SIP Design

Overview

The parallel modules of ACES II are programs written in Fortran 77 and C++ using POSIX Threads and MPI that provide a flexible environment to execute complex algorithms involving very large data structures and multiple high-order tensor contractions. The problem that spurred their development and design is the evaluation of Coupled Cluster singles and doubles (CCSD) energies and gradients.

The design philosophy draws on the similarity with the architecture of an early generation processor, for example the VAX 11/780. One thinks of the processor executing a stream of machine instructions written in assembly language. Each instruction takes a considerable amount of time, from a few processor cycles for an IF test, to fifty or more cycles for a floating point multiply. This is in contrast to modern micro processors that execute all instructions in one cycle and often execute multiple instructions each cycle. Writing efficient programs then requires a careful scheduling of instructions such that the amount of time that the processor must wait is minimized. This can be achieved by scheduling reading from memory to start way before the data are needed for e.g. a floating point operation, and by starting other operations while the writing to memory of the results of an earlier operation is still in progress.

The CCSD data structures are huge. For the design, the data structures are not considered as arrays of floating point numbers, but as arrays of blocks, each containing typically 10,000 floating point numbers. In the design, the algorithm is written in terms of block operations, not operations on individual floating point numbers. As a result, even on modern multi-GHz processors, every block operation takes a considerable amount of time. Because each block is large, e.g. a contraction of two blocks, each a 4-index object, over 2 indices to produce another 4-index object takes considerable time. Furthermore, this time can be tuned by changing the basic block size. It then becomes feasible to expect that every read from and write to some other task in the parallel program using MPI communication can be tuned to be less than the time for a contraction. With careful scheduling and pre-fetching of data, a large portion of all communication can be hidden behind computation, resulting in favorable parallel performance of the program.

Because of this similarity, we call the primitive operations executed by the new ACES II modules *super instructions* and we call the parallel program itself the *Super Instruction Processor* or SIP.

The algorithm to be performed is expressed in a special purpose language, called *Super Instruction Assembly Language*, or SIAL, pronounced “sail”.

SIP Components

Logically, SIP has the following components:

1. A component coordinating the work to be done by all tasks; this component executes in the master task during initialization;
2. A component for communication of basic data elements, blocks, between the cooperating tasks, the most visible aspect of this component is the **distributed array**;
3. A component for storing and retrieving large amounts of data, providing support for the **served arrays**;
4. A component for executing basic chunks of work in the form of super instructions; this component calls the communication and data storage components when necessary.

SIP is a parallel MPI program that consists of multiple tasks. Some tasks are dedicated to special functions, others are more general. To provide the necessary flexibility to tune the performance of the SIP, each task runs multiple POSIX threads for communication in addition to the main thread, which performs the coordination of all threads and executes super instructions.

The IOCOMPANY

Tasks are grouped into *companies*, which perform a given function cooperatively. For example, one company, called the *iocompany*, is dedicated to providing support for **served arrays**. Each task has several threads, ready to receive messages from tasks in other companies performing work on the algorithm. All data belonging to a **served array** is divided into blocks and blocks are always received into or sent from the memory of one of the tasks in the *iocompany*. The master task sets up tables designating which task will hold which block of every array using a simple algorithm, so that no searching for data blocks is necessary.

At a low priority, every task in the *iocompany* investigates the status of all data blocks in memory, and when the pool fills up above some threshold, the low priority thread starts to copy blocks to locally accessible disk storage. The algorithm is similar to that of managing paging space in a modern operating system. Blocks that are often read or changed regularly will remain in memory for quick access, whereas blocks that are used infrequently migrate to disk. If a request for a block is made that is not resident in memory, a delay will occur before the request completes, which is caused by the need to restore the block from disk.

The *iocompany* also has the capability to compute blocks of integrals. If a request arrives for a block and it is not found to be resident in memory or disk, the task assumes an integral block is needed in direct mode, and it starts an integral computation, which will be transmitted when the computation completes.

Other companies and platoons

Other companies execute SIAL programs. All tasks in a company execute the same SIAL program. Very complex algorithms may require the cooperation of multiple companies, each executing a different SIAL program. The tasks in different companies can communicate with each other through the *iocompany* by reading and writing **served arrays**, for example. They can also communicate directly.

A company can be divided further into *platoons*. All tasks in one platoon hold one copy of one or more **distributed arrays**. This allows optimization of data access as follows. As all tasks in a company are processing data for some algorithm, they will need to read from and write to one or more **distributed arrays**. Because the **distributed arrays** are replicated in each *platoon*, communication between all tasks in the *company* can be performed without any single task becoming a bottleneck. This will only work, of course, if the size of the data and the number of tasks is such that enough local memory is available in each task to hold the multiple copies of the same data.

Communication of **distributed array** data is performed as follows: Each task has one or more threads running that are listening for requests. When a request is made the necessary locks are acquired to ensure integrity of the data, and then the block is asynchronously sent or received. While the communication is taking place, more requests for other blocks can be processed. Multiple requests to read the same block are also processed at the same time, thus reducing wait time for the client of the **distributed array**.

Super Instruction processing

The activity of each task in all companies except the *iocompany* is controlled by an SIAL program. It is a list of *super instructions* to be executed. The *super instructions* can initiate communication, send or receive, or computation such as the tensor contraction of a two blocks of data into a third block. The computation can take a significant amount of time, depending on the block size. It is also possible that the instruction starts the computation of a block of integrals.

As much as possible, the *super instructions* are executed asynchronously: Communication operations are started and then control returns so that computation can be performed. When the data has is really needed, the task checks whether the communication instruction has completed successfully. The task can also look ahead in the instruction list and start certain operations early. The purpose of this flexibility is to try and maximize the hiding of communication delays behind computation work, thus minimizing over all waiting times in the execution of the parallel program.

APPENDIX: IOCOMPANY components

```
      served arrays client
            | ^
            | |
        prepare request
            | |
            | |
            v |   server process
            comm  block pool
            thrds
             <--->      block struct <--->
                  empty flag  I/O
                  copied flag thrds
                  write lock  | ^
                  data        | |
                              | |
                            write read (could be asynchronous)
                              | |
                              v |
                        Unix buffer cache
                              | ^
                              v |
                              disks
```

- the communication threads are the server front end and run at high priority, they are only blocked when all blocks in the pool are full and no blocks have the copy-to-disk-completed flag set to true.
- The I/O threads are the back end and run at low priority, they copy blocks to disk, which means make system requests to copy blocks to the buffer cache and the kernel will move them to disk when it needed.
- the block write lock is set whenever the front end updates a block with a prepare or the back end restores a previously deleted block from disk, and during that time all requests for the block are put in wait.