The super instruction processor parallel design pattern for data and floating point intensive algorithms

V. Lotrich, M. Ponton, L. Wang, A. Yau, N. Flocke, A. Perera, E. Deumens, R. Bartlett
AcesQC, Gainesville, Florida

ABSTRACT

A design pattern is considered in which a distributed memory, multi processor computer is viewed as an early generation processor. In such processors, each instruction operates on individual floating point numbers and consumes a variable yet significant number of cycles. In the design pattern, we consider the elementary data item for a modern parallel computer to be a block of many numbers. Each operation involves a significant amount of CPU work, since the operands and result are blocks. We also consider memory access operations to include the delays incurred sending blocks to remote nodes. By making all delays explicit and organizing the algorithm such as to provide sufficient work to make each operation take a measurable amount of time, a new paradigm for designing and optimizing data and floating point intensive algorithms emerges. The application of the design pattern to the construction of a parallel implementation of the Coupled Cluster Singles and Doubles energy and gradient calculation is discussed. The efficiency of developers and the performance of the created software are discussed.

1. The super instruction design pattern

Designing, writing and debugging programs that implement numerically intensive algorithms are demanding and labor intensive tasks. Trying to implement such algorithms on modern computer systems consisting of many CPUs connected by low-latency, high-bandwidth switching networks is notoriously difficult. The problem that we are considering comes from the domain of Computational Chemistry. In computational chemistry, one of the problems is to determine the wave function of a many electron system where all the electrons are moving in the Coulomb forces from the nuclei. The Hartree-Fock method gives a single particle approximation, essentially the product of orbitals. To correct the wave function to describe the fact that the electrons repel each other, one needs to do a correlation calculation. The ultimate brute full method is CI (configuration interaction), but that is too cumbersome and not feasible for any but the smallest systems. The coupled cluster singles and doubles (CCSD) method[1] gives a good result for a reasonable, but still large, effort. CCSD people write diagrams that look like Feynman diagrams from field theory. Each diagram is one term. There are many terms, resulting in very complex programs with large amounts of data and large numbers of operations to be performed.The first implementations of the energy[2] and gradient[3] calculations using this method on serial computers took several man-years of effort. Several attempts have been made to write parallel implementations[4,5] with varying success: Certain assumptions were made during the design and these turn out to limit the range of applications that can be treated effectively with the implementation. Retuning the implementation to support the new application is very labor intensive.

In our effort, we set out to build a more complex software system that would allow us, after completion of the code, to tune the performance to varying hardware platforms to a much greater extent than usual. To accomplish this design goal, we found it productive to set up a design pattern[6] that exhibits the flexibility needed for obtaining reasonable performance on a wide range of problems executing on a variety of hardware configurations.

1.1 Pattern inspiration

The design pattern draws on the similarity with the architecture of an early generation processor, for example the VAX 11/780. The processor is executing a stream of machine instructions. Each instruction corresponds to a small program written in microcode and takes a considerable amount of time, ranging from a few processor cycles for a simple logical test, to tens of cycles for a floating point multiplication. Writing efficient programs for the early generation processor then requires a careful scheduling of instructions such that the amount of time that the processor must wait is minimized. Although each instruction takes finite time and the time taken by CPU operations and memory operations are of similar magnitude, there is opportunity to keep the CPU busy while memory access is in progress. This can be achieved for example by scheduling read operations from memory to start a sufficient number of cycles before the data is needed for a CPU operation, and by starting new CPU operations while the write operation to memory of the results of the previous CPU operation is still in progress.

This is in contrast to modern micro processors that execute all instructions in one cycle and often execute multiple instructions in parallel units in each cycle. In this case the CPU operations are all significantly quicker than any memory operations, and writing efficient programs requires focusing on keeping the CPU busy. This might be fine for serial algorithms, but does not extend naturally to parallel algorithms. Even the fastest switches, such as the one used in the SGI Altix NUMA architecture, are still too slow.

1.2 Description of pattern

The design pattern must then consider the entire parallel computer system as if it were a single VAX 11/780 system with multiple CPUs. Each CPU performs work as part of the parallel program, which in our case is an MPI MIMD program and all tasks have data which is stored in the system RAM. In the pattern, all operations must be considered as time consuming. No operation is considered free or so fast that its execution time is negligible. In this way the VAX RAM can be a representation of the real systems memory which may be shared memory or distributed memory. The access time to RAM takes into account message latencies and bandwidth constraints.

To make the pattern work, the problem must allow for the organization of the data into a structure of “atomic data items” such that the algorithm can be written in terms of a set of “atomic operations” on the “atomic data items”. Then the application can be viewed as an assembler program for the VAX 11/780 and tuning the performance of the application becomes tuning the microcode of the VAX 11/780 instructions. Instead of trying to make the switch as fast as the CPU, we feed every CPU in the system “atomic data items” that are so large that every “atomic operation” takes a time comparable to the delays caused by transmitting the data from remote memory. In other words, we make the CPU operations as time consuming as the (distributed) memory access operations.

Because of this similarity, we call the primitive operations executed by the parallel computer *super instructions* and we call the parallel program implementing the instructions the *super instruction processor* or SIP. The parallel program is comparable to the *microcode*. The algorithm to be performed is expressed in a special purpose language, called *super instruction assembly language*, or SIAL (pronounced “sail”).

To apply the pattern in a design requires building a compiler for SIAL. In a disciplined software engineering project, it is possible to implement the SIAL functionality in a class library. But then there is no hard boundary between SIAL and microcode, which may jeopardize tuning efforts. In our implementation, we opted for a real compiler. It generates binary code, machine instructions, for the super instruction processor, which reads the instructions and executes them. The binary code consists of tables with numbers. The main table is the instruction table with instruction codes and operand addresses. The operand addresses are entries in data descriptor tables, which could be array tables or index tables. Some operations are like multiplications, which are compute intensive. Other instructions perform DO loop control, some for serial loops and others distributed loops.

For example, in our application, the atomic data item is a sub-block of a 4-dimensional matrix. The atomic operations are tensor contractions of the blocks. There are several such instructions to with different possibilities of what index of the first operand to combine with what index of the second operand and what index to map into which result index. If the instruction operates on a block, the SIP determines whether the block is local; if it is not a request is made from the owner task of the block. The SIP may look ahead and request several blocks that it expects will be needed in the next iteration. It can start a multiplication operation for which all operands are available. The programmer of SIAL does not know where the block resides, local or on a remote node of the real system, in the pattern the blocks just reside in RAM, which may be fast or slow to access.

The super instruction processor itself is a parallel MPI program and it is MIMD. Each MPI task executes the same super instructions program, but may not be at the same place in the program. There is no instruction level synchronization between the executing CPUs.

The SIP has no registers, but a stack. All RAM is divided into blocks that are big enough to hold any block that may ever come up. (This leads to an issue with waste if there is large variation in block size.) The SIP uses these blocks as a stack. It is important to make sure the block size has the correct relation to the cache of the microprocessor used to get optimal performance.

1.3 Applying the pattern

The use of the super instruction design pattern is somewhat different from object oriented programming with blocks, because the pattern does not only address the design of the parallel program, but also how to run and tune it. There are two levels of programming on a VAX: Application programmers focus on the algorithm and write e.g. in C or Fortran. System engineers write microcode to speed up floating point operations and read and write to RAM and disk.

The pattern we advocate to help build parallel HPC applications also defines two levels: one for the algorithm and one for the execution. The programmers writing in SIAL only have the capability to operate on blocks. The size of the blocks is not defined or accessible in that language. It is determined at run time only. The programmers writing SIP use C or C++ or Fortran and they try to write these operations as efficiently as possible. Users can execute the application on a single CPU, or on multiple CPUs, with shared memory or distributed memory, with a slow switch or with a fast one. The size of the blocks is one of the parameters the SIP programmers can work with to get efficient execution of the same algorithm on such widely different architectures.

The hard part is indeed the recognition of what we called an "atomic data item" on which one can operate meaningfully with "atomic instructions". The reading/writing and sending/receiving instructions then follow immediately. In our case blocks of the tensors are the atomic data, and contractions the atomic instructions.

Note that the pattern requires that the programmer cannot do anything to parts of a block. Every problem area probably has a few of the "bit operation" requirements. Ours does. So we have to introduce a few special instructions to do this. For the pattern to work well, there should be few of these and they should be called sparingly.

One thing to help in this hard part is to look at the problem through "fuzzy glasses", i.e. look at the large scale things in the problem. For example, this would be a useful approach to apply this pattern in many modern multiscale simulations.

The pattern will not work well, if the data morphs a lot during the computation. If it morphs in stages, the pattern may work inside the stages. But if the data must be operated on as a list in loop 1, and a matrix in loop 2, and something else in loop 3, then there is no way to make it a useful "atomic data" item.

The standard patterns[6] of data distribution, client/server, etc can be applied to the problem at the *microcode* level and at the SIAL level.

2. Analysis of the CCSD method

The CCSD data structures scale like N^4 and the number of operations scales like N^6 (N being the user’s problem size). The method uses an iteration scheme to solve a set of nonlinear equations. As an example, consider the energy only calculation. The variables are the cluster amplitudes T. At each step, a new set of amplitudes R are constructed. When the difference between the old and the new amplitudes is smaller than some convergence criterion, the equation is considered solved and the last set of amplitudes are taken as the solution. The expressions for the new amplitudes in terms of the old amplitudes are very involved and the number of amplitudes is large and therefore iteration takes a significant amount of time and memory. A typical CCSD calculation requires 10 to 15 iterations to converge. One of the most time and memory consuming terms is

$$R^{ab}_{ij} = \sum_{cd} V^{ab}_{cd} T^{cd}_{ij}$$

Here the indices i and j range from 1 to 20 for small systems and up to 100 for midsize systems, with the indices a, b, c and d ranging from 1 to 100 and up to 1,000, respectively. This means that there are 20x20x100x100= $4\ 10^6$ to 100x100x1,000x1,000 = 10^{10} T amplitudes and 10^8 to 10^{12} “two-electron integrals” V. This is 32 MB to 80 GB for T and 800 MB to 8 TB for V.

The integrals can be computed in advance, but for many chemical systems there are too many integrals to store so they are computed as needed. The algorithm to compute the integrals is itself quite complex and the computation of one integral usually takes a millisecond or more. However, the evaluation of integrals involves recursion among several related integrals and obtaining a group of integrals at once is far more efficient than computing them individually.

The use of the pattern implies that the data structures should not be viewed as arrays of floating point numbers, but as lists or arrays of blocks. With the above examples a good block is made by dividing all index ranges into segments of length 10 to 50. Then blocks of T and V each contain 10,000 to 6,250,000 floating point numbers, or 80 KB to 50 MB.

Next the pattern calls for the algorithm to be written in terms of block operations, not operations on individual floating point numbers. This means that the double loop above becomes a loop over block contractions. We define the contraction of one block of V with one block of T, which produces one block of R, as one contraction instruction. This instruction can be implemented as a DGEMM and involves $2x100^3$ to $2x2{,}500^3$ floating point operations. On a 1 GHz processor, this operation takes from 2 milliseconds to 16 seconds. The pattern also calls for computing integrals in blocks, which, as discussed above, is natural for their evaluation anyway. We therefore define computation of one block of integrals to be a single *super instruction*. The time to execute this integral instruction is comparable to the time to perform one block contraction of V with T.

Finally the pattern suggests thinking of all distributed memory of the parallel computer as accessible by read and write *super instructions* and include the communication delay as part of the time needed to complete the instruction. With a switch fabric capable of transmitting at 100 MB/s, it takes less than 1 ms to .5 sec to transmit one block of T or V. In the case of V the latency of the switch, from 100 microseconds for Gigabit Ethernet to 1 microsecond for Infiniband, is not relevant.

The time for each *super instruction* can be tuned by changing the segment size and thus the block size. It then becomes reasonable to expect that every read from and write to some other task using MPI communication can be tuned to be less than the time for a block operation. 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. In the example above of segment sizes of 10 to 50, the time to execute a computation *super instruction* and the time to execute a memory *super instruction* rage from similar to an order of magnitude different. It is likely that the segment size can be reduced in the larger example without jeopardizing the hiding of communication behind computation.

3. Implementation of a super instruction processor

3.1. 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. Tasks are grouped into *companies*, which perform a given function cooperatively.

3.2. The I/O company

One company, called the I/O company, 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 I/O company. 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 I/O company 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 I/O company 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.

3.3 Compute companies and platoons

Other companies, called compute 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 I/O company 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.

3.4 Super instruction processing

The activity of each task in all companies, except the I/O company, is controlled by a compiled SIAL program stored in a super instruction object file. It is a list of super instructions to be executed. The super instructions can initiate communication, i.e. send or receive, or do computation (such as the tensor contraction of 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 instruction that needs the data start executing, it first checks that the communication instruction has completed successfully and it will wait (stall the SIP) if the communication is still in progress. 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.

4. Conclusion

The super instruction design pattern proved to be a very effective tool in the design of the notoriously complex problem of implementing the CCSD energy and gradients computation on parallel computers. The performance and the scaling of the parallel software designed are competitive with the code produced using traditional style MPI programming.

One of the design goals was to have software that is more flexible for tuning to varying hardware configurations and it is our experience that this goal was achieved. The SIP has been instrumented with detailed timing and memory use reporting capabilities, thus allowing us to gather detailed statistics from every run. Because of the relatively heavy nature of super instructions, the overhead of this reporting facility is negligible.

One surprising outcome is the dramatic reduction in developer time needed to implement alternative algorithms. Writing SIAL code from CCSD formulas turned out to be simpler than writing Fortran code and our programmers often implemented several algorithms to compare performance and to debug them.

It has been shown[5] by the Tensor Contraction Engine team that automatic code generation is very effective for implementing complex algorithms. Our design of SIAL was done with automatic code generation in mind. All SIAL programs we have produced so far have been written by a person. We have the code generator ready and we are investigating a more customized approach than the TCE, where the automatic code generator produces SIAL code that is tuned to the specific sizes of each chemical system, or family of systems.

References

1. J.F. Stanton, J. Gauss, J.D. Watts, W.J. Lauderdale, R.J. Bartlett, ACES II Program System, 1994, University of Florida, Gainesville, Florida, 1994.

2. G.D. Purvis, R.J. Bartlett, “A full coupled-cluster-singles and doubles model: The inclusion of disconnected triples.”, J. Chem. Phys. 76 (1982) 1910-1918.
3. J.D. Watts, J. Gauss, R.J. Bartlett, “Coupled-cluster methods with non-iterative triple excitations for restricted open-shell Hartree-Fock and other general single determinant reference functions. Energies and analytical gradients.”, J. Chem. Phys. 98 (1993) 8718-8733.
4. A.P. Rendell, T.J. Lee, A. Komornicki, “A parallel vectorized implementation of triple excitations in CCSD(T): Application to the binding energies of AlH_3, AlH_2F, $AlHF_2$, and AlF_3 dimers.”, Chem. Phys. Lett, 178 (1991) 462-470
5. Baumgartner, Gerald; Auer, Alexander; Bernholdt, David E.; Bibireata, Alina; Choppella, Venkatesh; Cociorva, Daniel; Gao, Xiaoyang; Harrison, Robert J.; Hirata, So; Krishnamoorthy, Sriram; Krishnan, Sandhya; Lam, Chi-Chung; Lu, Qingda; Nooijen, Marcel; Pitzer, Russell M.; Ramanujam, J.; Sadayappan, P.; Sibiryakov, Alexander, “Synthesis of High-Performance Parallel Programs for a Class of ab-initio Quantum Chemistry Models.”, Proceedings of the IEEE “Special Issue on program generation, optimization and platform adaptation.” (2005), 93(2), 276-292.
6. T. Mattson, B. Sanders, B. Massingil, “Patterns for Parallel Programming, Addison Wesley Professional”, 2004.