Introduction

An industry standard for shared memory parallel programming has long been anticipated. From the moment it was announced late 1997, OpenMP was it. This set of directives and library routines, agreed upon by a powerful consortium of hardware vendors, provides application developers with a vendor-neutral API for parallelizing their Fortran, C and C++ codes to execute on the popular SMPs. As such, it has been quickly accepted by the community.

OpenMP has its roots in the efforts of the PCF Forum and the ANSI X3H5 committee, both of which met in the late eighties in an effort to provide just such a standard. Although much hard work went into the development of language features – for use with Fortran back then – the timing was wrong: vendor attention became firmly focussed on distributed memory computers and their APIs. The OpenMP developers did not just resurrect this effort. They selected the best of its features, producing a structured language with constructs that not only support the most popular elements of traditional shared memory programming models, such as parallel loops: they also enable incremental application development, conditional compilation, and the parallelization of large program regions. Their attention to such details has enabled the use of this API for rapid application development, with major programs sometimes being converted in a matter of hours.

OpenMP is based upon the traditional fork-join model of shared memory parallel programming, in which a single master thread begins execution and spawns worker threads to perform computations in regions that are marked by the user as being parallel. Directives are provided with which the programmer may indicate how the work in such regions is to be distributed among the executing threads and how threads are to be synchronized. Innovative features include orphan directives that are dynamically, but not lexically, within the scope of a parallel region. OpenMP does not demand a radical change in programming style. A serial program can be successively annotated with its directives, leaving the sequential code largely undisturbed. The compiler performs a relatively straightforward translation of these, usually to calls to a thread library.

The vendor community has established a non-profit organization called the OpenMP Architecture Review Board, or ARB, to deal with the further development of OpenMP. They have addressed problems with the initial specification, as well as defining features that help to keep it up-to-date. Few modifications have been found necessary so far, although a major revision to the Fortran API has greatly facilitated its use with Fortran 90. The user community is beginning to organize itself in the form of a User Group to address their concerns and liaise with the ARB.

This special issue of Scientific Programming presents a cross-section of papers that reflect current OpenMP activities. Several papers report on experiences using this paradigm to solve their computational problems, each of them from a somewhat different point of view. Further papers discuss tools to support the creation and optimization of OpenMP programs. The compilation of OpenMP, both in its traditional setting and for use with software distributed shared memory is described, as are on-going research efforts that aim to provide additional power within the language. The contributions were chosen from a variety of papers that were presented at two distinct workshops on OpenMP, its application and its tools, that wereorganized by the editors. One of these was held in San Diego in July 2000 and attracted a variety of participants from government labs, universities and industry. The other workshop took place in Edinburgh, Scotland, in September of the same year and attracted a similar range of attendees. It was interesting to witness the diversification that had taken place since the initial workshop the previous year, when few implementations were available. A surprising number of compilers and tools on a variety of platforms were already in use by the summer of 2000.

Application developers and parallelization experts are still learning how to use OpenMP. Experiences have been reported on a variety of codes and target systems, and general strategies proposed. In this issue, Kaiser and Baden discuss how to get performance out of the hybrid programming model obtained when MPI and OpenMP are combined in a code. This is particularly popular as a model for clusters of SMPs and tightly coupled machines that exhibit this hierarchical parallelism. They consider the use of OpenMP at a coarse granularity, and show how it is possible to overlap MPI communication with OpenMP computation. The paper focusses on the needs of regular (structured) stencil-based applications. Their results compare strategies in which a thread is reserved to perform communication with those that attempt to share both communication and computation among all threads. The usefulness of blocking for cache is demonstrated and the claim made that moving OpenMP parallelism to outer levels of the code will enable cache blocking without interference from these directives.

Smith and Bull also consider this hybrid model for SMP clusters, evaluating its overall usefulness. They show that this is not necessarily the most effective programming model but that it may provide significant benefits in some situations. This is particularly likely to be the case if the MPI code suffers from poor scaling, perhaps due to load imbalance or too fine grain problem size, or if the MPI program suffers from memory limitations due to large amounts of replicated data. The MPI implementation used may exacerbate such problems. The authors have created a QMC code that may execute with an arbitrary mix of OpenMP and MPI even on a pure SMP system. It scales well up to 32 processes under each of the programming models, yet the OpenMP version was easier to create.

Paolo Malfetti discusses the application of OpenMP to weather, wave and ocean models in his contribution to this volume. The target machine in this case is Silicon Graphic's Origin 2000, a ccNUMA architecture. The programs he works with were written for vector machines, so there are a variety of porting problems that need to be overcome. Malfetti addresses the suitability of this architecture for environmental simulations. He also discusses the benefits of OpenMP as a programming model, the need for single processor tuning in order to obtain performance under it, and the relationship between scalability and input size, cache sizes and the target processor.

Di Martino and colleagues discuss OpenMP workload decomposition strategies, based upon a case study involving a particle-in-cell code. Both domain and particle decomposition strategies are applied to the Hybrid MHD-Gyrokinetic Code and evaluated with regard to their time efficiency, memory usage and the program restructuring effort required to achieve them. The results in this case are sensitive to the relationship between the domain size and the number ofparticles. A major finding of their work is the trade-off implied between efficiency and recoding effort. A particle decomposition requires little code rewriting, but has drawbacks: one such approach leads to large memory use, and another results in comparatively low efficiency. The domain decomposition approach is superior in each of these respects but required considerable program modification. The authors conclude that ad hoc parallelization strategies, taking application characteristics into account, are required to achieve high efficiency in general.

Compiler strategies for dealing with explicitly parallel shared memory programs are comparatively new. Indeed, most OpenMP compilers refrain from optimizations that are not essentially sequential (i.e. within a single thread). Yet broader optimizations could potentially have a big impact on the performance of OpenMP codes, especially if they are translated to a ccNUMA system or target software distributed shared memory.

Sato and colleagues describe the compiler translation from OpenMP to SCASH, a software distributed shared memory system that works on clusters of PCs. They explain that data mapping is key to achieving good performance under this system and add a set of directives to specify data mapping and loop scheduling at page granularity.

In their paper on compiler optimization techniques for OpenMP programs, Satoh and colleagues extend standard dataflow analysis methods to cover OpenMP parallelism. They describe a flowgraph containing nodes to represent OpenMP constructs, and develop algorithms to handle problems such as reaching definitions, but also to analyze memory synchronizations and cross-loop data dependences. Although their goal is to target software distributed shared memory systems, much of their work on reducing synchronization overheads and improving data locality is equally applicable to generic OpenMP implementations.

Although it is easy to create an OpenMP program, it is not necessarily easy to create one that executes on a given parallel platform with high efficiency. A variety of optimizations may need to be manually applied to improve single processor performance as well as to reduce false sharing and a variety of overheads.

Park and co-authors propose a general methodology for developingparallel programs that includes the use of a parallelizing compiler to perform some of the translations. They describe a pair of independent yet complementary tools that were designed to support a set of tasks that are part of the development process. URSA MINOR is a performance evaluation tool and INTERPOL is an interactive tool for program transformation that can be used with OpenMP codes. Their work extends to consider the use of a database, storing information to support the diagnosis and possible solutions for various performance symptoms.

Ierotheou and colleagues at Greenwich as well as NASA Ames have extended a tool that supports development of MPI programs so that it can also generate OpenMP code. They break the process down into three stages: identification of parallel regions and parallel loops, optimization of these, and the actual code transformation, including the insertion of OpenMP directives into the input source program. An important aspect of this system is its extensive data dependence analysis, the results of which are used to classify loops. The user may inspect this classification and the dependences themselves, and is given the opportunity to improve upon the system's knowledge. The authors include several case studies to illustrate the workings of CAPTools.

One measure of the success of OpenMP as an API for technical computing on SMPs is the fact that there are many attempts to broaden its range of applicability.This includes proposals for features that address other kinds of computations, software to support its execution on distributed memory systems and clusters of workstations, as well as its implementation on cache-coherent NUMA systems. Work addressing each of these issues is to be found both in academia as well as in industry.

Labarta et al. consider extensions to OpenMP that support irregular data access loops. One of the advantages of OpenMP is that it can be used without difficulty to express the parallelism in unstructured computations. However, if there are dependences between the updates performed by different threads, these updates must be protected. This is unnecessarily inefficient in many cases. The paper proposes supporting an inspector/executor implementation so that those accesses requiring protection are identified and all others may be updated in parallel. They borrow from work previously performed to support HPF by adding an indirect clause and a named schedule. The authors point out the potential usefulness of the inspector/executor approach to enable runtime data dependence and corresponding construction of execution schedules, e.g. for wavefront execution.

Nested parallelism is the feature in the current OpenMP specification that has proved to be most resistent to implementation. Blikberg and Sorevik perform numerical experiments to support their claim that, if a problem has more than one level of parallelism, applying parallelism to all levels will enhance scalability. They take the inevitable load balancing issues in such situations into consideration and argue for the full implementation of this feature. The paper discusses two example codes in detail, one synthetic and one a real world use of nested parallelism. The overarching goal is to use OpenMP for adaptive mesh refinement codes, where multiple levels of parallelism will be present, albeit with difficult load balancing problems. The authors claim that many problems are typically broken down into smaller subproblems, leading to a code that can profit from multilevel parallelism; an outer layer generally has a few large tasks, and successive layers have more, but smaller, subtasks.

Happy reading!

Mark Bull and Barbara Chapman