TURBULENT COMPRESSIBLE CONVECTION WITH ROTATION ON THE CRAY C-90 AND KSR-1 MACHINES.

NICHOLAS H. BRUMMELL[1]
CLIVE BAILLIE[2], **XIN XIE**[1] **and NEAL E. HURLBURT**[3]

[1] Joint Institute for Laboratory Astrophysics
University of Colorado, Boulder, CO.

[2] Department of Computer Science
University of Colorado, Boulder, CO.

[3] Solar and Astrophysics Laboratory
Lockeed Research Laboratories, Palo Alto, CA.

Introduction.

An existing code for the study of turbulent compressible convection in a three-dimensional rectilinear geometry has been ported to two very different architecture parallel machines. The original code is a vectorized Cray code employing a hybrid pseudo-spectral/finite-difference spatial discretisation and a mixture of implicit Crank-Nicolson and explicit 2-level Adams-Bashforth schemes for time-stepping the linear and non-linear terms of the underlying equations respectively. The two target machines are the Cray C-90 with 16 processors at the Pittsburgh Supercomputer Center, and the 64-processor Kendall Square KSR-1 housed at the University of Colorado as part of the Center for Applied Parallel Processing (CAPP).

The basic idea behind the parallelisation of the code for the two machines is the same and is forced by the method of solution in the code. However, the foibles of the two machines are distinctly different. Essentially, the most long range communications occur in the fast Fourier transforms (FFTs) which are necessary for the implementation of the pseudo-spectral algorithm for the two horizontal spatial directions. The third (vertical) spatial direction is discretised as second or fourth order finite-differences and is therefore much more local in its communications. The ultimate strategy is then to assign a number of horizontal planes to each processor so as to keep the FFT communications local in processor memory. On both machines, the same portable vectorizing real-to-real FFT program is used.

Code development.

a) The Cray C-90

The Cray C-90 is a dual-pipe vector processor design with fast physically shared memory (512 Mwords), rated at a peak speed of 16 Gflops. The original code with few modifications runs reasonably efficiently on this machine. A $96^2 \times 64$ version runs at 115 Mflops on a single processor of the Cray Y-MP and at 250 Mflops on a single processor of the C-90. By forcing the implementation of the FFT routines as mentioned above using the `DO ALL` directive on the C-90, and by requesting maximum vectorisation and parallelisation on the compile line, the same code can run at 1.5 Gflops across 6 processors i.e. an effective 100% linear speedup for this number of processors. However, this speedup is a plateau and only the same performance can be acheived on all 16 processors of the C-90.

We are currently striving to account for this remarkable cutoff in scalable performance. The Cray C-90 Unicos system provides a number of performance tools to aid in this analysis. The most informative tool for parallelisation purposes is “Atexpert”. When compiled and run with the appropriate options, at very little overhead the code compiles statistics as to its efficiency of parallelisation on the number of processors employed and can make predictions as to the anticipated speedup on higher numbers of processors. According to this tool, the current code is 99.8% parallel and would run at a speedup of 15.3 on 16 processors. Obviously this speedup is not achieved. Profiling the code in an attempt to track down the slowest parts of its operation, we find that the code spends 97% of its time tranposing data. Each FFT operation requires a single transpose in order to be able to do each of the two horizontal directions. It appears as if this transpose is the key bottlneck of the code. At present, the transpose is a scatter/gather operation with the optimisation achieved by the compiler, but we are currently investigating, in co-operation with the Computer Science group under Oliver McBryan, more elegant, iterative tranpose operations which would remove the necessity for this gather/scatter operation.

b) The Kendall Square KSR-1

The KSR-1 is a distributed shared memory, multiple-instruction, multiple-data (MIMD) machine with 64 processors arranged in a heirarchical "fat tree" of 32 processor rings. The shared memory is implemented in hardware as a virtual address space thus being logically shared but physically distributed. In this manner, all the memory may be considered as "cache" memory rather than "main" memory i.e. data is moved to a point of reference on demand and there is no full-time physical location for an address. This machine may then be characterised as the "ultimate" shared memory machine since truly the user does not have to worry about where the data is in memory: on demand a copy of the required data is made from its current location to the *local* memory of the requesting processor thereby significantly speeding up subsequent accesses to it. The processors are arranged in rings of 32 and so the CAPP machine here in Boulder supports two rings for a total of 64 processors. These rings are then essentially "search engines" resolving references to addresses that are not found in the processor's local memory by fetching and caching the data – hence all data movement is transparent to the user. However, despite this convenience the user now has to take care as to *how* the data is stored rather than *where*, to overcome some major caching problems.

Porting of codes to the KSR-1 could be achieved in two ways: by direct parallelisation of the sequential code via compiler directives, or by rewriting the code as a message passing code. The KSR-1 supports message passing and through use of a less well-known system, TCGMSG, can actually outperform other message passing machines such as the Intel Delta or the Ncube (Baillie, private communication). However, since the conversion to message-passing requires a complete re-write of the code, the former strategy was adopted.

Since the sequential code was reasonably optimised on a single processor, it remained to parallelise the code for multi-processors and then to optimise the resulting code. Parallelisation is achieved by "tiling" – splitting up nested loops into blocks – perhaps known on other machines as "domain decomposition". Of the four methods of tiling (slice, mod, grab and wavefront), the slice method was utilised, as previously mentioned by sending multiple horizontal planes to each processor. The loops are most efficiently implemented by tiling across the outer loops and vectorising the inner so that the contiguous elements are tiled, requiring the first array dimension to be the longest. Finally, for best use of the cache, the tile sizes should be multiples of 16, assuming that the data is 64 bit, so that the data fits evenly into an integer number of sub-pages of memory.

Only one point in the code requires major movement of data. A tridiagonal solver which couples variables in the vertical direction is used in order to solve the implicit part of the problem. Due to this coupling, this section of the code must be tiled into vertical slabs as opposed to horizontal slabs which then involves some unavoidable data migration.

Having parallelised the code, some fairly intimate details of the memory management of the machine must be observed in order to optimise its operation. Most importantly, although the data movement is tranparent to the user, it is essential that the data layout in the code be such that "sub-cache thrashing" is avoided. The sub-cache, or local processor memory, is large (256kB) but the data is positioned there according to certain bits (bits 11-16) of the system virtual address space. Thus addresses which reside 128kB apart can cause the sub-cache to swap data in and out rapidly, or "thrash". Loading of more than 2 sub-pages into sub-cache with addresses 128kB apart must be avoided at all costs else a slowdown factor of more than two may be incurred i.e. Fortran arrays with a first index length of 128 or multiples thereof must avoided so that counting on the second or higher index does not stride with this disastrous value. Note that this problem is similar to that of memory bank conflicts on the Cray and that the solution on the Cray is similar – first dimensions which are powers of 2 should be avoided. The implication of this, as we found for the current code, is that well-written Cray codes immediately avoid the sub-cache problem on the KSR.

Other strategies may be employed to make use of the caching properties of the machine e.g. array aggregation. Since the cache brings in a sub-page of numbers and not just the data requested itself, then using the already-acquired data is much more efficient than moving onto another calculation involving unrelated data. Array aggregation (or array interleaving) involves writing a single interleaved array for five variable `arr(5,nx,ny,nz)` rather than having five separate arrays `arr1(nx,ny,nz), arr2(nx,ny,nz), ...` etc.

Using such methods the turbulent compressible convection code can achieve the speed of a single Cray Y-MP processor using 32 processors of the KSR-1. In figures, for a $96^2 \times 64$ problem, this means that the code runs at 3.5 secs per time step or 120 Mflops. However, the speedup on increasing the number of processors on the KSR is not linear. We have now attributed this fact to the amount of time spent in "barriers". Barriers are those events which synchronise the processors at the end of parallel loops before entering a serial section. If the load balance across processors is not perfect then some processors will have to wait in the barrier for others to finish. On examining the barrier performance of the KSR, it appears as though the barrier can wait even though all processors have completed their individual task, and that this extraneous wait time can increase linearly with the number of processors used for this code. Eventually, the time spent in barriers starts to be a significant percentage of the calculation time e.g about 20% for 32 processors. There are two ways around this problem which we are currently pursuing: loop fusion and new barriers. The former involves concatenating loops together in a cache-efficient manner on order to cut down on the number of barriers existing in the code. The latter, a project underway in the Computer Science department under Oliver McBryan, is actually building new software barriers, but these cannot be implemented without access to the source of the KSR system and its compilers, soon to be available.

Some of these experiences are being made available for others via a tool. Sage^{++} is an object orientated toolkit for building program transformation tools, although when supplied it comes complete with many language parsers but no real tools. The experiences here have helped to develop some tools such as loop fusion, loop distribution, loop unrolling, array aggregation and array permutation tools for inclusion into this toolkit for transformation of codes to the KSR.

Conclusions.

In conclusion, we have ported a complex hydrodynamic code to two parallel machines. The first machine, the Cray C-90, is only moderately parallel and not scalable, and, being the most familiar in architecture, is relatively simple to use and serves as the control for all other computational experiments. On this machine, forefront calculations can be made in the scientific areas of interest, due to the reasonable speed and the large memory available. The second machine, the Kendall Square KSR-1, is a sensibly-designed, shared memory machine, and although much more complicated in architecture and thus more difficult to program efficiently, can now be used for medium-scale production runs. The major problems on the KSR come with management of the sub-cache, but fortunately the structure of standard Cray codes means that a lot of these problems are automatically avoided, implying that porting codes from the Cray to the KSR is relatively trouble-free. Note also that preliminary investigations have shown that the cache management involved in speeding up the KSR may also reap benefits (of the order of a factor of two) on any machine based on a Dec Alpha chip, such as the Cray T3D.