Background on parallel computing

This web page contains information and pointers to information on parallelization in general.

Paradigms in parallel computing

This section concentrates on the general parallelization methods commonly used. It is important to analyse the problem at hand and decide what kind of parallelization is to be used, regardless of the parallelization soft- and hardware available. Two important classes have emerged in the history of parallel programming: data parallel and functional parallel.

Data parallel methods

The data parallel methods are essentially based on distribution of the data among several processors. Usually the processors execute the same kind of code on different pieces of data. Examples can be found in various techniques for matrix multiplication, matrix inversion and so on. Also in the case that memory at one processor is not enough to hold all data, data parallel programming can offer a solution.

A typical data-parallel code has the following structure:

1. distribute data among processors
2. processors communicate with other processors to exchange border data
3. processors perform calculations on their own and neighbours' data
4. exit if convergence has been achieved
5. if necessary, redistribute data among processors to obtain better load balancing
6. goto 2

Depending on the problem at hand, the code required can be difficult to write and maintain. Fortunately, there are tools available: ScaLAPACK is a subroutine library that contains data-parallelized versions of often used matrix operations.

Functional parallel methods

Functional parallel methods are based on the distribution of the execution of more or less independent tasks on several processors. Many problems can be transformed this way. Good examples are Monte-Carlo simulation where the same calculations are performed again and again, each time with different (randomly) chosen parameters. The child processes don't have to be identical: each child process can perform different calculations, perhaps on the basis of the input data at hand. In general, the farming paradigm is used: one parent process is responsible for allocating work to child processes, collect and process the answers.

A typical functional parallel program has the following structure:

After creation of parent and child processes:

Parent code

1. prepare list of tasks
2. wait for message from child process
3. if message = "ready"
   a. send task data to child
   b. goto 2
4. process answer from child
5. adjust list of tasks
6. exit if all tasks are done
7. goto 2

Child code

1. send "ready" to parent
2. wait for task data from parent
3. calculate answer from task data
4. send answer to parent
5. goto 1

In general, functional parallel programs are relatively easy to write and maintain. Furthermore, it is not too difficult to add or delete processors from a running farm, as long as the parent process is kept alive. Load balancing is done automatically, if the time spend in sending messages is small in comparison with the CPU time the child processes. The parent process should use as little CPU time for processing the answers of the child processes as possible.

In above example it is assumed that the child processes can perform their calculations independently. If this is not the case, the program becomes more complicated and probably some kind of synchronization is needed.

Taxonomy of parallel methods

A widely accepted classification of computer architectures is "Flynn's classification", dating back to the mid-sixties. The classification is based on the number of different program streams and data streams.

1. SISD: Single Instruction stream, Single Data stream computers: in fact an ordinary one-processor computer 2. MISD: Multiple Instruction stream, Single Data stream computers: these computers have never been built, all you can do on a MISD computer you can do on a more versatile MIMD computer 3. SIMD: Single Instruction stream, Multiple Data stream computers: typical examples are the traditional vector computers, when executing vector code: the same operations are performed on different data. 4. MIMD: Multiple Instruction stream, Multiple Data stream computers: these are now the most common parallel systems. Each processor can execute it's own program using it's own data.

Next to Flynn's classification, another subdivision is appropriate:

1. SM: Shared-Memory systems: the processors in a shared memory system can address the same, shared memory very fast 2. DM: Distributed Memory systems: the processors rely on message passing for exchange of data.

So, in principle you can have SM-MIMD, DM SIMD etc. systems. Things are even more complicated by the fact, that often SM machines are, on close inspection, in reality DM machines, with a very fast message passing network, firmware and software to present the system as a SM system for the programmer. Also some software packages (HPF, BSP) present a message passing system to the programmer as a SM system.

Literature: Overview of recent supercomputers, Issue 2010, Aad J. Van der Steen, NCF

Methods available methods for parallel programming

Here we will discuss the various software tools available at SURF to produce parallel programs: MPI, PVM3, OpenMP and name some higher-level numerical libraries.

MPI

MPI is an international accepted standard for a message passing library, for use in C en Fortran programs. (MPI information) MPI is available for most computer systems and operating systems. MPI-1 is the old standard. A new, upward compatible standard, MPI-2, has been developed. There exist several implementations of MPI including:

- MPICH Developed at the University of Chicago and the Mississippi State University
- LAM-MPI Owned by the Indiana University
- open-mpi An attempt to combine the strengths of several MPI implementation

One of the strengths of MPI is, that it is a well defined standard to which all implementations should comply. The fact that there are several brands invoked a sound competition which has led to very efficient implementations. MPI is now the most used tool for parallelization of high performance applications for distributed and shared memory systems. MPI is suitable for implementing data-parallel and function-parallel paradigms.

PVM3

PVM3 (Parallel Virtual Machine, version 3) is a message passing library, to be used in Fortran and C programs. PVM3 is available for most operating systems, including Unix and Windows. PVM3 is suitable for implementing data parallel and function parallel paradigms. There is not a standard for PVM3: the standard is effectively defined by the implementation by the Oak Ridge National Laboratory.

OpenMP

OpenMP is a standard, supporting shared memory multiprocessing programming in C/C++ and Fortran on most operating systems, including Unix and Windows. OpenMP is defined as an extension to the C/C++ and Fortran languages, so one needs special compilers to use it. Vendors of shared memory systems often offer OpenMP-aware compilers with auto-parallelizing capabilities.

OpenMP is most suitable for implementing data parallel paradigms.

Combination

Often it makes sense to combine OpenMP with MPI, especially if the computer consists of a number of nodes, each equipped with 2 or more processors or processor-cores. MPI is used for distributing the work between the nodes, while OpenMP is used to distribute the work on a node between the processors.

ScaLAPACK

ScaLAPACK (SCAlable Linear Algebra PACKage) is a library of parallelized Linear Algebra routines. Information

FFTW2/3

FFTW2 and 3 (the Fastest Fourier Transform in the West, version 2 and 3) are libraries of parallelized multi-dimensional Fast Fourier routines. Both are parallelized using multi threading for shared memory machines. FFTW2 is also parallelized using MPI. Information

Debugging

Parallel programs are notoriously hard to debug. Careful insertion of print statements is often the easiest way to pinpoint an error. Also it is worthwhile to keep the parallel code as clean and simple as possible.

For debugging parallel programs the vendors of the systems often have available parallel debugging tools, specific for the system at hand. These can sometimes be used to pinpoint the reason for deadlocks, livelocks and other kinds of misbehavior of programs.

Optimization

Two important things are to be considered when optimizing a parallel program:

1. optimize the code without caring about the parallelization
2. optimize the parallelization

In more detail this means:

1. Optimizing the "serial" code includes:
    a. identify the "hot spots" in your program, and concentrate on these
    b. choose the right algorithm (a simple example would be: change a bubblesort algorithm for a quicksort algorithm)
    c. clean up the code: in general, compilers are better able to optimize cleanly laid-out loop constructs than "spaghetti code"
    d. rewrite program loops, so the compiler is able to optimize them
    e. inline frequently used small subroutines
    f. use the appropriate compiler options
    g. optimize I/O
2. Optimizing the parallelization code
    a. get an impression of how much time is spent in message passing vs CPU usage. If time spent in message passing is relatively high (e.g. 50% or more) compared with CPU usage, than you should probably try to optimize the parallelism. On the other hand, if message passing time is low (e.g. 10% of the CPU time), than it is probably not worthwhile to try optimizing the parallelism.
    b. identify hot spots in message passing, and concentrate on those
    c. try to reduce message passing overhead:
        i. combine several messages into one
        ii. use another algorithm
        iii. investigate the possibility of using a standard library
        iv. use another message passing library (e.g.: switch from PVM to MPI)
        v. optimize number of processors
3. optimize I/O

CPU time versus wall clock time

In optimizing serial code, one can use the amount of CPU time used as a guideline: the less, the better. In optimizing parallel code, however, the total amount of CPU time used is almost meaningless: the higher the degree of parallelization, the more CPU time a program will use. Instead of CPU time, minimization of wall clock time is the goal.