A Study of Performance Scalability by Parallelizing Loop Iterations on Multi-core SMPs

Prakash Raghavendra, Akshay Kumar Behki, K. Hariprasad, Madhav Mohan, Praveen Jain, Srivatsa S. Bhat, V.M. Thejus, Vishnumurthy Prabhu

Department of Information Technology, National Institute of Technology Karnataka, Surathkal
srp@nitk.ac.in, aksbeks@gmail.com, harinitk2007@gmail.com, madhavmon@gmail.com, prgu_jain@yahoo.com, bhat.srivatsa@gmail.com, thejus.vm@gmail.com, prabhuvishnumurthy@gmail.com

Abstract. Today, the challenge is to exploit the parallelism available in the way of multi-core architectures by the software. This could be done by re-writing the application, by exploiting the hardware capabilities or expect the compiler/software runtime tools to do the job for us. With the advent of multi-core architectures ([1] [2]), this problem is becoming more and more relevant. Even today, there are not many run-time tools to analyze the behavioral pattern of such performance critical applications, and to re-compile them. So, techniques like OpenMP for shared memory programs are still useful in exploiting parallelism in the machine. This work tries to study if the loop parallelization (both with and without applying transformations) can be a good case for running scientific programs efficiently on such multi-core architectures. We have found the results to be encouraging and we strongly feel that this could lead to some good results if implemented fully in a production compiler for multi-core architectures.

1 Introduction

Parallel processing requires program logic to have zero dependency between the successive iterations of a loop. To run a program in parallel we can divide the task between multiple threads or processes executing in parallel. We can also go up to the extent of running these parallel pieces of code simultaneously on different nodes in a high speed network. However the amount of parallelization possible depends on the program structure as well as the hardware configuration.

OpenMP [3] is an Application Program Interface (API) specification for C/C++ or FORTRAN that may be used to explicitly direct multi-threaded, shared memory parallelism. It is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. It has been implemented on most of the popular compilers like GNU (gcc), Intel, IBM, HP, Sun Microsystems compilers.

Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or FORTRAN source code. The use of the pre-processor directive (starting with #) along with the OpenMP directive instructs the compiler during pre-processing to implement parallel execution of the code following the OpenMP directive.

There are various directives available, one of them being '#pragma omp parallel for' whose implementation was our prime interest in the project. The '#pragma' directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. The '#pragma omp parallel for' directive instructs the compiler that all the iterations of the 'for' loop following the directive can be executed in parallel. In that case, OpenMP compiler will generate code to spawn optimized number of threads based on the number of cores available. Consider the following example of a typical C/C++ program using OpenMP pragmas:

```
#include <omp.h>
main ()  {
 int var1, var2, var3;
 /*** Serial code ***/
      .
      .
 /*** Beginning of parallel section. Fork a team of threads ***/
 /*** Specify variable scoping ***/
     #pragma omp parallel private(var1, var2) shared(var3)
        {
        /*** Parallel section executed by all threads ***/
               .
               .
        /*** All threads join master thread and disband ***/
        }

 /*** Resume serial code ***/
     .
     }
```

The variables var1 and var2 are private to each of the threads spawned and the variable var3 is shared among all the threads. The intent in this work is to study some OpenMP programs and see if these scale well on multi-core architectures. Further, we would also like to parallelize non-parallel loops by applying transformations (using known techniques like unimodular and GCD transformations [5] [6] [7]) and see if they too scale well on such architectures. We used some known OpenMP pragmas as case studies and implemented them in our compiler to study the performance. In Section 2, we describe the way we implemented these OpenMP pragmas. In Section 3, we discuss unimodular transformations which we used to parallelize non-parallel loops. In Section 4, we tabulate and explain all the results. Section 5 concludes the paper and suggests some directions for future work.