VSIPL++: Parallel Performance

Mark Mitchell
CodeSourcery, LLC
mark@codesourcery.com

Jeffrey D. Oldham
CodeSourcery, LLC
oldham@codesourcery.com

May 27, 2004

1 Introduction

VSIPL++[1] is the object-oriented "next-generation" version of the Vector Signal and Image Processing Library (VSIPL).[2] Like VISPL, VSIPL++ specifies an Application Programming Interface (API) for use in the development of high-performance numerical applications, with a particular focus on embedded real-time systems performing signal processing and image processing. VSIPL++ contains a number of improvements relative to VSIPL including a simpler, more intuitive programming model, simpler syntax, and greater flexibility and extensibility. The most significant of VSIPL++'s improvements is its support for multi-processor computation. This parallel support requires only that the user specify the way in which data should be distributed across processors. The VSIPL++ library automatically manages the transmission of data between the processors as necessary to effectively perform the desired computations.

CodeSourcery was awarded funding under the Air Force Small Business Investment Research (SBIR) program to develop a prototype version of the parallel functionality described in the VSIPL++ specification and to obtain measurements of VSIPL++ performance when executing on parallel systems.[3] Our prototype implementation achieves a near-linear speedup on multi-processor systems demonstrating that, despite the high level of abstraction present in VSIPL++, it is nevertheless possible to obtain excellent performance. Thus, VSIPL++ has the potential to allow programmers to easily and rapidly develop systems that are both highly portable and highly efficient. In our presentation, we will describe the parallel VSIPL++ programming model, our parallel performance benchmark, and the results we obtained.

2 Benchmark Description

Beamforming is the detection of energy propagating in a particular direction while rejecting energy propagating in other directions. A beamformer consists of an array of sensors capturing signals and a signal processing algorithm to extract signals from one or more particular directions and one or more particular frequencies. The k-Ω beamformer we consider assumes uniform spacing of omnidirectional individual sensors along the x-axis. No assumptions about the signal's structure are made except that the signal is periodic and that the signal source is sufficiently far away that the signal appears planar to the sensors, and noise is assumed to be uniformly distributed across the signal.

The beamformer computes the power of the incoming signal for various bearings (k) and frequencies (Ω). Those k-Ω pairs where the power is strongest indicate incoming signals. Each sensor samples the input signal over time. After enough samples have been obtained, a computation is performed (involving the inputs from all of the sensors) to determine the power for the k-Ω pairs. First, FIR filters remove higher-order frequencies from the signal matrix. Then a real-to-complex FFT is applied to the rows of the matrix, optionally the data is reordered into a column-major matrix, and finally a complex-to-complex FFT is applied to the columns. Generally, the collection of data and determining of power is repeated multiple times. The final power reported for a given k-Ω pair is the average of that computed for the various iterations of the process.

[1] `http://www.hpec-si.org/private/vsipl++specification.html`
[2] `http://www.vsipl.org`
[3] SBIR Contract FA87450-04-C-0017

3 Implementation

We implemented the beamformer using several different programming methodologies. One implementation was written using C and VSIPL. After that implementation was complete, we developed a C++ and VSIPL++ implementation. The VSIPL and VSIPL++ implementations are similar in structure, but the VSIPL++ implementation is shorter than the VSIPL implementation because of the higher levels of abstraction provided by VSIPL++. Each implementation runs the k-Ω beamformer multiple times and computes a "running average" power spectra.

We modified the VSIPL++ reference implementation, developed by CodeSourcery under contract from MIT Lincoln Laboratory, to contain support for a subset of the functionality being considered for the parallel VSIPL++ specification. In particular, we created a data storage abstraction called `DistributedBlock` to represent a single one- or two-dimensional array that is stored across multiple processors. We specialized VSIPL++ algorithms, e.g., computing FIR filters and FFTs, to perform only local computations when operating on a `DistributedBlock`. We also modified VSIPL++ to implement a specialization of the two-dimensional FFT algorithm so that, when the input is a row-distributed `DistributedBlock` and the output is a column-distributed block, the algorithm performs the "corner-turn" required.

The VSIPL++ specification is written so as to be independent of any particular message-passing or threading system. However, in our implementation we chose to use the popular Message Passing Interface (MPI)[4] to transmit data between cooperating processors.

After making these modifications to the VSIPL++ implementation, we made minor changes to our VSIPL++ benchmark program. These changes consisted only of modifications to the types used to declare particular arrays in the benchmark program. For example, some arrays were modified to use `DistributedBlock` to indicate distribution across processors. The types of these arrays indicate the arrays are distributed by rows or columns.

4 Results

The following table demonstrates that we were able to obtain a near-linear speedup with parallel VSIPL++ relative to serial VSIPL++ and VSIPL. Times are shown for the VSIPL implementation of the benchmark, the serial VSIPL++ implementation, and the parallel VISPL++ implementation with one and two processors. Times for two hundred iterations of one problem instance are presented. Times for other instances are similar and will be presented in the extended version of this paper. In all cases, the times were obtained by running on a dual-processor Intel Pentium 4 Xeon GNU/Linux machine. The times given reflect only time spent in the execution of the beamforming computations. They do not include time required for initialization and finalization of the application and its libraries. All times shown are "wall-clock time," i.e., the total number of seconds required to execute the beamformer including time spent in the operating system kernel.

	serial, no corner turn		distributed VSIPL++	
	VSIPL	VSIPL++	1-processor	2-processor
FIR filter	20.7	20.7 + 0.1	20.7 + 0.3	20.7/2 + 0.2
1st FFT	12.7	12.7 + 0.1	12.7 + 0.0	12.7/2 + 0.3
Corner Turn	—	—	6.2 + 2.9	6.0 + 8.9
2nd FFT	10.0 + 9.2	10.0 + 9.1	10.0	10.0/2 + 0.1

The corner-turn times indicate the seconds required to transpose a row-major matrix to a column-major matrix plus the time for an MPI All-to-All computation. With one processor, only the transpose occurs. With two processors, MPI communication also occurs. The serial implementations do not perform the transpositions, leading to an increase in the time for the second FFT.

[4]`http://www-unix.mcs.anl.gov/mpi/`

Abstract Submission Details

1. Title: VSIPL++: Parallel Performance

2. Authors:

   - Mr. Mark Mitchell
     CodeSourcery, LLC
     9978 Granite Point Ct
     Granite Bay, CA 95746
     +1.916.791.8304 (voice)
     +1.916.914.2066 (fax)
     `mark@codesourcery.com`
     USA citizenship

   - Dr. Jeffrey D. Oldham
     CodeSourcery, LLC
     144 Wyandotte Dr
     San Jose, CA 95123-3727
     +1.408.578.5684 (voice)
     +1.408.578.5684 (fax)
     `oldham@codesourcery.com`
     USA citizenship

3. First Author: Mitchell
   Corresponding Author: Oldham
   Presenting Author: Oldham

4. Submit for any session.

5. Prefer talk, not poster.

6. Work areas:

   - Middleware Libraries and Application Programming Interfaces
   - Software Architectures, Reusability, Scalability, and Standards