Scientific Programming 11 (2003) 79–80
IOS Press

Introduction

This volume of Scientific Programming is the second issue that has been devoted to a collection of papers on the recent parallel programming API OpenMP. OpenMP is a set of directives with language bindings for Fortran, C and C++, along with a few library routines, that have enjoyed broad support from computer vendors. In the recent past, the original language definitions have been updated to accommodate features of Fortran 90 and to eliminate minor differences between the versions for the different programming languages.

The current collection of papers includes contributions from vendors as well as from researchers. The topics indicate the diversity of related work. As with the first issue, most of these papers have been selected from the submissions to the European Workshop on OpenMP 2001, EWOMP, that took place in Barcelona,Spain, and was organized by the European Center for Parallelism of Barcelona (CEPBA).

Our first contribution comes from one of the developers of this API: Timothy Mattson from Intel is chairman of the OpenMP Architecture Review Board (ARB), the organization that was founded to maintain the language definition. In this work, he first gives us a brief history of OpenMP and then considers how to evaluate a programming model. He points out the strength of OpenMP, the ease with which it can be used to create parallel programs, and then the difficulty with this model, the need for hard work to obtain required performance levels under OpenMP in some large computations. OpenMP is actively maintained and the ARB is working to extend the language in several ways to alleviate this problem. Mattson outlines some of the relevant work currently being performed by the ARB.

Several vendors have produced OpenMP versions of popular numerical libraries, including the BLAS. Addison and colleagues discuss their experiences when developing pure OpenMP versions of the parallel BLAS and LAPACK libraries. As part of this effort they describe problems that had to be overcome and some that still cause difficulties. One of these has to do with data locality. Data locality is of critical importance on almost all modern machines: even on shared memory platforms, it is of the utmost importance to maketo the best possible use of cache. This implies that an OpenMP user who desires to get the best possible performance must take care to use cache well throughout parallel regions. When these regions include multiple calls to OpenMP libraries, it can be exceedingly difficult to ensure consistency between cache usage.

The SPEC corporation has created a suite of OpenMP benchmarks to enable the evaluation of OpenMP on the current generation of computer hardware and their compilers. It contains 11 programs in Fortran and C from a variety of different application areas, and with both medium and large data sets. In their contribution, Aslot and Eigenmann describe these benchmarks and their behavior in considerable detail. The reasons for less than ideal parallel speedup is discussed for each of the codes separately.

In another paper related to benchmarking activities, Mueller proposes several tests that have the purpose of determining whether compilers have implemented several different optimizations. There is some tension in OpenMP between the notion of performance transparency, and the faithful translation of the user's specification, and opportunities for optimization that override those directives. For example, some constructs require a barrier synchronization unless explicitly suppressed by the programmer. If the compiler is able to determine that the barrier is unnecessary, the question arises whether it should eliminate it. There is not complete agreement on the answer to this. However, Muller's experiments are able to determine just what conclusions the compiler implementers have reached on a number of issues.

A good deal of compiler and tool research focusses on techniques designed to improve the performance of some class of OpenMP codes, on extending the range of applicability of OpenMP, especially by enabling it to run across clusters, and on assessing the need for extensions to the OpenMP language standard in order to accommodate the needs of certain kinds of application programs. This volume would be incomplete without a cross-section of papers in these areas. We have chosen three.

ISSN 1058-9244/03/$8.00

In the first of these, Barekas and colleagues discuss experiences with their own OpenMP compilation and execution environment, developed for Intel SMP systems. The environment includes a resource management system and includes features that exploit the dynamic execution capabilities of OpenMP: these permit them to adapt executables to exploit whatever CPUs the resource manager makes available to them. The system is tested using the NAS benchmarks in both dedicated mode and in a heavily used system where the application must compete with many other programs for resources.

In their work, Nikolopoulos and colleagues discuss the possibility of extending the ways in which OpenMP loops can be shared among the executing threads. They consider the needs of unstructured computations, in which the array elements that are adjacent in a mesh may not be stored contiguously. Good cache reuse can be obtained by creating a custom loop schedule and reuse it in subsequent instances of the same loop or a different loop. This ensures that threads will execute the same iterations, enabling cache reuse. There is some overhead to creating such schedules and a full solution thus requires that they can be reused.

The final contribution from this community considers the automatic generation of OpenMP programs. In their CAPO tool, Jin and colleagues have implemented analyses and transformations that are able to identify parallel loops even in the presence of procedure calls. The resulting system is able to generate OpenMP code that goes beyond the current standard in that it will parallelize multiple levels of a loop nest (if it has determined that they may execute in parallel), define groups of threads and stablish precedence relations. The availability of the Nanos research compiler supporting these extensions has enabled them to evaluate their work on benchmarks and a complete application.

The last paper in our collection describes an effort to parallelize a PIC code hierarchically for a cluster of SMPs. They consider workload decomposition strategies that map the computation to the SMPs and then, as the next level of the hierarchy, decompose the computation assigned to an SMP and map its parts to the individual CPUs within the SMP. Alternative strategies are described, analyzed and their performance predicted. The different versions are implemented using HPF to realize the higher level interaction between SMPs and OpenMP to realize the computation within each of the SMPs. Their performance is compared with the expected performance.

There is still much to learn about OpenMP, its implementation and application on a variety of platforms, and its use in conjunction with other programming models on SMP clusters, as are widely deployed. Given the variety of topics of interest, it is not an easy task to select papers for publication in a volume such as this one. We hope that this collection will prove to be interesting and useful to readers, and that the issues raised will stimulate many of them to further research in this area.

Enjoy!

Eduard Ayguade and Barbara Chapman