Introduction to HPC: Übung #2

Abgabe am Montag, 03. November 2014

Günther Schindler, Klaus Naumann, Christoph Klein

Inhaltsverzeichnis

MPI Ring Communication

```c
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main (int argc, char **argv) {

    int size, rank, m = 3000, signal = 666,i;
    double starttime, endtime;
    char hostname[50];
    MPI_Status status;

    MPI_Init (&argc, &argv);

    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);

    gethostname( hostname, 50);


    if (rank == 0 && size > 1) { /*at least two processes*/
        starttime = MPI_Wtime();
        for (i=0; i<m; i++) { /*main process sends the first m messages*/
            MPI_Send( &signal, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
        }
        for (i=0; i<m; i++) { // main process waits for the last m messages
            MPI_Recv ( &signal, 1, MPI_INT, size-1, 0, MPI_COMM_WORLD, &status);
        }
        endtime = MPI_Wtime();
        double dt = endtime - starttime;
        double N = m*size;
        double average = dt/N;
        printf( "%f %f\n", dt, average);
    }
    else if (size > 1) { // at least two processes
        int source = rank - 1;
        for (i=0; i<m; i++) { // side processes wait for m messages
            MPI_Recv ( &signal, 1, MPI_INT, source, 0, MPI_COMM_WORLD, &status);
        }


        int destination = (rank + 1) % size;
        for (i=0; i<m; i++) { // side processes send m messages
            MPI_Send( &signal, 1, MPI_INT, destination, 0, MPI_COMM_WORLD);
        }
    }
    else printf ("No communication with only one process.");

    MPI_Finalize();
    return 0;
}
```

Minimize average time per message for 12 processes

If you test explicit this program, you achieve smallest average times per message for all processes on one node. Due to the small amount of calculations the program's time intensive parts are made up of communication between the processes. Therefore the program is slower if it has to communicate via Ethernet. But if you think about a more calculation intensive program, which has to make a ring communication it is sensible to assign one process to one core. This means for the computers creek01-08, which have octa cores, eight processes per computer. That means in case with 12 processes you should assign the first 8 to one node and the last 4 to the next computing node. In the measurements we got the times:

	average time per message [μs]
12 processes on one node	≈ 1.5
8 processes and 4 processes on node	≈ 4.5

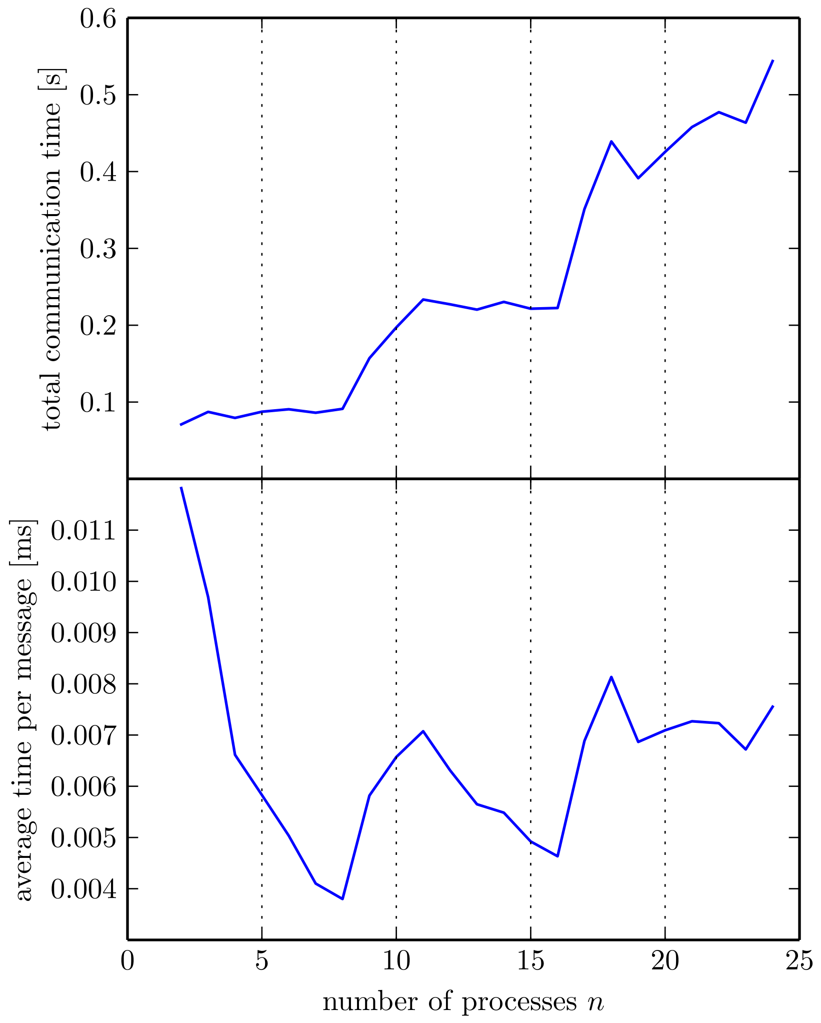

Abbildung 1: For every number of processes the total communication time was measured ten times. One point in the graph represents the average of the ten measurements. The average time per message was not measured in a direct way, rather it was calculated out of the total time. Every process sent 3000 messages to another process.

Barrier Synchronization

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

void barrier(int *rank, int *size) {
     int i;
     int signal = 1;
     MPI_Status status;

          if (*(rank) == 0) {

               // root sends message to processes [1 ... *(size) - 1]
               for(i = 1; i < *(size); i++) {
                    MPI_Send(&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                    printf("%d: sent message to Process No. : %d\n", *(rank), i);
               }

               // root receives message from processes [1 ... *(size) - 1]
               for(i = 1; i < *(size); i++) {
                    MPI_Recv(&signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &status);
                    printf("%d: received message from Process No. : %d\n", *(rank), i);
               }

          }
          else {
               // Process *(rank) receives message from root (rank 0)
               MPI_Recv(&signal, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
               printf("%d: received message from Process No. : %d\n", *(rank), status.MPI_SOURCE);

               // Process *(rank) sends back message to root (rank 0)
               MPI_Send(&signal, 1, MPI_INT, status.MPI_SOURCE, 0, MPI_COMM_WORLD);
               printf("%d: sent message to Process No. : %d\n", *(rank), status.MPI_SOURCE);
          }
}

int main(int argc, char **argv) {
     int i, size, rank;
     double t, starttime, endtime;
     int iterations = atoi(argv[1]);

     MPI_Init(&argc,&argv);        // Initialisation of MPI

     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);

     for(i = 0; i < iterations; i++) {
          starttime = MPI_Wtime();
          //MPI_Barrier(MPI_COMM_WORLD);
          barrier(&rank, &size);
          endtime = MPI_Wtime();
          t += (endtime - starttime);
```

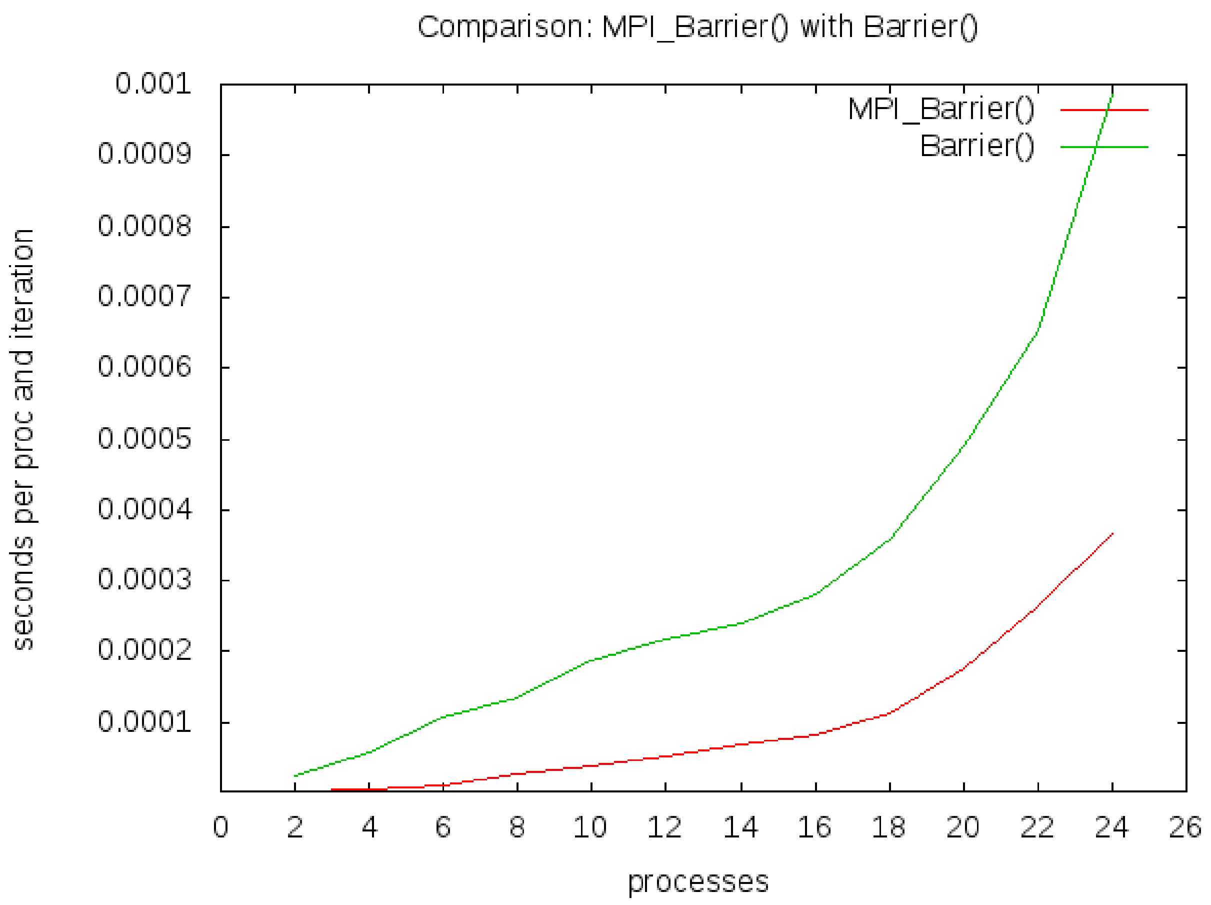

Abbildung 2: Comparison between built-in MPI_Barrier() and the implemented centralized Barrier(). The measured time is the average time needed for a process to reach the barrier for one iteration. Both barriers where tested for 2 up to 24 process in parallel and 100 iterations to achieve stable results.

```
    }

    MPI_Finalize();              // Deinitialisation of MPI
    printf("Time elapsed per iteration: %f\n", t/iterations);
    printf("Time elapsed: %f\n", t);
    return 0;
}
}
```

Comparison between built-in MPI_Barrier() and centralized barrier Barrier()

The amount of time needed for a process to reach the barrier increases with increasing number of parallel processes. As you can see in the chart the gap between MPI_Barrier() and the implemented centralized Barrier() gets bigger with rising amount of involved processes. For 24 processes MPI_Barrier() performs 300 percent better than the centralized Barrier().

Matrix multiply - sequential version

Non-optimized Matrix

As part of the second assignment in Intro HPC we have to implement a sequential, non-opitmized, version of the matrix multiply. This implementation should multiply two double precision floating point matrices. The matrices should be initialized by random values. In addition, we have to measure the execution time for the operation.

For all operations with matrices, we introduced the structure sMatrix. This structure contains one integer for rows, one integer for coloumns and a pointer to the address of the matrix itself.

```
typedef struct sMatrix
{
  int iRow;           // for rows
  int iCol;           // for columns
  double** ppaMat;    // double[][]
} sMatrix;
```

To allocate memory, we implemented the function vAllocMatrix().

```
int vAllocMatrix(sMatrix *pM, int iRow, int iCol)
{
  int i=0, iRet=0;
  /* Init rows and cols in matrix structure */
  pM->iCol=iCol;
  pM->iRow=iRow;
  /* Alloc Mem for Rows-Pointer */
  pM->ppaMat = malloc(iRow * sizeof(double *));
  if(NULL == pM->ppaMat)
    iRet=1;
  /* Allocate Memory for Rows */
  for(i=0; i < iRow; i++)
  {
    pM->ppaMat[i] = malloc(iCol * sizeof(double *));
    if(NULL == pM->ppaMat[i])
      iRet=1;
  }
  return iRet;
}
```

The function vFreeMatrix() will free the allocated memory again.

```
void vFreeMatrix(sMatrix *pM)
{
  int i;
  /* free the Rows */
  for(i=0; i<pM->iRow; i++)
    free(pM->ppaMat[i]);
  /* free cols */
  free(pM->ppaMat);
}
```

The function vInitMatrix() initializes the matrix elements with random numbers, based on the commited seed value.

```
void vInitMatrix(sMatrix *pM, int iSeed)
{
  int i,j;
  /* Initializes random number generator */
  srand((unsigned) iSeed);
  /* Fill the matrix-elements with random numbers */
  /* matrix[zeile][spalte] */
  for(i=0; i<pM->iRow; i++)
  {
    for(j=0; j<pM->iCol; j++)
      /* Generate numbers fronm 0 to 50 */
      pM->ppaMat[i][j]=(double)rand();
  }
}
```

The function vMatrixMultiply() multiplies two matrices (based on a naive algorithm).

```
void vMatrixMultiply(sMatrix *pMa, sMatrix *pMb, sMatrix *pMRes)
{
  int i,j,k;
  /*
   * In order to multiply 2 matrices, one must have the same amount of rows that the
   * other has columns.
   */
  if(pMa->iRow == pMb->iCol)
  {
    for(i=0; i<pMa->iRow; i++)
    {
      for(j=0; j<pMb->iCol; j++)
      {
        for(k=0; k<pMa->iCol; k++)
       pMRes->ppaMat[i][j] += pMa->ppaMat[i][k] * pMb->ppaMat[k][j];
      }
    }
  }
}
```

In order to measure the time we use the functions dstartMeasurement() and dstopMeasurement().

```
/* Start time-measurement */
double dstartMeasurement(void)
{
  struct timeval tim;
  gettimeofday(&tim, NULL);
  return tim.tv_sec+(tim.tv_usec/1000000.0);
}
/* Stop time-measurement */
double dstopMeasurement(double dStartTime)
{
  struct timeval tim;
  gettimeofday(&tim, NULL);
  return (tim.tv_sec+(tim.tv_usec/1000000.0)) - dStartTime;
}
```

Execution time

The implemented program should be executed on one idle node of the creek servers.

The time measurement resulted in a value of ($t \approx 146.87s$).
The GFLOP/s (Giga Floating Point Operations Per Second) get calculated by the sum of floating point operations divided by the elapsed time. In order to determine the number of floating point operations we take a look at the algorithm.

```
if(pMa->iRow == pMb->iCol)
{
  for(i=0; i<pMa->iRow; i++)
  {
    for(j=0; j<pMb->iCol; j++)
    {
      for(k=0; k<pMa->iCol; k++)
        pMRes->ppaMat[i][j] += pMa->ppaMat[i][k] * pMb->ppaMat[k][j];
    }
  }
}
```

The number of operations to be due to the single loop iterations:
$\mathcal{O}(i * j * k)$
For quadratic matrices ($i = j = k$) we get a cubic order:
$\mathcal{O}(n^3)$.
For each loop iteration we get two floating point operations. This results:
$(2 * (2048^3)) = 1.717986918 * 10^{10} \approx 17.18 GFLOP$
And for the spezific time measurement we get:
$(17.18 GFLOP/146.87s) \approx 116.97 MFLOP/s$.

Gap between achieved and theoretical GFLOP/s

The theoretical peak performance for the Intel Xeon E5-1620 CPU, used in the creek-Server, is (according to Intels datasheet: http://download.intel.com/support/processors/xeon/sb/xeon_E5-1600.pdf) 115.2 GFLOP/s. The huge gap between the theoretical and the achieved GFLOP/s is due to size of the matrices. The matrices are too big to be handled in the Cash, thus they need to be moved to the RAM. At this, the slowless to access the RAM and the slowless of the RAM itself (compared to the Cash) is the bottleneck in this system.

Optimized matrix multiplication

In order to overcome the locality problem of this program, we use a technique to improve the Cache performance. This technique is called cache blocking, or cache tiling. Here, matrices that are too big to fit in the cache are broken up into smaller pieces that will fit in the cache. Following algorithm shows our blocking matrix-multiplication.

```
int iTilledMatrixMultiply(sMatrix *pMa, sMatrix *pMb, sMatrix *pMRes, int iBlockSize)
{
int i,j,k, ii, jj, kk;
  for(ii=0; ii<pMa->iRow; ii+=iBlockSize)
  {
    for(jj=0; jj<pMb->iCol; jj+=iBlockSize)
    {
      for(kk=0; kk<pMa->iCol; kk+=iBlockSize)
      {
        for(i=ii; i<ii+iBlockSize; i++)
        {
          for(j=jj; j<jj+iBlockSize; j++)
          {
            for(k=kk; k<kk+iBlockSize; k++)
            pMRes->ppaMat[i][j] += pMa->ppaMat[i][k] * pMb->ppaMat[k][j];
          }
        }
      }
    }
  }
 return 0;
}
```

The block size (iBlockSize) is a very important part in this implementatio. The bigger the blocks are, the greater the reduction in memory traffic. But only up to a point. Three blocks must be able to fit into cache at the same time. In our case, the cache is able to store 10 MB. Each matrix element is double precision (8 Byte). So we can calculate the theoretical optimized block size by:
$\sqrt[2]{10MB/3/8B} \approx 645$.
For our algorithm we need a number based on the power of two. The next smallest is in this case 512. So, 512 is theoretical the best block size for the matrix-multiply on this system.
In reality the time measurements lead to a different result. Here we get the best result by a block size of 16. For the block size of 16 we achieve 323.23 GFLOP/s.

Block Size	Time in Sec
4	55.06
8	65.74
16	53.15
32	54.76
64	55.06
128	56.47
256	65.06
512	72.76
1024	85.46
2048	146.53