Introduction to HPC: Übung #3

Abgabe am Montag, 10. November 2014

Günther Schindler, Christoph Klein, Klaus Naumann

Contents

Moores Law

Apply Moore's Law

As part of the third exercise we should apply Moore's Law to the currently fastest supercomputer worldwide and predict the achievement of one Exaflop.
According to the list of june 2014 published on www.top500.org the currently fastest supercomputer worldwide is Tianhe-2, a supercomputer developed by China's National University of Defense Technology. It has a performance of 33.86 Pflop/s on the Linpack benchmark.
In the mid 1960s Gordon Moore made the observation that computer power, measured by the number of transistors that could be fit onto a chip, doubled once every 18 months. This law can be stated in mathematical terms as

$K_{(t)} = K_0 \cdot 2^{t/T_2}$,

where K is the computer power at the reference year t and T_2 (= 18 months) is time for duplication. After reorganizing we get

t = $\frac{T_2 \cdot ln(K_{(t)}/K_0)}{ln(2)}$.

Based on the 33.86 Pflop/s of Tianhe-2 we predict

$\Delta t = \frac{18 \cdot ln(1000/33.86)}{ln(2)} \approx 87.91 Month \approx 7.33 Years$

to exceed the Exaflop performance (2021).

Determine Growth Rate

Further, we should determine the growth rate of the TOP500 list by using the fastest system from November 2007 and November 2011. According to this growth rate we should predict the exceeding of one Exaflop by a supercomputer.
The fastest supercomputer worldwide was at November 2007 the BlueGene/L System (478.2 Tflop/s) and at November 2011 the K Computer (10 Pflop/s).
We determine the Compound Annual Growth Rate by

$CAGR(2007, 20011) = (\frac{10 \cdot 10^{15}}{428.2 \cdot 10^{12}})^{\frac{1}{2011-2007}} - 1 \approx 1.20$.

So, the calculated growth rate is 120% per year. We can now predict the exceeding of one Exaflop, based on the value of 2011, by

$\Delta t = \frac{ln(10^{18}/10^{16})}{ln(1.1+1)} \approx 6.2 Years$.

According to the growth rate of the TOP500 list (Nov. 2007 and Nov. 2011), a supercomputer will exceed one Exaflop in the year 2017.

Amdahl's Law

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{1}$$

Part 1:

$$Speedup = \frac{1}{(1-0.4) + \frac{0.4}{10}} = \frac{1}{0.6 + 0.04} = 1.5625 \tag{2}$$

According to Amdahl's law the Speedup is 1.5625.

Part 2:

Possibility 1:

$$Speedup = \frac{1}{(1-0.2) + \frac{0.2}{10}} = \frac{1}{0.8 + 0.02} = 1.2195 \tag{3}$$

Possibility 2:

$$Speedup = \frac{1}{(1-0.5) + \frac{0.5}{1.6}} = \frac{1}{0.5 + 0.3125} = 1.2308 \tag{4}$$

As one can see possibility 2 results in an higher speedup and is therefore the optimal solution.

Part 3:

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{5}$$

$$100 = \frac{1}{(1-P) + \frac{P}{128}} \tag{6}$$

$$100 = \frac{128}{128 - 127P} \tag{7}$$

$$100(128 - 127P) = 128 \tag{8}$$

$$P = 0.9978 = 99.78\% \tag{9}$$

$$S = 1 - P = 1 - 0.9978 = 0.0022 = 0.22\% \tag{10}$$

In order to achieve a speedup of 100x, the serial fraction can't be higher than 0.22%.

Measure Latency

```
#include <iostream>
#include <stdlib.h>
#include "mpi.h"

#define MS     1024*1024      // define max. message size
using namespace std;

int main(int argc, char** argv)
{
    double      *signal;
    int          rank, size, loops;
    double       starttime, endtime, t;

    MPI_Init( &argc, &argv );

    MPI_Status   status;
    MPI_Request  req;

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

    if(size < 2) {
     cout << "Minimum of 2 Processes required! Aborting programm!";
     MPI_Abort(MPI_COMM_WORLD, 1);
    }

    for (int k = 1024; k <= MS; k *= 2) {
     loops = 100;

        signal = (double *) malloc(k * sizeof(double));

            if (rank == 0) {
                /* Synchronize both processes */

          MPI_Barrier(MPI_COMM_WORLD);
                starttime = MPI_Wtime();

          for (int j = 0; j < loops; j++) {

                    MPI_Isend( signal, k, MPI_DOUBLE, 1, j, MPI_COMM_WORLD, &req );
                    MPI_Wait( &req, &status );
                    MPI_Irecv( signal, k, MPI_DOUBLE, 1, j, MPI_COMM_WORLD, &req );
                    MPI_Wait( &req, &status );

                }
                endtime = MPI_Wtime();
          t = (endtime - starttime) / loops;

            }
            else {
                /* Synchronize both processes */
```

```
        MPI_Barrier(MPI_COMM_WORLD);

        for (int j = 0; j < loops; j++) {

                MPI_Irecv( signal, k, MPI_DOUBLE, 0, j, MPI_COMM_WORLD, &req );
                MPI_Wait( &req, &status );
                MPI_Isend( signal, k, MPI_DOUBLE, 0, j, MPI_COMM_WORLD, &req );
                MPI_Wait( &req, &status );

            }
        }


    if (rank == 0) {

        cout << "Full round-trip:\t" << k << "\t" << t << endl;
     cout << "Half round-trip:\t" << k << "\t" << t/2.0 << endl;

    }
}

MPI_Finalize( );
return 0;
}
```

As shown in the figure 1 on the next page both the half round-trip and the full round-trip on two nodes requires more time than both measurements on one node with two processes. The gap between both measurements gets higher with rising message size. A reason for this is a higher intra-node bandwidth compared to two nodes connected through a network. So the sending / receiving time ascends with bigger messages.

The latency for the intra-node ping-pong test is about 14 times faster than for the inter-node version for 1024 kB. Because of the fact that latency increases for inter-node message parsing it is of primary importance to design algorithms which are efficient and can bridge this latency gap to achieve a high performance.

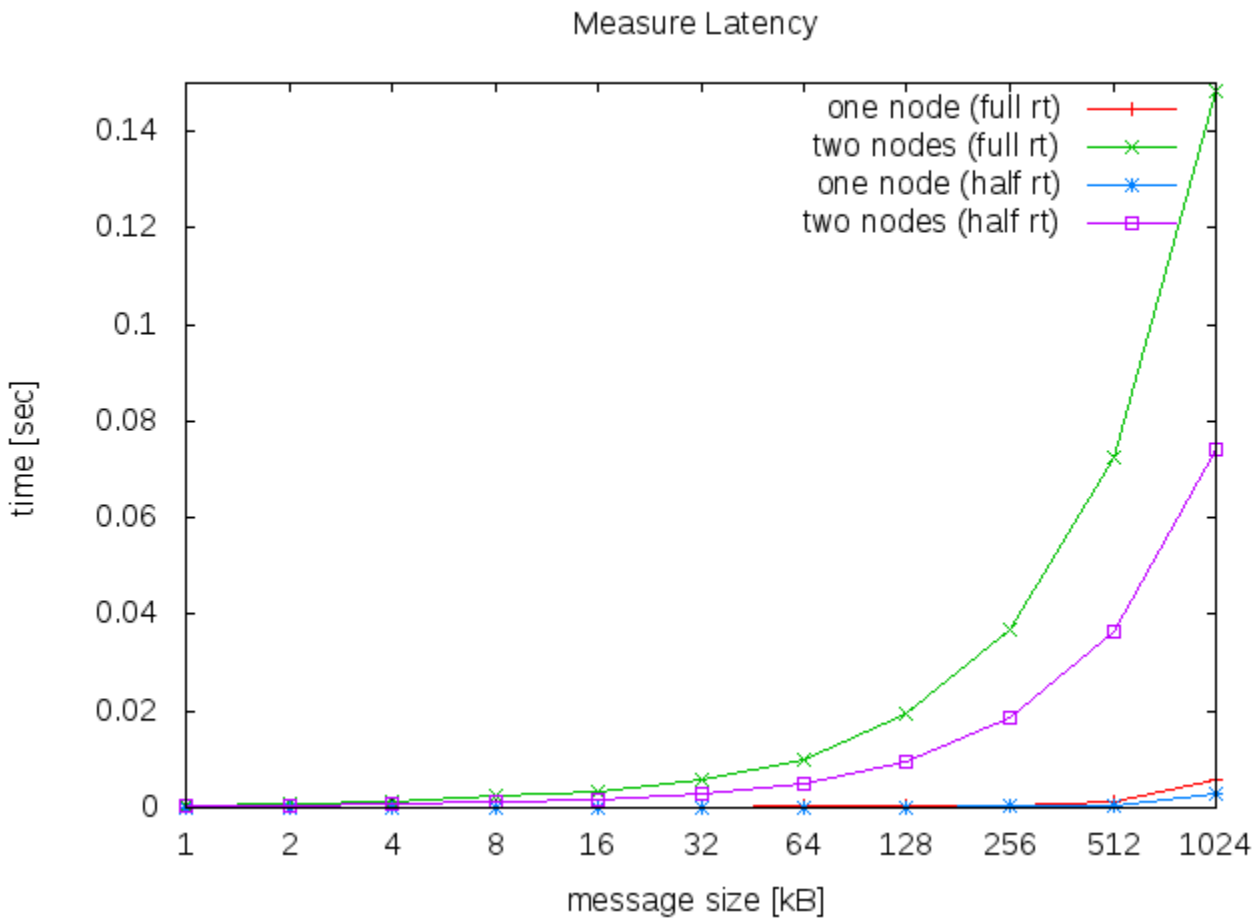

Figure 1: Full and Half round-trip latency depending on message sizes from 1 kB to 1024 kB and whether the program is executed on one (with two processes) or two nodes

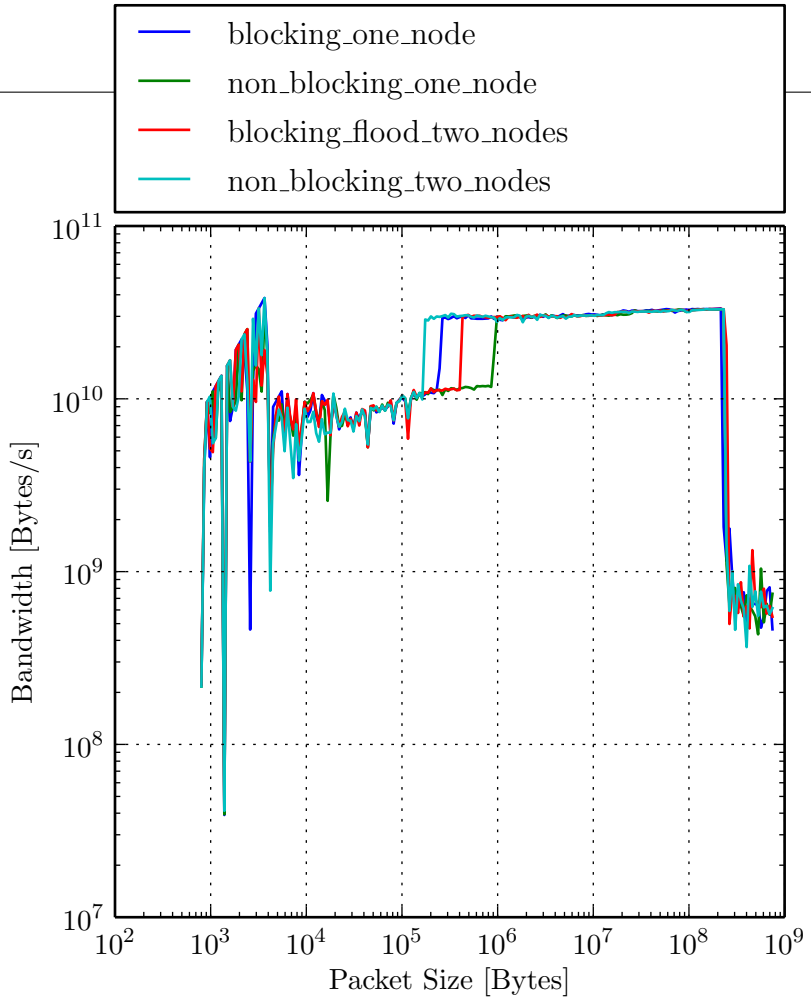

Figure 2: Bandwidth for blocking and non-blocking MPI send commands with processes on one and two computing nodes.

Measure Bandwidth

Figure 2 shows the measured bandwidths for different package sizes. You can see the bandwidth measured between two processes on one and two computing nodes with blocking and non-blocking MPI send commands. You can see that the bandwidth for non-blocking send increasis linearly, whereas the blocking send reaches one maximum performance and breaks down for high package sizes. Furthermore the increase in bandwidth of blocking send for one node at about $2 \cdot 10^5$ [Bytes] is earlier as the analog step of the blocking send for two computing nodes, due to the transfer process via network. The peaks at the non-blocking send commands might represent MPI packaging processing. At the peaks the package size might be big enough that MPI must put parts of the data into a new sending block. At low package sizes we can see startup influences at the measured bandwidths.