

CSE 557

Spring 2000

Solution for Assignment #2

Anirudh Modi

February 20, 2000

Abstract

This work attempts to determine the performance characteristics of a network of workstations. A simple program is written in C to determine the peak processing speed and the cache sizes (L1 and/or L2) of a given workstation. Another C program using the MPI libraries incorporating message send/receives is used to determine the message start-up time, incremental message cost and the communication bandwidth of a given network of workstations (NOWs). The variation of the communication bandwidth with the number of processors is studied.

1 Approach

Performance characteristics of two machines are sought in this work: COCOA and the CSE Pond 101 network of workstations. COCOA (COst effective COmputing Array) is a Penn State Department of Aerospace Engineering initiative to bring low cost parallel computing to the departmental level. COCOA is a 50 processor cluster of off the shelf PCs (Dual Pentium-II 400 Mhz with 512 MB RAM) connected via fast-ethernet (100 Mbit/sec) and runs Linux for the operating system. A single Baynetworks 24-port fast-ethernet switch with a backplane bandwidth of 2.5 Gbps is used for the networking. The machines in Pond 101 consists of 40-odd SUNSparc workstations (of varying speeds) connected via a switched 100 Mbit/sec network (fast ethernet) with two 24 hubs linked together through a single ethernet connection.

An explanation of the performance metrics that we are looking for, is explained below:

1. **Mflops:** “Mflops” is a standard measure of performance for any processor used primarily for numerical computations, and is calculated by keeping track of all the floating point operations (additions, subtractions, multiplications and divisions) that are carried on during the execution of a program. These calculations are only counted when performed on `float` or `double` type variables, and not for integers (since integer arithmetic is relatively simpler and is handled in a

different way by the processor). One Mflops stands for one million (10^6) floating point operations per second. Mflops rating is indicative of a processor's performance for numerical computations, and is usually directly proportional to its clock speed (for the same class of processors).

2. Cache:

- **L1 cache:** Level 1 (L1) Cache consists of high speed memory built into the processor. By using this cache, the processor can access frequently-requested data more quickly. The amount of Level 1 cache varies from processor to processor, and is not upgradable. L1 cache usually range from 8 KB to 128 KB.
- **L2 cache:** Level 2 (L2) Cache is separate from the processor and it is upgradeable. It is an order of magnitude slower than the L1 cache but several times faster than the main memory (RAM). The Level 2 cache works in conjunction with the microprocessor's internal cache (L1) to provide maximum performance. The total amount of supported Level 2 Cache also varies from computer. L2 cache usually range from 128 KB to 4 MB.

Other hardware being the same, in most cases, a larger cache size usually (but not necessarily) leads to faster performance. When the problem fits entirely within the cache size, the performance of the program can be several times faster than on a machine without cache. Thus, the knowledge of cache size is quite beneficial in predicting the performance of a specific numerical code.

3. **Message start-up time (t_s) or Latency:** Latency, a synonym for delay, is an expression of how much time it takes for a packet of data to get from one designated point on the network to another. Another way of looking at it is to visualize it as the time taken to establish a connection between two points on a network, before any communication takes place. When there are several small messages being transmitted/exchanged on a network at different times (which is a fairly common occurrence for a lot of applications), latency is the biggest bottleneck for the performance.
4. **Incremental message cost (t_w):** This is the time taken to transmit/receive every additional byte of information between two nodes in a network, once the communication is established between them. This is just another form of measuring communication speed, as its reciprocal gives the achievable communication bandwidth (i.e., the amount of data that can be transmitted in a fixed amount of time), which is a more commonly used performance metric.

Two small programs are written in ANSI C using the MPI (Message Passing Interface) libraries for message passing, thus making the implementation portable across all UNIX platforms. The first program barely uses MPI calls except for the timing routine (using `MPI_Wtime()`), and is used to determine the peak Mflops rating and the cache sizes of the processor. To measure the peak Mflops of the processor, this program does a simple set of floating point calculations of the form $x_i = ay_i + by_{i+1} + cy_{i+2} + dy_{i+3}$ using loop unrolling to minimize the cost due to loop overhead. To measure the cache sizes, the arrays x_i and y_i are initially allocated

a large number of elements (10^6 in our case), and only a continuous subset of their elements are accessed in increasing order to determine the Mflops. Then, the discontinuities in the array size vs Mflops graph, if any, depict the cache sizes of the processor.

The second program uses `MPI_Send()`/`MPI_Recv()` calls in MPI to communicate between processors. The size of the message being communicated is varied in a loop to determine its effect on the communication time, and the same process is repeated for several pairs of processors communicating in parallel. The graph then reveals the latency and bandwidth for the network, and the effect due the increasing number of communicating pairs.

Both the programs are attached at the end of the report for perusal.

2 Results

The output from the first program is used to generate the plots in Figures 1 and 2. The programs were compiled using the best possible optimization flags for the given architecture (i.e, `mpicc -fast` on the Pond Lab machines, and `mpicc -O6 -mpentiumpro -funroll-loops` on COCOA). They were run at the highest priority (using the `nice -19` command in UNIX), as otherwise, a lot of noise is generated in the results owing to the constant swapping in and out of the processes due to the *multitasking* nature of the operating system.

Looking at Figure 1, it is clear that the L1 cache on one of the machines in Pond 101, `melmak.cse.psu.edu`, is 32 KB, since we get a sharp discontinuity around that point. The other discontinuity, although not so sharp, leads us to believe that the L2 cache is most likely 1 MB (i.e., 1024 KB) as the performance starts to deteriorate much more sharply beyond that point [Note: The cache sizes are usually a power of 2, hence we eliminate predictions such as 900 KB or 1100 KB]. The plot also reveals the peak processing speed of `melmak` to be around 175 Mflops, which is obtained when the problem fits entirely in the L1 cache. Once the problem becomes large enough to not fit in either the L1 or L2 cache, the processing speed goes down drastically and becomes a constant 84 Mflops.

Similarly, looking at Figure 2, we find that the L1 and L2 cache sizes for the COCOA server (Intel Pentium II Xeon 450 Mhz processor) are 16 KB and 1 MB respectively, while that for the the COCOA client nodes (Intel Pentium II 400 Mhz processors) are 16 KB and 512 KB respectively. A possible explanation for the performance drop-off around 16 KB, which is well below the Intel specified size of 32 KB for both the CPUs, is because the stated L1 cache actually consists of two parts: 16 KB of data cache and 16 KB of instruction cache. For our application, it is the data cache that is relevant (and is measured), as the instruction cache is primarily utilized for the CPU instructions (as the name clearly states). The peak processing speed in this case is seen to be 255 Mflops for the Pentium II Xeon 450 Mhz processor and 225 Mflops for the Pentium II 400 Mhz processor (note: $255/450 \approx 225/400$ as $0.567 \approx 0.563$, which shows that the speeds are directly proportional to their clock speeds). Once the problem size becomes large enough to not fit in either of the cache, the processing speed drops down to a constant of about 70 Mflops in both the cases.

Relating these performance figures to the relaxation problem that was discussed in *Assignment #1*, we can say that the peak Mflops rate is only achievable for the problem if the total data size of the problem fits entirely in the L1 data cache of the processor (which is 32 KB for some of the Pond Lab machines, and 16 KB for COCOA). Assuming that we are running the second version of the relaxation algorithm, which uses only a single $k \times k$ array of type `double` (8 bytes), we can fit a problem with a grid as large as 45×45 in 16 KB of L1 data cache, and 64×64 in 32 KB of L1 data cache.

The output from the second program (run on COCOA) is used to generate the plots in Figures 3 and 4. Figure 3 plots the communication time vs message size for all the processor pairs as a scatter and fits two straight lines on the data, one each for the two different slopes clearly seen in the plot. The discontinuity is seen at around a message size of 1500 bytes, which can be easily explained by the fact that the Maximum Transfer Unit (MTU) set on the ethernet cards for each of the nodes on COCOA is also 1500 bytes. Thus, messages smaller than 1500 bytes often end up leaving holes in the packet, thus decreasing the bandwidth. From the figure, the start-up time (t_s : y-intercept of the line) for messages upto 1500 bytes is noted as 181.6 μsec , and that for messages larger than 1500 bytes is seen to be 275.2 μsec . The incremental message cost (t_w : slope of the line) for message smaller than 1500 bytes is seen to be 0.1509 $\mu\text{sec}/\text{byte}$, corresponding to a bandwidth of 53.02 Mb/s. For messages greater than or equal to 1500 bytes, the incremental message cost goes down to 0.0918 $\mu\text{sec}/\text{byte}$, corresponding to a bandwidth of 87.15 Mb/s. The run on the Pond Lab machines could not be completed due to the unavailability of sufficient resources and the lack of time, but a preliminary analysis showed that the start-up time of its network was approximately 240 μsec for small message sizes.

From figure 4, we can see that although the net communication bandwidth (i.e., sum of communication bandwidths of each communicating pair of processors) increases with the number of processors, it is not exactly linear. When the number of simultaneously communicating processor pairs becomes large, the communication time between every pair increases, as the backplane bandwidth of the switch gets saturated, and the messages can no longer be communicated at the same speed. It can be clearly seen in the figure, that the 12 processor case is slower by about 50 μsec as compared to the 2 or 6 processor case. Once the number of communicating pair becomes sufficiently large, the net communication bandwidth will be solely dictated by the backplane bandwidth of the switch (which is 2.4 Gbps in the case of COCOA), and can in no circumstances exceed that.

3 Conclusions

The performance characteristics for a network of workstations has been successfully measured in this work with the help of the two programs discussed above. The peak processing speed for `melmak.cse.psu.edu` was measured to be 170 Mflops, for the COCOA server (P-II Xeon 450) to be 255 Mflops, and for each of the COCOA client nodes (P-II 400) to be 225 Mflops. The L1 and L2 cache sizes (data cache only) for `melmak` was found to be 32 KB and 1024 KB respectively, for COCOA

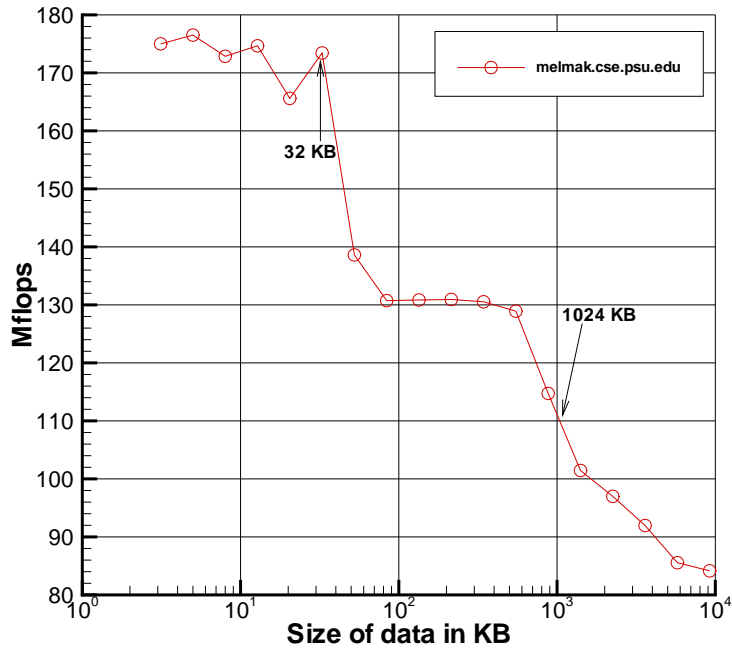


Figure 1: Cache size and Mflops on *melmak.cse.psu.edu*

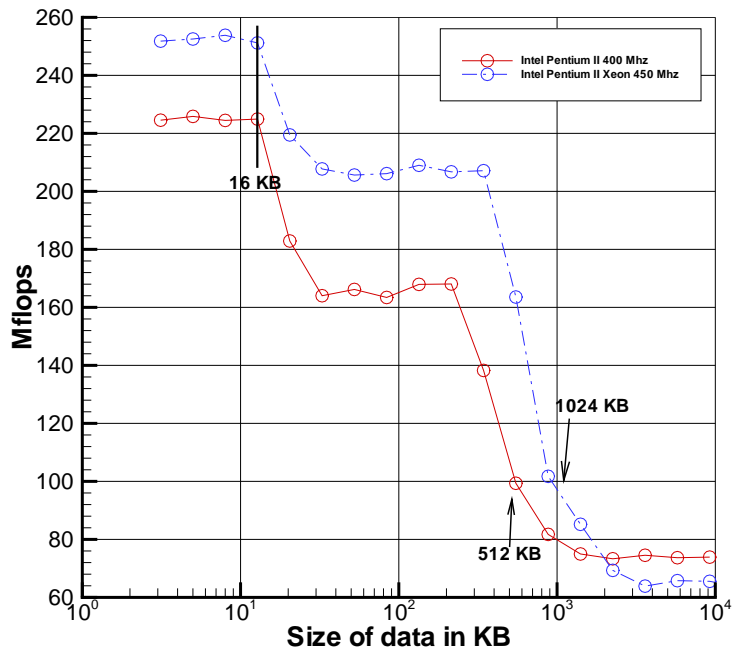


Figure 2: Cache size and Mflops on *COCOA*

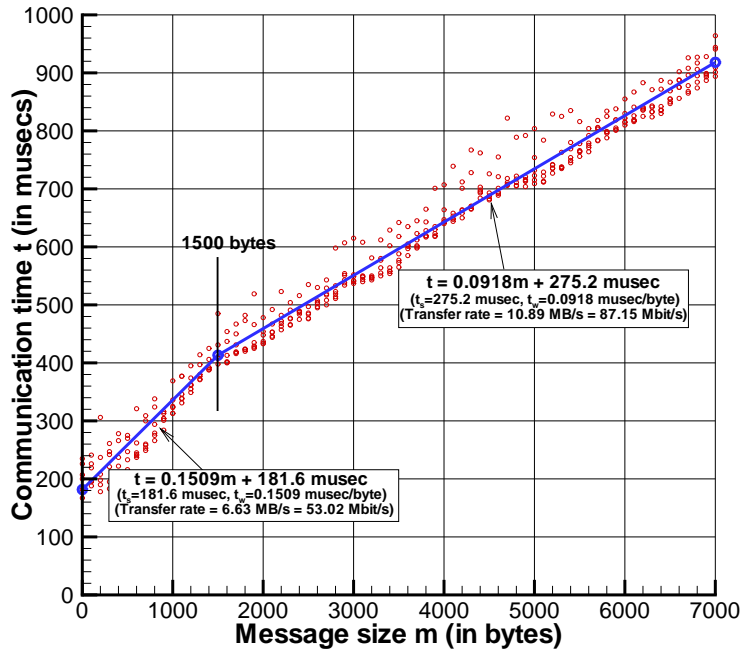


Figure 3: Communication time vs. message size on *COCO*A

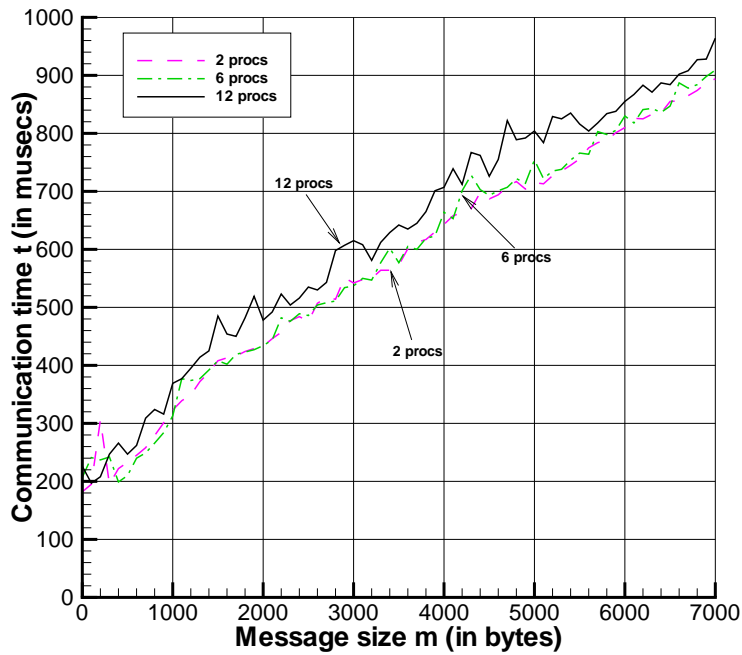


Figure 4: Communication time for different number of processors on *COCO*A

server to be 16 KB and 1024 KB respectively, and for the COCOA client nodes to be 16 KB and 512 KB respectively. It can be seen that the processing speed drops drastically once the problem size grows larger than the L1 and L2 cache (A drop $175 \rightarrow 130 \rightarrow 84$ Mflops is seen for me.lmak, $255 \rightarrow 207 \rightarrow 65$ Mflops for COCOA server, and $225 \rightarrow 166 \rightarrow 75$ for the COCOA client nodes, corresponding to the data residing in L1 cache, L2 cache, and main memory, respectively). To achieve the peak processing speed, care should be taken to make the problem fit entirely in the L1 data cache, and the best optimization flags for the available compiler should be used.

For the communication characteristics, it is seen that as the message size goes above 1500 bytes (for COCOA), the message start-up time (i.e., latency) increases (from 0.182 ms to 0.275 ms, i.e., 51% increase), but so does the communication bandwidth (from 53.02 Mb/s to 87.15 Mb/s, i.e, 64% increase). This has been explained due the MTU on the ethernet card (or raw packet size) being set as 1500 bytes. Also, it is impressive to note that a communication bandwidth as high as 87% of the peak theoretical bandwidth (of 100 Mb/s) can be obtained on COCOA. It is also seen that the net communication bandwidth tends to saturate with the increase in the number of communicating pairs. This can be attributed to the finite backplane bandwidth of the switch and the possibility of other processors (owned by other users) communicating at the same time.

```

/*****
/*   "prog1.c"
/*****
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
/*****
#define STOPCOUNT 19
/*****
int main(int argc, char *argv[])
{
int      maxsize, nsamples, unroll = 4, count;
int      i, k, isize, ip1, ip2, ip3;
double   *x, *y, a, b, c, d;
double   drand48();
double   factor, flops_per_loop, nflops, mflops, size, starttime, time;
FILE     *out;

out = fopen("prog1.out", "w");

MPI_Init(&argc, &argv);

maxsize = 1000000; // 2 arrays of this size = 16MBytes

x = (double *) malloc (maxsize * sizeof(double));
y = (double *) malloc (maxsize * sizeof(double));

printf("Maxsize = %g KB\n", maxsize * sizeof(double) / 1024.0);

isize = 200; factor = 1.6; flops_per_loop = 28.0, count = 1;

// main loop, keep increasing active set size until as large as allocated memory
while (isize < (maxsize - unroll))
{
// set nsamples large relative to tick, but not too large
nsamples = 1000;
size = nsamples * sizeof(double);
// total number of flops per timing (note done as double)
nflops = flops_per_loop * nsamples * 1.0 * isize / unroll;

// initialize and get active set in cache
for (i = 0; i < isize + unroll; i++)
{
x[i] = drand48(); y[i] = drand48();
}
a = drand48(); b = drand48(); c = drand48(); d = drand48();

// timing loop
starttime = MPI_Wtime();
for (k = 0; k < nsamples; k++)
{
// actual calculation, unrolling to minimize indexing overhead
for (i = 0; i < isize; i += unroll)
{
ip1 = i+1; ip2 = i+2; ip3 = i+3;
x[i] = a*y[i] + b*y[ip1] + c*y[ip2] + d*y[ip3];
x[ip1] = b*y[i] + c*y[ip1] + d*y[ip2] + a*y[ip3];
x[ip2] = c*y[i] + d*y[ip1] + a*y[ip2] + b*y[ip3];
x[ip3] = d*y[i] + a*y[ip1] + d*y[ip2] + c*y[ip3];
}
}
time = MPI_Wtime() - starttime;

// calculate and print

```

```
mflops = nflops * 1.0e-6 / time;
printf("size(%d) = %g KB; mflops(%d) = %e;\n", count,
      2.0*isize*8.0/1024.0, count, mflops);
fprintf(out,"%g %g\n", 2*isize*8.0/1024.0, mflops);
fflush(out);
count++;

if (count == STOPCOUNT) break;

isize = (int) (factor * ((double) isize));
}

fclose(out);
MPI_Finalize();
return 0;
}
/*****/
```

```

/*****
/*   "prog2.c"
/*****
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include "mpi.h"
/*****
// Program 2 outline
#define START_LEN 0
#define FINISH_LEN 101000
#define NUM_INCR 101
#define NUM_SAMPLES 1000
#define SLEEP_VAL 1
#define MASTER (myid == 0)
/*****
int main(int argc, char **argv)
{
char    *buffer;
int     count1, count2;
int     dest, order;
long    s_val = SLEEP_VAL;
int     np, myid, p, i, j, start_len, finish_len, inc;
double  time1, time2, work, t;
FILE    *out = NULL;
MPI_Status mpistat;

start_len = START_LEN;
finish_len = FINISH_LEN;
inc = (finish_len-start_len)/NUM_INCR;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

printf("myid = %d, np = %d\n",myid,np);
fflush(stdout);

if (MASTER)
    out = fopen("prog2.out", "w");

if ((myid % 2) == 0)
    {
        dest = myid + 1;
        order = 0;
    }
else
    {
        dest = myid - 1;
        order = 1;
    }

// character array for sending back and forth...
buffer = (char *) malloc(sizeof(char)*finish_len);

count1 = 1;
// loop over pairs of processors...
// increase the number of pairs by 2 each time through...
for (p = 2; p <= np; p += 2)
    {
        count2 = 1;
        i = start_len;
        while (i < finish_len)
            {

```

```

// sync everyone up...
MPI_Barrier(MPI_COMM_WORLD);

time1 = MPI_Wtime();
if (myid < p)
{
    for (j = 0; j < NUM_SAMPLES; j++)
    {
        if (order == 0)
        {
            MPI_Send(buffer,i,MPI_BYTE,dest,11,MPI_COMM_WORLD);
            MPI_Recv(buffer,i,MPI_BYTE,dest,11,MPI_COMM_WORLD,&mpistat);
        }
        else
        {
            MPI_Recv(buffer,i,MPI_BYTE,dest,11,MPI_COMM_WORLD,&mpistat);
            MPI_Send(buffer,i,MPI_BYTE,dest,11,MPI_COMM_WORLD);
        }
    }
}
time2 = MPI_Wtime();

// wait a small amount of time to avoid interrupting
// other processes
sleep(s_val);

// get the avg of the times across the participating procs
t = time2-time1;
work = t;
MPI_Allreduce(&work,&t,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
t /= p;
if (MASTER)
{
    int microsec = floor(t/(2*NUM_SAMPLES)*1.0e+6);
    printf("size = %d bytes, t = %d microsec, %d processors\n",
        i, microsec, p);
    fprintf(out,"%d %d %d %d %d\n",i, microsec, p,
        count1, count2);
    fflush(stdout);
    fflush(out);
}
count2++;
// some rule for incrementing message size with inc
i += inc;
}
count1++;
if (MASTER)
{
    fprintf(out,"\n");
    fflush(out);
}
}

free(buffer);

if (MASTER)
    fclose(out);

MPI_Finalize();

return 0;
}
/*****/

```