Teaching High Performance Computing Parallelizing a Real Computational Science Application

Giovanni Aloisio, Massimo Cafaro, Italo Epicoco, and Gianvito Quarta

Center for Advanced Computational Technologies,
University of Lecce/ISUFI, Italy
{giovanni.aloisio, massimo.cafaro,
italo.epicoco, gianvito.quarta}@unile.it

Abstract. In this paper we present our approach to teaching High Performance Computing at both the undergraduate and graduate level. For undergraduate students, we emphasize the key role of an hands on approach. Parallel computing theory at this stage is kept at minimal level since this knowledge is fundamental, but our main goal for undergraduate students is the required ability to develop real parallel applications. For this reason we spend about one third of the class lectures on the theory and remaining two thirds on programming environments, tools and libraries for development of parallel applications. The availability of widely adopted standards provides us, as teachers of high performance computing, with the opportunity to present parallel algorithms uniformly, to teach how portable parallel software must be developed, how to use parallel libraries etc. When teaching at the graduate level instead, we spend more time on theory, highlighting all of the relevant aspects of parallel computation, models, parallel complexity classes, architectures, message passing and shared memory paradigms etc. In particular, we stress the key points of design and analysis of parallel applications. As a case study, we present to our students the parallelization of a real computational science application, namely a remote sensing SAR (Synthetic Aperture Radar) processor, using both MPI and OpenMP.

1 Introduction

Introducing parallel computing in the undergraduate curriculum provides current students with the knowledge they will certainly need in the years to come. For undergraduate students, we emphasize the key role of an hands on approach. The study program provides students with a degree in Computer Engineering; the program can be considered at the bachelor level. We refer to just one course of the undergraduate program in this paper. We do also have master level courses (Parallel Computing I and Parallel Computing II) and Ph.D. level courses. In the undergraduate program parallel computing theory is kept at minimal level since this knowledge is fundamental, but our main goal for undergraduate students is the required ability to develop real parallel applications. For this reason

V.S. Sunderam et al. (Eds.): ICCS 2005, LNCS 3515, pp. 10–17, 2005.
© Springer-Verlag Berlin Heidelberg 2005

we spend about one third of the class lectures on the theory and remaining two thirds on programming environments, tools and libraries for development of parallel applications.

At the undergraduate level we simply introduce briefly the need for parallelism, the concepts of speedup, efficiency and scalability, and the models underlying message passing and shared memory programming. We rely on Foster's PCAM design methodology [1] when designing message passing applications, and on dependency analysis of loop variables for shared memory programming.

Performance analysis include Amdahl [2] and Gustafson-Barsis [3] laws, the Karp-Flatt metric [4] and iso-efficiency analysis. The availability of widely adopted standards provides us, as teachers of high performance computing, with the opportunity to present parallel algorithms uniformly, to teach how portable parallel software must be developed, how to use parallel libraries etc. We utilize both MPI OpenMP. The course introduces the most important functionalities available in the MPI 1.2 specification, and all of the OpenMP library. The main programming language is C.

Each student is required to parallelize, as a short project, a real application. We have found that assigning projects to groups of students does not work as expected. We thought that organizing students in groups would have fostered the key concept of collaboration, and provided fertile ground for discussions etc. This proved to be true for graduate students, whilst for undergraduates the net effect was that only one or two students per group actually did the job assigned. Therefore, we require that undergraduate students carry out individually their projects. The project is not necessarily done during the course: each student is required to present his project when actually taking the course examination (which can also happen several months after the end of the course, since we do have ten examination per year for each course). Thus, a student may work on his/her project as much as he/she needs. Likewise, we do allow up to one year for the final bachelor thesis (this differs from many universities both in Europe and USA, but is quite common in Italy); in turn we usually get very satisfactory results.

This paper presents the parallelization made by one of our undergraduate students of a real computational science application, namely a remote sensing SAR [5] raw data processor, using both MPI and OpenMP. SAR processing [6] applies signal processing to produce a high resolution image from SAR raw data. High image precision leads to more complicated algorithms and higher computing time; in contrast, space agencies often have real-time or near real-time requirements. As matter of fact, SAR processing algorithms are computationally intensive and require fast access to secondary storage. In order to accelerate the processing, SAR focusing has been implemented on special purpose architectures and on HPC platforms. Nevertheless, special purpose architectures have relatively high cost, when compared to HPC platforms that are now becoming increasingly popular for this task. The paper is organized as follows. Section 2 recalls the SAR processor application and the rules of the parallelization contest we organized. Section 3 describes the winning parallel SAR processor and Section 4 concludes the paper.

2 SAR Image Processing

The SAR sensor is installed on a satellite or aircraft that flies at constant altitude. SAR works transmitting a beam of electromagnetic (EM) radiation in the microwave region of the EM spectrum. The back scattered earths radiation is intercepted by the SAR antenna and recorded. The received echoes are digitalized and stored in memory as a two dimensional array of samples. One dimension of the array represents the distance in the slant range direction between the sensor and the target and it is referred to as the range direction. The other dimension represents the along-track or azimuth direction.

The main goal of SAR processing is to reconstruct the scene from all of the pulses reflected by each single target. In essence, it can be considered as a two dimensional focusing operation. The first, relatively straightforward, is range focusing; it requires the de-chirping of the received echoes. Azimuth focusing depends upon the Doppler histories produced by each point in the target field and it is similar to the de-chirping operation used in the range direction. This is complicated however by the fact that these Doppler histories are range dependent, so azimuth compression must have the same range dependency. It is also necessary to correct the data in order to account for sensor motion and Earth rotation.

SAR focusing has been implemented, generally, using the classic range-Doppler algorithm [7] or chirp-scaling algorithm [8]. The range-Doppler algorithm does first range compression operation and then azimuth compression. During azimuth processing, a space-variant interpolation is required to compensate the migration of signal energy through range resolution cells. In general, interpolation may require significant computational time.

The AESAR package, a sequential range-Doppler SAR image processor developed by the Italian Space Agency, has been selected for our last year parallelization contest. The contest rules for undergraduate students were: (i) students can freely decide how to parallelize the code, (ii) modifications to the legacy code must be kept at a minimum due to engineering costs, and the target architecture is an HP AlphaServer SC machine. This machine is a cluster of SMP nodes, and each node contains four alpha processors. For graduate students the target machine was an HP RX6000, a cluster of Itanium 2 nodes, each node containing two processors, and the code could be refactored and reengineered as needed. We describe now the chosen computational science application and how the sequential range-Doppler algorithm works. This is is the most widely used algorithm for SAR focusing. We describe first the sequential algorithm. The core steps of range-Doppler algorithm follow.

After raw data have been read, the image frame is divided into blocks, overlapped in azimuth direction. Then, a Fast Fourier Transform (FFT) is performed in the range direction; subsequently range compression is performed through a complex multiplication of the range lines with a range reference function. The range reference function is obtained from Doppler rate, extracted by a parameter file. Finally, an IFFT (Inverse FFT) is performed in the range direction. Before azimuth FFT, the corner turning operation must be performed. It con-

sists of a transposition of the memory arrangement of 2-dimensional array of data. Then, the FFT in azimuth direction is performed, followed by range cell migration correction which requires a shift and interpolation operation. The azimuth compression requires a complex multiplication of the azimuth column by the azimuth reference function. The azimuth reference function is calculated for each azimuth column, using the Doppler centroid value estimated before. Finally, an IFFT in azimuth direction is performed to complete the focusing process.

3 Parallel SAR Processor

After a careful analysis of the sequential algorithm, the student decided to instrument and profile code execution in order to determine computationally intensive numerical kernels. He found that the majority of the time is spent on Range and Azimuth Compression. According to the range-Doppler algorithm, the student then proposed an hybrid parallelization approach. Course grain parallelism for this application entails distributing the image frame segments to MPI processes. The entire raw image frame is divided into a fixed number of segments, and for each segment range and azimuth compression is computed sequentially. The segments are independent of each other and partly overlapped as needed by the focusing algorithm. The size of the overlap region is imposed by phisical constraint on the processing.

Fine grain parallelism, usually not suitable for MPI applications, is instead effective using OpenMP. Therefore, our student parallelization strategy distributes the lines belonging to a given segment to available threads. Given a segment, both range and azimuth compression are computed in parallel, one after the other. The hybrid MPI/OpenMP approach takes advantage of the benefits of both message passing and shared memory models, and makes better use of the proposed architecture, a cluster of SMP nodes. Indeed, since the number of segments is fixed, so is the number of MPI processes. In such a situation, requiring a specific number of processes severely limits scalability. Instead, the simultaneous use of OpenMP allows exploiting additional CPUs: the natural MPI domain decomposition strategy for the application can still be used, running the required number of MPI processes, and OpenMP threads can be used to further distribute the work among threads.

The frame-level parallelization has been implemented using MPI. To optimize the performance, the student made segment computation independent from other segments. Indeed, he tried first sending the overlapped lines needed by a segment computation to the process in charge of that segment. Even though the communication network was a Quadrics QS-Net, he found that for this application and target machine it is best to avoid inter-node communication. This of course leads to an implementation that includes redundant computation: to process each segment independently from the others, it is necessary that each process is also responsible for the rows in the overlap region. Then, for the MPI implementation there is no communication overhead.

The image segmentation mechanism must satisfy the following requirement: the size of segments, must be greater than the number of overlapped lines, because this is the length of the filter used to process raw data in azimuth direction. Moreover, a bigger segment size implies reduced performances due to the FFT routines. This leads to a total of nine segments. The constraint on the number of segments entails that when the number of MPI processes does not divide evenly the number of the segments, the computational load is not balanced properly and so the parallel algorithm should include a load balancing strategy.

The segment-level parallelization model has been implemented using OpenMP. The student correctly identified and removed loop carried dependencies in order to parallelize loops. In order to achieve better performances, the student tried to minimize parallel overhead. The main issue and source of overhead is the presence of critical sections, where multiple threads potentially can modify shared variables. The student minimized this overhead partially rewriting the sequential code so that each thread, when possible, has its own copy of the variables, even though the approach entails the use of additional space. Other factors that contribute to parallel overhead are: (i) parallel directives used to execute parallel loops, (ii) the loop scheduling to balance the computational load and the atomic construct used to provide exclusive access to variables being updated; (iii) accesses to different locations in the same cache line (set of entries in a single cache location). The former two sources of overhead increase linearly with the number of threads involved. The latter depends on the number of threads that read and/or write different locations in the same cache line and on the amount of data assigned to each thread.

3.1 Parallel Model

Here we describe the student model for this application that predicts the parallel time when using p MPI processes and t OpenMP threads. Given

- n number of segments;
- r total number of rows;
- c total number of columns;
- o number of overlapped rows between contiguous segments;
- T_i time spent for data inizialization, Doppler evaluation;
- T_{ec} time spent for echo correction for one row;
- T_{r_conv} time spent to compute the convolution between one row and chirp signal;
- T_{a_conv} time spent to compute the convolution between one column and estimated chirp signal along range direction;
- T_{rcm} time spent for range cell migration correction for one azimuth column;
- T_{file} time spent to write a line to file.

$$T(p,t) = T_i + \left\lceil \frac{n}{p} \right\rceil (T_{range} + T_{azimuth}) \tag{1}$$

where T_{range} is defined by:

$$T_{range} = (T_{ec} + T_{r_conv})(\frac{r}{n} + o)\frac{1}{t} \quad (2)$$

and $T_{azimuth}$ is

$$T_{azimuth} = (T_{rcm} + T_{a_conv} + T_{file})\frac{c}{t} \quad (3)$$

These parameters have been evaluated profiling the application. The sequential code exploited the traditional Cooley-Tukey FFT algorithm. The student was aware, due to class lectures, that better alternatives exist. He substituted the FFT calls with the corresponding functions from the FFTW library [9] and estimated that performances are better for 4096 complex elements. Considering this, he fixed the number of segments (nine).

The model has been validated against experimental runs of the application in order to assess its ability to predict the parallel time, and related measures such as speedup and efficiency, as shown in Figures 1, 2 and 3. The application was run varying the number of MPI processes from one to three, and the number of threads per process from one to four, since the parallel queue available to students on the target machine is made of three SMP nodes, each one containg four CPUs. As shown in Figure 1, the model correctly approximates the parallel execution time; in particular, the slightly superlinear speedup obatined when using a single MPI process and a varying number of OpenMP threads, up to four, is due to cache effects. Finally, when using two MPI processes and four OpenMP threads, for a total of eight CPUs, we observe a decrease of efficiency. This is expected, since in this case the computational load is not perfectly balanced because one process is responsible for five segments, whilst the other gets the remaining four segments.

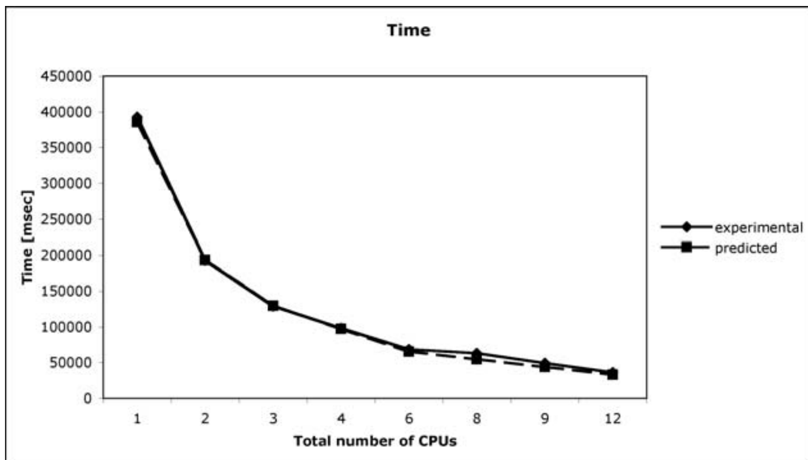

Fig. 1. Parallel Time

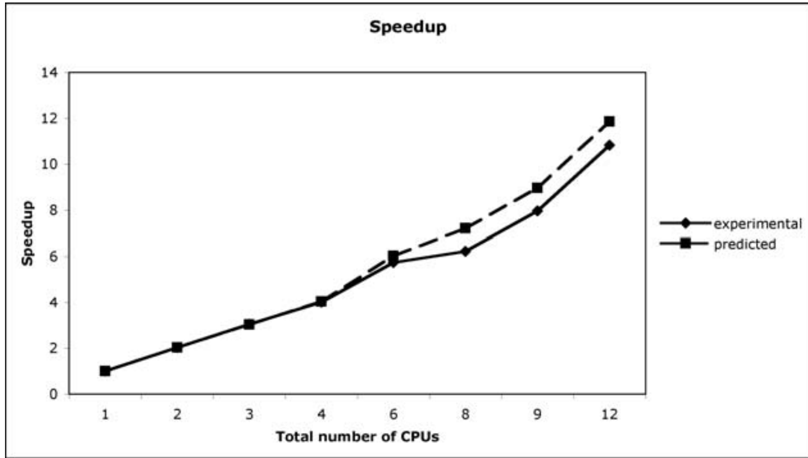

Fig. 2. Speedup

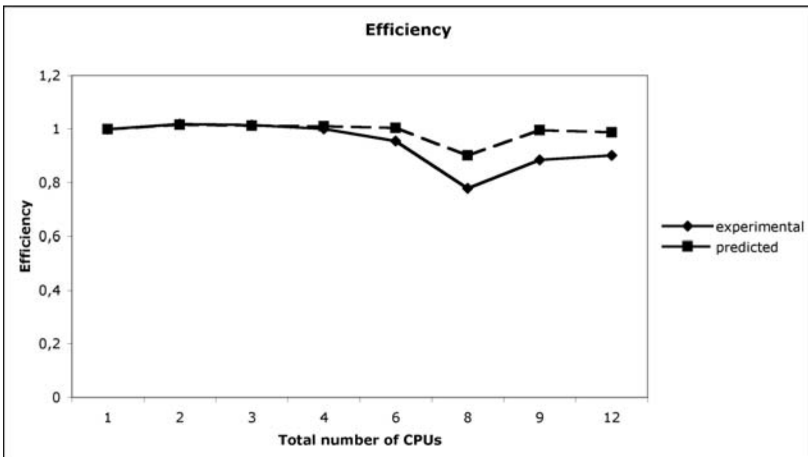

Fig. 3. Efficiency

4 Conclusions

In this paper we have described the parallelization of a real computational science application, SAR processing, reporting the experience of an undergraduate student parallelizing a range-Doppler legacy code using an hybrid MPI/OpenMp approach. When the students are given enough time, the experience reported in this paper is a good representative of average outcomes for this HPC course. The one-student team approach was feasible because students had enough time (several months if needed) to complete their homework project. And it was interesting to see that students did not require too much help from teachers/assistants.

Moreover, cooperation was explicitly forbidden during the project: there is no point in having one student teams if students can collaborate. However, exchange of experience is always beneficial and we do allow this during the course. We have found that, besides teaching traditional examples of parallel applications such as matrix multiplication etc, students like the hands on approach we use in our Paralleling Computing course. The parallel contest we organize as part of the course proves to be extremely useful, especially for undergraduate students to better understand parallel computing theory and related practical issues. The student was able to parallelize the proposed application and to correctly model its parallel execution time, thus meeting the main goals of the course.

References

1. Foster I.: Designing and Building Parallel Programs, Addison-Wesley, 1995
2. Amdahl G: Validity of the single processor approach to achieving large scale computing capabilities, Proc. AFIPS, Vol. 30, pp. 483–485, 1967
3. Gustafson, J. L.: Reevaluating Amdahl's law, Communications of the ACM 31(5), pp. 532–533, 1988
4. Karp A. H., Flatt H. P.: Measuring parallel processor performance, Communications of the ACM 33(5), pp. 539–543, 1990
5. Elachi, C.: Spaceborne Radar Remote Sensing: Applications and Techniques, IEEE Press, 1988
6. Barber B.C.: Theory of digital imaging from orbital synthetic-aperture radar, INT. J. Remote Sensing, 6, 1009, 1985
7. Smith A. M.: A new apporach to range-Doppler SAR processing, Journal Remote Sensing, 1991 VOL. 12, NO 2, 235-251
8. Raney R.K., Runge H., Bamler R., Cumming I.G., Wong F.H.: Precision SAR Processing Using Chirp Scaling. IEEE Transactions on Geoscience and Remote Sensing, 32(4):786-799, July 1994
9. Frigo M., Johnson S. G.: FFTW: An Adaptive Software Architecture for the FFT. ICASSP conference proceedings 1998 vol. 3, pp. 1381-1384