Using Parallelization Tools To Exploit Multi-Layered Hardware For Real World Application Codes

Parallel Software Products TR-2009-08-01

Abstract

With the advancement of multi-core processor technology, the task of adapting application codes so that they can effectively exploit the full power of such processors becomes crucial. The scientific community expect that the performance on such processors should reflect the number of cores available, particularly as the clock speeds of such processors restrict the single core performance.

This also leads to the increased availability of parallel systems where each node is a shared memory system (one or more multi-core processors), that together form a distributed memory machine. Significant effort is therefore required to create application codes that can feed the processing power of such systems.

To assist this process, parallelization environments such as ParaWise can be used to perform multi-level parallelizations of application codes to exploit parallelism between distributed memory nodes, within each shared memory node and between the cores of each multi-core processor. Using such an environment, parallel code can be produced in a far shorter time scale than manual parallelization without sacrificing the resultant parallel performance.

1 Introduction

The continual evolution of processor technology that has been enjoyed for many years is now reliant on the exploitation of multi-core processors. Many current systems use dual or quad-core processors and soon larger numbers of cores will be commonplace. To realize the power of such processors, some scalable form of parallelism must be exploited, representing a significant challenge to compiler and tool writers.

Whilst most of the mass-market will be satisfied by a small number of packages that have been parallelized and tuned for multi-core processors, the scientific community often have their own application codes and demand performance in relation to the number of cores available. Additionally, most high performance computing systems will consist of numerous multi-core processors, connected in a single address space or through a communication infrastructure with distributed memory, or some combination of the two.

The complexity, skill and time required to adapt application codes to effectively exploit such systems will restrict their impact, so any help compilers and tools can provide to accelerate this process should be beneficial.

The creation of effective and scalable application codes for such hardware often requires, or can be greatly assisted by, a number of factors. These include the code author who can provide their insight about the application code and information about program variables and algorithms that is not explicitly stated within that code. Also, an expert in manual parallelization can use their experience to identify parallelism within an application code and make the alterations to harness the most profitable sources of parallelism to build an effective parallel code. Additionally, the compiler can also contribute by applying optimizations for instruction level parallelism and pre-fetching etc. as well as some automatic parallelization.

Individually, they each have limitations. The code author typically specializes in a particular field of science and should not be required to become an expert in parallelization. The expert parallelizer must perform much laborious, monotonous effort along with some amount of more sophisticated work in key code sections to produce the parallel code, increasing the time and cost of such a parallelization. Such experts are also a scarce commodity, further restricting the rate of emergence of effective parallel codes.

Automatic parallelization by the compiler relies on a fairly rapid analysis of the application code with very limited or no assistance from the user, and is therefore often restricted to inner loops in loop nests leading to fine-grain parallelism that can incur significant runtime overheads. This in turn can lead to serial operation being forced when profitability is assessed during runtime. This, together with code sections where no parallelism was detected, often leads to a large proportion of the code that must operate in serial and therefore poor scalability. Whilst this may be tolerated on a dual core system, it is not acceptable when a large number of cores need to be kept busy.

In an attempt to address these issues, parallelization tools and environments have been developed [1] to automate as much of the parallelization process as possible, allowing user intervention so that the information from the code author and the compiler technology can be used to produce code of comparable quality to that of the code parallelization expert. The role of the parallelization expert can then become fine tuning the crucial sections of the application code, avoiding the other mundane tasks they currently need to perform.

2 Requirements of a Successful Parallelization

A successful parallelization should show good speedup (i.e. a speedup over a single processor core

execution time of a reasonable proportion of the number of processor cores used) and scalability onto large numbers of processor cores. To achieve this, firstly, all computationally significant code must be performed in parallel to avoid the inherent restrictions indicated by Amdahl's law, and secondly, the overheads of the parallelization methodology used must be kept to a minimum.

2.1 Keys to Success in Message Passing, Domain Decomposition Parallelization

When a domain decomposition is used to divide the computational domain over a number of processes that each have their own separate memory areas, the main runtime overheads are communication costs and any related idle time. The communication cost consists of the latency of initiating a communication and the time to transfer the data. The lowest overhead is incurred when a few large messages are sent as infrequently as possible, so ensuring communications are placed in as fewer loops as possible is crucial. Packing and unpacking data to be transferred into buffers where necessary is also essential as the cost of buffering is typically more than outweighed by the reduction in communication startup latency. The data transfer time can be further reduced on many systems by taking advantage of asynchronous communications that operate whilst other computation is being performed, with synchronization points also required to ensure that correct data is used at all times. Load balance is also critical to ensure processor idle time is kept low.

For an efficient parallelization, most communications will only involve a small subset of the full processor topology (ideally a few nearest neighbor processors) and a small subset of the communication infrastructure, avoiding the need for global synchronization and allowing communications to be performed in parallel. Reduction style operations, such as global summations and global maximum computations, do require some synchronization (either globally or in one or more dimensions of the domain decomposition). Grouping a number such operations into a single operation can reduce the runtime overhead they represent.

For a manual parallelization, the creation of correct and efficient message passing code presents numerous challenges to the code parallelizer. A detail inspection of the application code is required to identify parallelism and determine all of the program arrays that need to be distributed between the processes. The basic code alterations involve adjusting loop limits so that only the “owned” sub-section of the program data is processed, adding execution control masks (i.e. “IF (OWNER(A))” for the statement “A=. . .”) to accesses that are not within a relevant loop nest, adjusting any distributed array declarations and adding calls to a communication library. The management of such a task in a large real world application code is clearly a major challenge and the implementation is a very laborious, time consuming and error prone process. Correctly identifying the communication requirements based on actual data accesses in the code can involve examining thousands of lines of code in numerous routines and using a minimal number of rarely instigated communication calls requires a level of expertize. Many manual parallelizations are based on assumptions about the data accesses of the application code as a precise investigation is not feasible, where the simplicity of implementation, despite potentially suffering some unnecessary data transfers, makes this the preferred option.

Clearly, a manually implemented message passing parallelization has enormous scope for errors to be introduced. Although many impressive parallel code debuggers exist, the task of identifying and correcting errors can still be very demanding. In particular, the common problem of an omitted, incorrect or misplaced communication can be very difficult to trace as this can often introduce subtle numeric differences into the computations as out-of-date values can be used.

These requirements and issues have meant that the manual production of efficient message passing parallel code is a highly skilled and very time consuming process.

2.2 Keys to Success in OpenMP Parallelization

For OpenMP parallelizations, the most frequent runtime overheads relate to parallel region startup and shutdown and barrier synchronizations. The region startup costs can be reduced by starting regions infrequently and keeping them active for as long as possible, whilst the barrier synchronizations that are applied by default to parallel loops should be avoided whenever possible using the "nowait" clause on the OpenMP ENDDO directive. Only those barrier synchronizations relating to shared data where dependencies and loop distributions indicate that an assignment on a thread may interfere (i.e. provide or destroy data) with a usage or assignment on different thread must be enforced. Additionally, the selection of outer loops in a loop nest for parallel operation incurs less frequent runtime overheads.

For correct operation, frequent barrier synchronizations will be necessary involving all processors or all processors in a nesting level of OpenMP. This can represent a significant overhead and restriction to scalability to larger numbers of threads.

Although, superficially, the process of developing an OpenMP version of an application code seems far simpler than the process required for message passing, it is still a complex and error prone task. The same investigation into legal parallel execution is needed, however, to exploit a parallel loop now requires the user to determine which variables need to be shared by all threads and which must be duplicated to allow every thread to own a unique copy. To exploit the effective parallelism in an outer loop in a loop nest in a real world application code can involve inspecting numerous routines to list the variables that need to be explicitly stated as shared or private in the parallel region. With variables local to routines called within a parallel region (known as an orphaned routine) being defined as private by default and all variables in common blocks or save statements being shared by default, additional OpenMP directives and code transformations are often needed to achieve the correct variable scoping. Decisions on when barrier synchronizations are necessary can also be complex, but the incentive to remove them from inner loops is often significant due to their impact on parallel performance leading to the possibility of incorrect assumptions.

Debugging an OpenMP code is also a highly skilled task, although the use of tools to check for race conditions etc. (such as Intel's Thread Checker and Sun's Thread Analyzer) are an enormous benefit to this process. Correctly addressing the identified issues remains a challenging task for the code parallelizer.

As with manually implemented message passing parallelization, these requirements keep most effective OpenMP parallelizations of real world application codes in the realm of highly skilled experts.

3 The ParaWise Automatic Parallelization Environment

The creation of effective parallel code can be greatly assisted by a parallelization environment such as ParaWise [1,2]. It uses compiler technology to detect parallelism within an Fortran application code using an in-depth analysis whose runtime would not be commercially acceptable for a compiler, but is more reasonable in an interactive environment as it can greatly reduce the effort required from the user. This analysis builds an accurate dependence graph based on data flow and variable re-uses throughout the application code, allowing parallelism to be identified. Based on this dependence graph, ParaWise can then generate message passing (e.g. MPI) code and/or OpenMP to exploit parallelism.

3.1 Message Passing Parallelization Using ParaWise

For message passing, ParaWise uses a domain decomposition of the program data across the processor topology [3,4,5]. The user selects a single index of a single array to be decomposed

across the processors, this is then used to automatically determine the full data partition of all relevant arrays throughout the application code. The details of the data partition are not computed until runtime, based on a simple block, cyclic or block-cyclic decomposition, or on a more sophisticated graph partition [6,7]. From the decomposition of the program arrays, computations are distributed amongst the processors and communications are added to ensure correct operation [3,8]. Almost all of the complex tasks performed manually by a parallelization expert are accurately performed automatically by ParaWise. Other optimizations and/or the decomposition of further array indices can then be performed (to create a multi-dimension data decomposition [4]). The size of the processor topology to be used is determined at runtime where, for example, 64 processor cores could be used with a one-dimensional partition by specifying a "pipe64" topology, and for a two-dimensional partition, by specifying a grid8x8 or grid4x16 topology.

To create an effective parallelization, numerous techniques are employed to allow simple implementation of the parallel code and keep runtime overheads low. A number of rules are used to extend the control of which processor performs which computation from the original "owner computes" concept to cover as much code as possible. This includes call sites so that parallel loops containing call stacks can be efficiently exploited in parallel. Communication requirements are detected based on data accesses and are then migrated up the control flow graph of the application code to as earlier point in the code execution as possible, exiting loops and moving through call sites, where all related requests are merged to leave a small number of infrequently instigated communications. Asynchronous communications can be exploited as an optimization with the associated synchronization points (as required to ensure correct data usage) added automatically [9].

3.2 OpenMP Parallelization Using ParaWise

For OpenMP, ParaWise automatically detects parallelism and determines the variable scoping to exploit that parallelism [10]. The necessary OpenMP directives are automatically generated in the resultant source code, including all information about PRIVATE and SHARED data, REDUCTION operations, MASTER and CRITICAL sections and all necessary barrier synchronizations. The automatic code generation also attempts to minimize the runtime overheads in a number of ways to maximize performance. Barrier synchronizations are avoided by examining data accesses, the related dependencies in the dependence graph and the loop distribution of any relevant loops to determine if interference between threads is possible. Additionally, if a barrier synchronization is necessary, it is deferred as long as is possible, often allowing it to be placed outside of previously surrounding loops and enabling a single barrier to satisfy a number of cases that require synchronization.

Parallelism is selected in the outermost loops of loop nests, given a number of restrictions including consideration of the number of iterations a loop performs, as outer loops typically incur less frequent runtime overheads. Parallelism in loops containing deep call stacks is clearly part of the process of exploiting outermost loops and is possible due to the in-depth inter-procedural dependence analysis performed by ParaWise. Any additional directives such as THREADPRIVATE and code transformations to allow correct variable scoping in orphaned routines are also applied automatically. Parallel regions are started as rarely as is feasible by generating them outside of as many loops as possible, also moving into caller routines, and then merging the subsequent regions. This often includes some code in parallel regions that is not within a parallel loop that involves assigning shared data. To allow this, OpenMP MASTER directives are used to ensure only a single thread performs the assignment and any related synchronizations that are necessary are added automatically.

3.3 User Interaction In the ParaWise Parallelization Environment

All the stages in both parallelization processes can be assisted by the user, where browsers explain the decisions made in the automatic parallelization and techniques are provided to enable the user to

address those issues [11]. Figure 1 shows the ParaWise “Why Communication” Browser, in this case listing the 69 usages that led to the selected communication to allow the user to investigate, and also the ParaWise “Why Directive” Browser, in this case listing the reasons why the selected loop was not chosen for parallel operation using OpenMP. Additionally, facilities to import runtime profile information are provided to focus the users attention on those code sections that are having an adverse effect on the overall performance. The ParaWise Expert Assistant [12] can also be used where many of the issues relating to data dependencies or computation distribution etc. are automatically analyzed to allow questions to be posed on the application code algorithm and variables alone and presented to the user. For the communication and loop selected in Figure 1, the Expert Assistant is accessed by pressing the “Investigate” buttons.

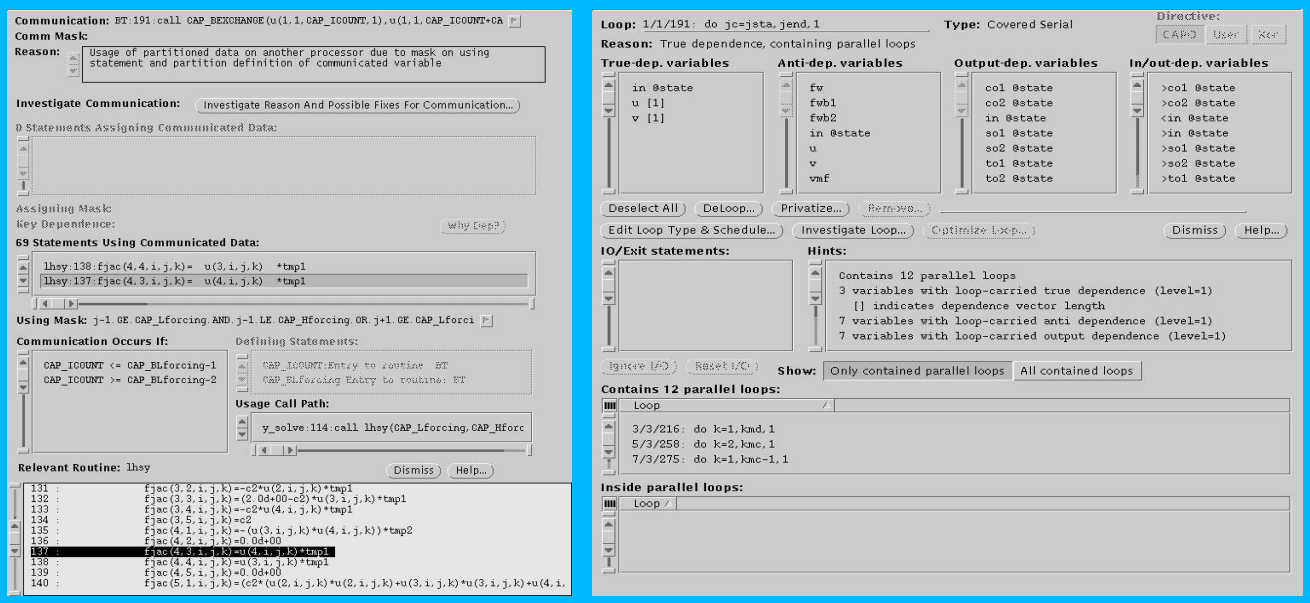

Figure 1: ParaWise “Why Communication” Browser detailing the reasons for the selected communication being generated, and the ParaWise “Why Directive” Browser detailing the reasons that the selected loop was not chosen for OpenMP parallelization.

User interaction is often vital as any non-trivial serial code section or a single expensive communication or a single frequently enforced OpenMP barrier synchronization can severely limit scalability, where minimal user intervention can often alleviate the issue.

4 Multi-Level Parallelization using ParaWise

Multi-Level parallelization is often required to feed the processor cores of large parallel systems. This can typically consist of using message passing to exploit parallelism between the distributed memory nodes of the machine, using OpenMP or message passing between the processors within each shared memory node and using OpenMP or message passing between the processor cores within each processor. If the number of processors and/or cores per processor is relatively small, each node can be treated as a single level shared memory system. Although message passing could be used for all levels of parallelism, typically, the flexibility of OpenMP parallelization in terms of parallel loop selection in a nest of loops and simple load balancing loop schedules make it a more attractive choice when a single address space is available. The resultant multi-level parallelization can therefore be either a multi-dimension message passing parallelization, a nested OpenMP parallelization or a hybrid of message passing and OpenMP parallelization. Examples of automatically generated parallel code for each case are shown in the ParaWise Browsers in Figure 2. The cost of barrier synchronizations in OpenMP often restrict its scalability so using a multi-level parallelization allows a reasonably small number of threads to be used for each OpenMP nest level, even when a large number of processor cores are to be used. The current requirement for OpenMP parallelization of the inner level parallel regions being created within parallel loops of the outer

level does (as shown in Figure 2), however, greatly restrict the performance by implying frequent parallel region start and stops making nested OpenMP only viable where the inner level loops consist of sufficient computation to allow the parallelism to outweigh these overheads.

Figure 2: Two dimensional partition message passing parallelization, nested OpenMP parallelization, thread un-safe communication hybrid parallelization and thread-safe communication hybrid parallelization.

Using ParaWise to create a multi-level hybrid parallelization should start by implementing the more rigid message passing parallelization and then adding the more flexible OpenMP parallelization. Firstly, a message passing parallelization in one or more dimensions can be performed and the resultant parallel code then passed through the OpenMP code generator, exploiting a single level or nested OpenMP parallelism. A single level of OpenMP parallelism can take advantage of large parallel regions with less frequent region starts as they can cover the message passing parallel loops (avoiding the restriction for nested OpenMP). For implementations of MPI that are not thread safe, performing the OpenMP phase after the message passing phase allows the user to specify that loops containing communication calls must not be executed in parallel using OpenMP. This order of implementation is preferable to performing an OpenMP parallelization first and then adding a message passing parallelization as the OpenMP parallelization would select the outermost parallel loops available in each loop nest where many of these loops would also be exploited in the message passing parallelization as that will need to exploit a specific loop in each nest. Consideration of non-thread safe message passing would also be problematic as some communications would almost inevitably be necessary within a surrounding OpenMP parallel loop (e.g. where the assignment and usage of data requiring communication exist within an OpenMP parallel loop).

The message passing communications that are within parallel regions, but not OpenMP parallel loops, are controlled by the MASTER directive with any necessary barriers added to ensure correct data movement and usage. For message passing communications within OpenMP parallel loops, as

is legal with thread safe communication libraries, individual thread to thread communications require additional marking (such as message tags in MPI) to ensure the correct message reaches the correct thread. Figure 2 shows examples of OpenMP for thread un-safe and thread safe communications in the ParaWise Directive Browser. For both versions, the parallel region is automatically defined in a calling routine. For the thread un-safe communication version, the inner loop has been selected for parallel execution and the communications are controlled using the OpenMP MASTER directive as they are within the parallel region. For the thread-safe version, the outer loop is selected for parallel execution and the communications will operate between matching threads as they are within this loop.

The development of such a multi-level parallelization can be simplified by building it in a number of stages. Firstly, a single dimension of the message passing parallelization can be performed and optimized. This should be repeated for any other dimensions for which the user wishes to exploit message passing parallelization with the initial tuning based on the newly partitioned dimension only (e.g. using a processor topology of "grid1x64" to avoid the influence of the first dimension partition). The OpenMP parallelization can then be added and the resultant code optimized based on executions involving a single message passing node (e.g. using a "grid1x1" processor topology). The use of ParaWise makes it far more feasible to experiment with differing message passing dimensions and alterations to the loop selection in OpenMP etc. in order to construct the most effective parallel version of the application code. Additionally, the choice of OpenMP or message passing for each level of parallelization can be experimented with as the best combination may be application code and hardware dependent.

5 Parallelization Strategies for Commonly Used Classes of Application Code

The choices of strategy for commonly used, well explored, classes of data structure in application codes can have a significant impact on parallel performance.

5.1 Structured Meshes in a Single Block

This class of code use a three-dimensional mesh with most computations nested in at least three loops, one for each dimension (where the ParaWise loop Factorization transformation can automatically extract these loops if a linearized form were used in the application code implementation). A common example of this class of code is a finite difference code.

For a message passing parallelization, a single dimension parallelization would only exploit parallelism in one of the loops in each loop nest, restricting the number of nodes (shared memory nodes or processor cores) that can be exploited and increasing the potential for load imbalance due to the relatively low number of iterations performed. If a second dimension parallelization is added, two of the loops in the loop nests can be exploited, greatly reducing the restriction on the number of nodes that can be exploited and the potential imbalance. Additionally, the communication time needed as compared to the computation time should be proportionally less for a two dimensional partition (i.e. since the surface area of sub-domains is a smaller proportion as compared to the volume). For large parallel systems, a multi-dimensional decomposition is essential if message passing is to be used for all levels of parallelization. Many ParaWise parallelizations exhibit fairly good scalability (for example, speedups of 40-60 on 64 processor cores in a number of application codes where with a sufficiently large data set).

For OpenMP parallelization, the loop nests often provide a simple source of nested OpenMP parallelism, or hybrid parallelism with some loops operating in parallel via message passing and others using OpenMP. For a single level OpenMP parallelization, parallel code generated by ParaWise can exhibit reasonable scalability (for example, speedups of 15-25 on 32 processor cores,

with some scalability on 64 processor cores), although typically not exhibiting scalability as good as for a comparable message passing parallelization.

[Note: Results for an application using 1D message passing, 2D message passing, OpenMP, nested OpenMP and hybrid message passing +OpenMP to be included here]

5.2 Unstructured Mesh in a Single Block

This class of code operates over a mesh of entities whose interconnection is defined by other data structures, such as finite element application codes.

For message passing, the entities (e.g. finite elements) can be distributed across the processor topology using graph partitioning utilities such as Jostle [6] or Metis [7] in an attempt to minimize load imbalance and interprocessor communication. Such codes often consist of a sufficient number of entities to divide the mesh across all the processor cores of a parallel system and this has enabled numerous efficient parallelizations to be performed [13,14]. For Finite Element codes, the computation time per element is typically fairly small so the potential for exploiting parallelism using OpenMP for loops used to process a single element are often fairly limited. For an OpenMP only or hybrid parallelization, the same loop over entities would typically need to be exploited for any message passing parallelism and the OpenMP parallelism.

For a single level message passing and OpenMP, codes generate by ParaWise can exhibit similar speedup and scalability to structure mesh codes.

[Note: Results for an application using message passing, OpenMP and any possible hybrid to be presented here]

5.3 Multi-Block Application Codes

Codes in this class consist of a number of separate block meshes consisting of structured or unstructured meshes that are interconnected using additional data structures. To exploit large parallel systems, parallelism between blocks and within blocks usually needs to be harnessed. Some parallelizations have avoided a multi-level parallelization by splitting larger blocks into sufficient numbers of smaller blocks to provide work for every processor core, although this can have the disadvantage of decoupling the solution, slowing convergence.

The exploitation of parallelism between blocks can be performed using OpenMP or message passing, where the only message passing communications relate to the data required for interconnected block interactions. An example of a nested OpenMP parallelization of such a code is the TFS application code from RWTH Aachen [15,16] where the large amount of work for each block meant that the relatively frequent parallel region starts required for the inner level of parallelism (the outer loop of each 3D block computation) did not significantly restrict performance. The automation of the message passing parallelization of such a code is still under development in ParaWise, however a number of application codes with a manually implemented message passing parallelization have had OpenMP parallelizations within blocks automatically added by ParaWise. A prominent cases is that of INS3D from the NASA Ames Research Center [17] where the multi level parallelization showed a significant improvement in speedup and scalability over the original single level inter-block parallelization.

5.4 General Application Codes

For codes that do not fit into the above classes, message passing and/or OpenMP parallelization can still be effective. The ParaWise message passing relies on a data domain (one or more arrays) that

can be distributed across nodes, where that also evenly distributes all the related computations (and those computations include all computationally significant code sections) without requiring excessive communications. If the application code has large arrays that can be partitioned to achieve this, message passing parallelization is a possibility. The flexibility of OpenMP allow it to be effective in achieving a reasonable parallelization to a very wide range of application codes. Effective hybrid and other multi-level parallelizations depend on whether message passing could be effectively implemented and if several loops exist in nests surrounding all the computationally significant code sections.

6 Dynamic Load Balancing

For any large parallel system, load balance is an essential attribute of the parallel computations where, in particular, any node or processor core performing more work than the others can lead to significant idle time throughout the machine. To resolve such issues, either the workload can be moved or the number of threads devoted to a particular computation can be altered.

For systems without a single address space, movement of workload from busy nodes to less busy nodes requires the adjustment of the data partition. This involves the transfer of all information relating to the portions of the program data that have been re-assigned to a different node. Based on measurement of CPU time or idle time, the amount of work that needs to be moved to improve load balance can be determined. For a structured mesh block decomposition, this can require rows, columns or surfaces of cells to be moved if the regular nested loop structure of the application code is to be preserved, whilst the inevitably introduced non-neighbor communications can be handled by more sophisticated communication call utilities [18]. For unstructured mesh codes, individual entities can be migrated (e.g. individual elements for a finite element code) to improve load balance with the selection of entities to be moved made by a graph partitioner given weightings based on runtime [19].

When a single address space is available, the simplest option is to use the OpenMP dynamic runtime loop schedule. This is practical for outer loops involving large numbers of iterations that are exploiting parallelism where an improved load balance outweighs the extra overhead of this OpenMP schedule. In many cases, a more sophisticated approach is necessary. For a nested OpenMP parallelization, the number of threads used at the second level by each thread of the first level can be varied to improve load balance [20]. In either case, the implementation is far simpler than for the distributed memory case.

7 Massively Parallel Multi-Core Processor Architectures

The precise strategies for exploiting massively parallel multi-core processors heavily depend on the nature of those processors and in particular their speed of access to shared memory. The current leading edge processors such as the Intel 80-Core research processor [21] and the SPARC 16-Core processor [22] provide some early insight into how massive on-chip parallelism may be exploited.

Although a single address space allows for the OpenMP programming model, the early experience and Teraflop performance reported for the Intel processor uses message passing between cores. If this remains the optimal way to exploit massively parallel multi-core processors then a domain decomposition message passing paradigm between processor cores may be necessary at the fine-grain, inner loop level of the application code. This may also make using shared memory paradigms between processors undesirable (i.e. where each core of a processor has access to shared memory with one core from every other processor) as the processor cores will need to make frequent access to shared memory, so a full message passing parallelization may be required.

Other techniques that may allow shared memory paradigms to be effective, such as tiling, would be

implemented manually or, preferably, automatically by the compiler in commonly occurring simple loop nests. As well as improving cache usage, such optimizations can add the additional overlap area to tiles to allow message passing to be used between cores within a processor, in a shared memory parallelization.

For the present, our experiences indicate that the shared memory performance of the current quad-core processors allows the more flexible OpenMP parallelization to be used in most cases.

8 Conclusions

Parallelization environments can offer considerable assistance to the challenging task of adapting application codes to effectively exploit large multi-core processors and multi-layered parallel systems. Definitive answers for the selection of sources of parallelism and parallelization methodology cannot be given as the nature of the application code and the data provided for an execution have a fundamental influence, along with the precise nature of the hardware to be used. Using automated parallelization allows experimentation with differing options to enable the selection of the best performing version, where such an investigation may not be feasible using manual parallelization.

9 References

1PSP Inc. The ParaWise Automatic Parallelization Environment. www.parallelsp.com

2C. Ierotheou, S.Johnson, P.Leggett, M.Cross,E.Evans, H.Jin, M.Frumkin, J.Yan, “The Semi-Automatic Parallelization of Scientic Application Codes” Scientific Programming, Volume 9, Numbers 2-3, pp 163-173, 2001

3S. Johnson, C. Ierotheou and M. Cross, “Automatic Parallel Code Generation For Message Passing On Distributed Memory Systems”, Parallel Computing 22, pp 227-258 (1996).

4E.W. Evans, S.P. Johnson, P.F. Leggett and M. Cross. “Automatic And Effective Multi-Dimensional Parallelisation Of Structured Mesh Based Codes”. Parallel Computing 26, pp 677-703, 2000.

5S.P. Johnson, C. Ierotheou and M. Cross. “Computer Aided Parallelisation Of Unstructured Mesh Codes”. In Proceedings for PDPTA 1997, Volume 1, pages 344--353, July 1997.

6C. Walshaw and M. Cross, ”Parallel Optimisation Algorithms for Multilevel Mesh Partitioning”, Parallel Computing, Vol 26, Issue 12, pp 1635-1660, Nov 2000.

7G. Karypis and V. Kumar "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs" TR95-035, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1995

8K McManus, S Johnson and M Cross “Converting Best Manual Practice into Generic Automatable Strategies for Unstructured Mesh Parallelization” Concurrency: Practice and Experience, Vol 11, pp 593-614, 1999

9E.W.Evans, S Johnson, P Leggett and M Cross, “Automatic Code Generation of Overlapped Communications in a Parallelization Tool”, Parallel Computing, 23(10), pp 1493-1523, November 1997

10H Jin, M Frumkin, J Yan “Automatic generation of OpenMP directives and its application to computational fluid dynamics codes” International symposium on High Performance Computing, Tokyo, Japan, 2000, LNCS 1940, pp 440-456.

11P.F. Leggett, A.T.J. Marsh, S.P. Johnson, and M. Cross. “Integrating User Knowledge With Information From Parallelisation Tools To Facilitate The Automatic Generation Of Efficient Parallel FORTRAN Code”. Parallel Computing, 22:259--288, 1996

12S.Johnson, E Evans, H Jin and C Ierotheou, “The ParaWise Expert Assistant – Widening Accessibility to Efficient and Scalable Tool Generate OpenMP Code” in Shared Memory Parallel Programming with OpenMP, pp 67-82, 2004, LNCS 3349

13K McManus, S Johnson, P Leggett and M Cross, “Modeling Continuum Mechanics Phenomena using Three Dimensional Unstructured Meshes on Massively Parallel Processors” in “Parallel Computational Fluid Dynamics: Recent Developments and Advances Using Parallel Computers”, pp 553-560, 1998, North-Holland Elsevier

14W Gropp, D Kaushik, D Keyes and B Smith, “Performance Modeling and Tuning of an Unstructured Mesh CFD Application”, Proceedings of Supercomputing 2000, Dallas, Texas, 2000

15S.Johnson and C. Ierotheou, “Parallelization of the TFS multi-block code from RWTH Aachen using the ParaWise/CAPO tool” PSP Inc. Technical Report TR-2005-09-02, 2005.

16S.Johnson, C.Ierotheou, A, Spiegel, D. an Mey, I. Horschler, “Nested Parallelization of the Flow Solver TFS using the ParaWise Parallelization Environment” Proceedings of IWOMP 2006, Reims, LNCS

17C Kiris, D Kwak, W Chan, “Parallel Unsteady Turbo-Pump Simulations for Liquid Rocket Engines”, Proceedings of Supercomputing 2000, Dallas, Texas, 2000

18J Rodrigues, S Johnson, C Walshaw, and M Cross, “An Automatable Generic Strategy for Dynamic Load Balancing

in a Parallel Structured Mesh CFD Code" in "Parallel Fluid Dynamics: Towards Teraflops, Optimization and Novel Formulations", pp 345-354, Elsevier, 2000
19C Walshaw, M Cross and M Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes", Journal of Parallel and Distributed Computing, 47(2), pp 102-108, 1997
20A Spiegel, D an Mey and C Bischof, "Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing", Applied Parallel Computing, LNCS, pp 433-441, 2006
21T Mattson, R van der Wijngaart and M Frumkin, "Programming the Intel 80-Core Network-on-a-Chip Terascale Processor", in Proceedings of Supercomputing 08, 2008
22M Tremblay and S Chaudhry, "A Third-Generation 65nm 16-Core 32-Thread Plus 32 Scout-Thread CMT SPARC(R) Processor", ISSCC 2008