Anomalies

The following issues might make the performance of a parallel program look different than it is:

When running a program in parallel on many processors, each processor has its own cache, so the total cache size increases. This can give smaller-size problems a speedup that is beyond parallelization.

For small problems, communication cost can be disproportionally large, in particular if only little communication has to take place.

Suppose we are searching in some domain. Once the desired element has been found, the search can terminate.

Now suppose this search is in a tree whose branches have different length and we parallelize this search over these branches.
Then those tasks, that work on shorter branches finish earlier, the \textit{average} run time of the parallelized algorithm thus is shorter than sequential runtime divided by $n$.

(A similar effect can happen if each test involves a backtrack search, where a “yes” answer often is obtained much faster than a “no” answer.)

The reason for this extreme speedup is that the algorithm actually changed when parallelizing – a “simultaneous” search would also produce a better sequential algorithm.
MPI

MPI (http://www.mcs.anl.gov/mpi) is a standardized package for parallel communication:

- “Cooperative” communication: Messages must be sent and received
- Portable
- Large hardware base
- Can use special hardware
- “Standard” Unix version MPICH
Setting up parallel processes

MPI consists of two layers: The top layer gives the actual user interface, the bottom layer provides the actual communication.

This bottom layer can be machine dependent (for example on a specialized multiprocessor machine).

However MPICH provides for a mechanism (called “p4” device) which uses standard TCP/IP (the internet protocol) methods for communication.

This way one can simply use a set of workstations, connected by a network.
To set up a parallel calculation MPI has to know what programs to run on which machines. For this we set up a file `mymachines` that contains the names of the machines to use for the calculation (one per line):

```
stokes
stokes
stokes
schur
...
```

We then start the program with `mpirun`. This will actually start programs on all machines and take care of communication (Note: ‘mpirun’ must be in your path):
mpirun -machinefile mymachines -np 5 a.out

The -np option indicates how many processes to spawn. They will run on the machines as given in the order in the mymachines file. This means that the same program is running on all machines. The processes have to find out themselves what “number” they have.

If you want to use Master/Slave processing the processes have to decide based on this number whether they are master or slave.

(There are various other possibilities as well, for example if the program paths differ on different machines, or if the machines are even of a different architecture. See the documentation for details.)
Remote Access

MPICH uses remote login for communication:

- rsh (old, insecure)
- ssh (secure)

Many current systems block rsh.

To avoid retyping the password for every new process, you can enable ssh to trust the machine you come from, if they share the same file system:

Call ssh-keygen \(-t\) dsa. Then go in the .ssh directory and copy id_dsa.pub to authorized_keys.

After this you should be able to log in automatically from ssh. (Test with ssh \(-n\) ‘hostname’ date for example).
Ground Rules

Include `mpi.h` to have prototypes for the MPI functions.

(This will mean that the corresponding include file and library file have to be available to the compiler. In the worst case, we have to add them explicitly:

```bash
cc myprog.c -I /usr/local/mpich/include/ -L /usr/local/mpich/lib/ -lmpich
```

Call `MPI_Init(&argc, &argv);` to start MPI.

This function should get the program arguments (`argc`, `argv`) passed.

At the end, call `MPI_Finalize();` to end MPI.
All non-MPI routines are local. So for example `printf` will produce output on the (different) machines.

MPICH will transmit screen output of the slaves to the master machine and display it there.

There are two functions to get the total number of processes as well as a processes index number:

`MPI_Comm_size(MPI_COMM_WORLD,&numprocs);` assigns to the int variable `numprocs` the total number of processes.

`MPI_Comm_rank(MPI_COMM_WORLD,&idnum);` assigns to the int variable `idnum` the local index number (starting at 0).
A basic MPI program

#include <stdio.h>
#include "mpi.h"

int main(argc, argv)
int argc; char *argv[];
{ int myid, numprocs;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  printf("Hello from %d\n",myid);
  printf("Numprocs is %d\n",numprocs);
  MPI_Finalize();
}
**Communication**

MPI has two functions to send/receive data from one process to another (the processes may block until the communication is established.)

```c
MPI_Send(buf, count, type, destination, tag, comm);
```

sends a message to process number `destination` (an `int`).

The message comes with an integer `tag` number (which identifies the message).

The *communicator* is typically set to `MPI_COMM_WORLD`.

`buf` is a pointer to a memory buffer that contains the data, the data consists of `count` objects of data type `type`.

The data types (for example `MPI_INT`, `MPI_DOUBLE_PRECISION`) help to translate data between different architectures.
To receive data, we use:

MPI_Recv(buf,count,type,source,tag,comm,status);

Here source indicates the process to receive from. MPI_ANY_SOURCE indicates that any message source is accepted. tag is the message tag, MPI_ANY_TAG accepts any.

status is a pointer to a status variable of type MPI_Status that can indicate errors.
Example

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

float integral(float a, int i, float h, int n)
    [...]

void main(argc,argv)
int argc; char *argv[];
{
    int n, p, i, j, ierr, myid, source, tag;
    float h, result, a, b, pi, my_result;
    MPI_Status status;
    pi = 3.1415926; a = 0.; b = pi*1./2.; n = 500;
    tag = 123;
}```
MPI_Init(&argc, &argv);  
MPI_Comm_rank(MPI_COMM_WORLD, &myid);  
MPI_Comm_size(MPI_COMM_WORLD, &p);  
h = (b-a)/n/p;  
i = myid;  
my_result = integral(a, i, h, n);  
printf("Process %d has the partial result of %f\n", myid, my_result);  
if(myid == 0) {  
    result = my_result;  
    for (i=1; i<p; i++) {  
        MPI_Recv(&my_result, 1, MPI_REAL, i, tag, MPI_COMM_WORLD, &status);  
        result += my_result;  
    }  
    printf("The result is %f\n", result);  
} else MPI_Send(&my_result, 1, MPI_REAL, 0, tag,
MPI_COMM_WORLD);
    MPI_Finalize();
}

Running the code we get:

/usr/local/mpich/bin/mpirun -machinefile mymachines a.out

Process 0 has the partial result of 0.309017
The result is 1.000000
Process 2 has the partial result of 0.221232
Process 4 has the partial result of 0.048943
Process 1 has the partial result of 0.278768
Process 3 has the partial result of 0.142040