow activated for the messages the butterfly structure is even easier to see. However in my opinion the white-arrows are quite hard to see on a printout. I've included screenshots in the *.tar.zip file.

The next plot shows the same (first) iteration with less data, i.e. just 8 doubles. We recognize a lot of black regions, which are obviously something like idle time but only in sense of MPE not knowing what to log. I suppose those things are due to the case selections, i.e. the "*if*" statements enabling or disabling MPE, and the MPE calls (to get an event ID using *MPE_Log_get_event_number()*). Since the time scale on this plot is much smaller we can now see those regions whereas in the plot before we could not.

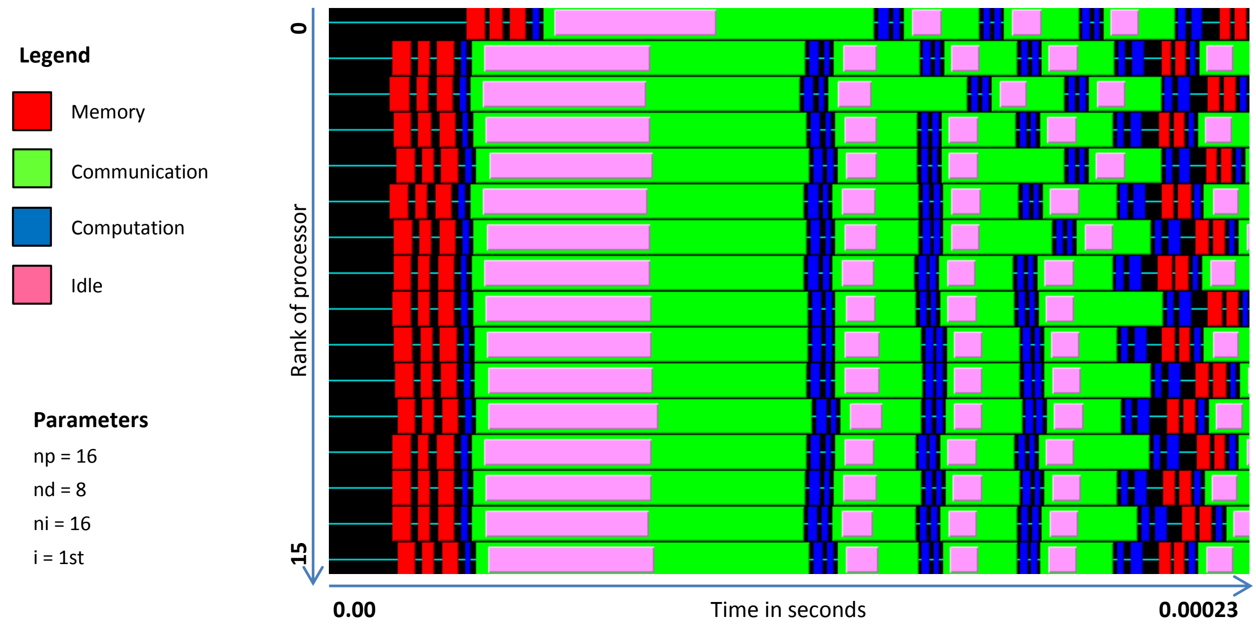

Fig. 2 Measurement with 16 processors and 8 doubles running one iteration

Another interesting thing is the shift of process number 0 (the master process). This shift is due to the *MPE_Describe_state()* function calls. Therefore we can clearly see what impact MPE already has on our program. We can also see that this impact is not scaling, since we did not observe such black areas in the plot before (not because they were not there but just because the time scaling was higher and therefore the region too small). Again we can see the butterfly but it is much harder to see when the message size is very small and only latency (which is kind of constant) plays a role. We only observe that all the receive partners from process 0 get stuck behind, just because that process is behind from the beginning. Therefore after the last phase half of the processes are also behind (kind of synchronized with process 0), whereas the other half is still looking quite good. We can also see that we have a lot of idle time in the very first message sending / receiving call. This alone confirms my previous homework sets in that sense that warm up runs are indeed necessary since the other phases have less idle time after the sending statement and also perform the receive much faster.

Question:

Run your program with and without MPE, timing the *Allreduce()* execution time. Does the inclusion of MPE profiling reduce performance?

Answer:

Indeed it does. I ran the program with the same setup (number of processors: 16, number of doubles: 8, number of iterations: 16) two times. The output is displayed in the next section. The total difference for this setup was $3 \cdot 10^{-4}$ s or nearly 50% of the execution time without MPE. Overall in the program that included MPE we can say that the MPE instructions made like 30% of the whole program execution time. This is indeed a reduced performance, but since we can assume that the

MPE time does not scale linearly with the program, the bigger a program is the less performance MPE consumes in comparison to the main program.

Program output:

```
The final result is 960.000000
PROCESSORS:        16
VECTORLENGTH:      8
ITER 16     TIME 1.002204e-03       MPEP ON
Disabling the clock synchronization...
---
The final result is 960.000000
PROCESSORS:        16
VECTORLENGTH:      8
ITER 16     TIME 7.105557e-04       MPEP OFF
Disabling the clock synchronization...
```

The first output is with MPE active, while the second one used the "*–wompe*" command line argument of the program to disable the execution of MPE profiling.

Code printout

```
#define ITERATIONS 16
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "mpi.h"
/* global logging event variables                          */
int   eventMem_s, eventMem_e;        /* for memory          */
int   eventComm_s, eventComm_e;      /* for communication   */
int   eventComp_s, eventComp_e;      /* for computation     */
/* prototype of (slightly modified) my_allreduce function  */
void my_allreduce(double* sndvalue, double* recvalue, int count, unsigned
            int rank, int processors, int tag, MPI_Comm comm, int wo_mpe);

int main(int argc, char* argv[]) {
      int         i, j;              /* loop counters       */
      int         my_rank;           /* rank of process     */
      int         p;                 /* number of processes */
      int         tag = 0;           /* tag for messages    */
      int         count = 1;         /* vector size         */
      double      *sndvec;           /* send vector         */
      double      *recvec;           /* receive vector      */
      double      final = 0.0;       /* final result        */
      double      starttime;         /* timing starttime    */
      double      endtime;           /* timing endtime      */
      int         wo_mpe = 0;        /* without mpe flag    */
      /* Command line args parser (shortened for -n & -wompe)*/
      for(i = 1; i < argc; i++)
      {
            //[...] same as in Homework 3 - now only additional:
            else if(strcmp(argv[i], "-wompe") == 0)
                  wo_mpe = 1;
      }
      /* Start up MPI */
      MPI_Init(&argc, &argv);
      starttime = MPI_Wtime();
      if(wo_mpe == 0)
      {
```

```c
        /* Start up MPE */
        MPE_Start_log();
        /* Getting and setting some MPE_Log numbers */
        eventMem_s = MPE_Log_get_event_number();
        eventMem_e = MPE_Log_get_event_number();
        eventComm_s = MPE_Log_get_event_number();
        eventComm_e = MPE_Log_get_event_number();
        eventComp_s = MPE_Log_get_event_number();
        eventComp_e = MPE_Log_get_event_number();
    }

    /* Find out process rank  */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    /* Find out number of processes */
    MPI_Comm_size(MPI_COMM_WORLD, &p);
    /* Check if processor count is %2 - else finish */
    if(p % 2 != 0)
    {
        if(my_rank == 0) printf("Limited to np = 2^n.");
    }
    else
    {
        /* setup the events */
        if(my_rank == 0 && wo_mpe == 0)
        {
            /* define log for M (memory) */
            MPE_Describe_state(eventMem_s, eventMem_e,
                                              "Memory", "red");
            /* define log for C (communication) */
            MPE_Describe_state(eventComm_s, eventComm_e,
                                              "Communication", "green" );
            /* define log for A (arithmetic) */
            MPE_Describe_state(eventComp_s, eventComp_e,
                                              "Computation", "blue" );
        }
        if(wo_mpe == 0)
            /* S,LOG(M) */
            MPE_Log_event(eventMem_s, 0,
                                  "Allocating + filling sendvector");
        /* create vector for sending data */
        sndvec = (double*)malloc(count * sizeof(double));
        /* filling the vector with data - my_rank */
        for(i = 0; i < count; i++)
            sndvec[i] = (double)my_rank;
        if(wo_mpe == 0)
            /* E,LOG(M) */
            MPE_Log_event(eventMem_e, 0,
                                  "Sendvector allocated + filled");
        /* measuring process */
        for(i = 0; i < ITERATIONS; i++)
        {
            final = 0.0;
            if(wo_mpe == 0)
                /* S,LOG(M) */
                MPE_Log_event(eventMem_s, 0,
                                      "Allocating + filling recvector");
            /* create and set the receive vector */
            recvec = (double*)malloc(count * sizeof(double));
            for(j = 0; j < count; j++)
                recvec[j] = 0.0;
            if(wo_mpe == 0)
```

```
                /* E,LOG(M) */
                MPE_Log_event(eventMem_e, 0,
                              "Recvector allocated + filled");
            /* call the specified function */
            my_allreduce(sndvec, recvec, count,(unsigned int)my_rank,
                              p, tag, MPI_COMM_WORLD, wo_mpe);
            if(wo_mpe == 0)
                /* S,LOG(A) */
                MPE_Log_event(eventComp_s, 0,
                              "Adding everything together");
            /* gather the data for the final result (chk) */
            for(j = 0; j < count; j++)
                final += recvec[j];
            if(wo_mpe == 0)
                /* E,LOG(A) */
                MPE_Log_event(eventComp_e, 0,
                              "Finished all additions");
            /* print out final result if all iterations done */
            if(my_rank == 0 && (i + 1) % ITERATIONS == 0)
                printf("The final result is %f\n", final);

            if(wo_mpe == 0)
                /* S,LOG(M) */
                MPE_Log_event(eventMem_s, 0,
                              "Clearing recvvector");
            /* clear memory */
            free(recvec);
            if(wo_mpe == 0)
                /* E,LOG(M) */
                MPE_Log_event(eventMem_e, 0, "Recvector cleared");
        }

        if(wo_mpe == 0)
            /* S,LOG(M) */
            MPE_Log_event(eventMem_s, 0, "Clearing sendvector");
        free(sndvec);

        if(wo_mpe == 0)
            /* E,LOG(M) */
            MPE_Log_event(eventMem_e, 0, "Sendvector cleared");
        if(my_rank == 0)
        {
            printf("PROCESSORS:\t%d\n", p);
            printf("VECTORLENGTH:\t%d\n", count);
        }
    }

    if(wo_mpe == 0)
        /* Shut down MPE */
        MPE_Stop_log();
    endtime = MPI_Wtime();

    if(my_rank == 0)
        printf("ITER %d\tTIME %e\tMPEP %s\n", ITERATIONS,
            endtime - starttime, wo_mpe == 0 ? "ON" : "OFF");

    /* Shut down MPI */
    MPI_Finalize();

    return 0;
} /* main */
```

```

void my_allreduce(double* sndvalue, double* recvalue, int count, unsigned
      int rank, int processors, int tag, MPI_Comm comm, int wo_mpe) {
      /* is only designed for L->H (up) in no verbose */
      int                 i, j;        /* Loop counters */
      unsigned int       mask = 1;     /* the bit mask  */
      unsigned int       dest = 0;     /* destination   */
      MPI_Status         status;       /* status buffer */
      double             *tmpvalue;    /* temporary vec */
      if(wo_mpe == 0)
            /* S,LOG(M) */
            MPE_Log_event(eventMem_s, 0, "Allocating and filling Tempvec");
      /* allocate memory for receive vector */
      tmpvalue = (double*)malloc(count * sizeof(double));
      /* get the receive vector set up */
      for(j = 0; j < count; j++)
            recvalue[j] += sndvalue[j];
      if(wo_mpe == 0)
            /* E,LOG(M) */
            MPE_Log_event(eventMem_e, 0, "Tempvec allocated and filled");
      for(i = 1; i < processors; i *= 2)
      {
            if(wo_mpe == 0)
                  /* S,LOG(A) */
                  MPE_Log_event(eventComp_s, 0, "Starting Bitshift");
            /* bit shift to det. the partner */
            dest = mask ^ rank;
            if(wo_mpe == 0)
            {
                  /* E,LOG(A) */
                  MPE_Log_event(eventComp_e, 0, "Bitshift ended");
                  /* S,LOG(C) */
                  MPE_Log_event(eventComm_s, 0, "Starting send/recv");
            }
            /* communication */
            MPI_Send(recvalue, count, MPI_DOUBLE_PRECISION, dest, tag,
                        comm);
            MPI_Recv(tmpvalue, count, MPI_DOUBLE_PRECISION, dest, tag,
                        comm, &status);
            if(wo_mpe == 0)
            {
                  /* E,LOG(C) */
                  MPE_Log_event(eventComm_e, 0, "Send/recv ended");
                  /* S,LOG(A) */
                  MPE_Log_event(eventComp_s, 0,
                                    "Adding values and bitshifting");
            }
            /* do the desired operation - in this case sum up */
            for(j = 0; j < count; j++)
                  recvalue[j] += tmpvalue[j];
            /* do the bit shift - for L->H (up) */
            mask = mask << (unsigned int)1;
            if(wo_mpe == 0)
                  /* E,LOG(A) */
                  MPE_Log_event(eventComp_e, 0,
                                    "Values added and bitshifted");
      }
} /* my_allreduce */
```