Advanced Radar Signal Processing on General-Purpose Commercial Multiprocessor Systems

Thomas F. Steck and Gerard G.L. Meyer
JHU Parallel Computing and Imaging Laboratory

Peter G. Vouras and Vilhelm Gregers-Hansen
NRL Radar Division

Abstract - Presentation

Real-time signal processing applications have found widespread use in areas such as radar processing and communications, to name a few. In military systems, the added demand of compact, rugged hardware has resulted in these applications being developed almost exclusively on special-purpose hardware. At the same time, general-purpose high performance computers, such as the HP Exemplar, SGI Origin2000, and the IBM SP3 systems are able to meet the real-time requirements while minimizing hardware and software development costs. The Johns Hopkins University Parallel Computing and Imaging Laboratory in concert with the Naval Research Laboratory Radar Division has implemented an experimental advanced radar signal processor for use in upgrades to the Navy's Aegis radars that meets stringent real-time requirements. High performance portability is made possible through parameterization of the algorithms to allow tuning to a particular architecture. Parameters to control the number of processors utilized, the floating-point precision, the FFT length, and the communication method are introduced. Two communication methods are implemented and contrasted in terms of performance and portability. The first approach uses the Message Passing Interface (MPI) for all interprocessor communication. The second is a hybrid approach using MPI and shared memory for communication. The hybrid approach attains greater than 10% improvement in execution time over the pure MPI approach. In addition, systems such as the IBM are not capable of executing MPI on all of the processors in a node for a single job. The penalty for using a hybrid approach is reduced portability due to non-standard shared-memory directives between platforms. In this paper, we discuss our implementation and compare the approaches on the HP SPP2000 Exemplar.

The core radar signal processing problem is shown in Figure 1. Data arrives on four channels from the radar: the main beam or Sum channel, the Sidelobe Blanking (SLB) channel, and two monopulse difference channels, á and â. Two approaches to EMI mitigation are taken for the Sum and SLB channels, and as a result, there are effectively six channels at its output. The pulse compression task is the most computationally expensive task as it involves convolving complex reference signals with the data signals. The noncoherent integration builds up the output from multiple radar pulses in an effort to improve signal to noise. To give an idea of the processing requirements, a single pulse takes 27 ms to process on a single HP PA8500 processor. The system must be able to keep pace with incoming pulses using minimal buffering. This cannot be done without the use of multiple processors.

The basic approach to implementing the problem on a parallel architecture is assisted by the relatively small amount of data that must be shared between channels. Both Sum and SLB data are needed for EMI detection. A list of EMI detections must then be shared across all channels in order to perform EMI mitigation. During noncoherent integration, Sum and SLB data must be gathered from all pulses in the waveform. During post-processing (not shown above), data from all channels and all pulses must be combined and communication requirements become significantly more demanding. Although none of the processors evaluated are fast enough to process an entire pulse in real time, one processor per channel is not absolutely necessary for most of the systems evaluated. As a result, the number of processors is left as a parameter in our implementation. This greatly enhances portability as can be seen by the fact that the multiprocessor code currently runs on the HP Exemplar SPP2000 and V2500, the SGI Origin2000, the IBM SP3, and Intel multiprocessor platforms running either Windows NT or Linux. In addition, as new functionality is needed, additional processors can be assigned as necessary; or on the other hand, if a faster processor is available, the required number of processors can easily be reduced to reduce cost, physical volume, and power consumption.

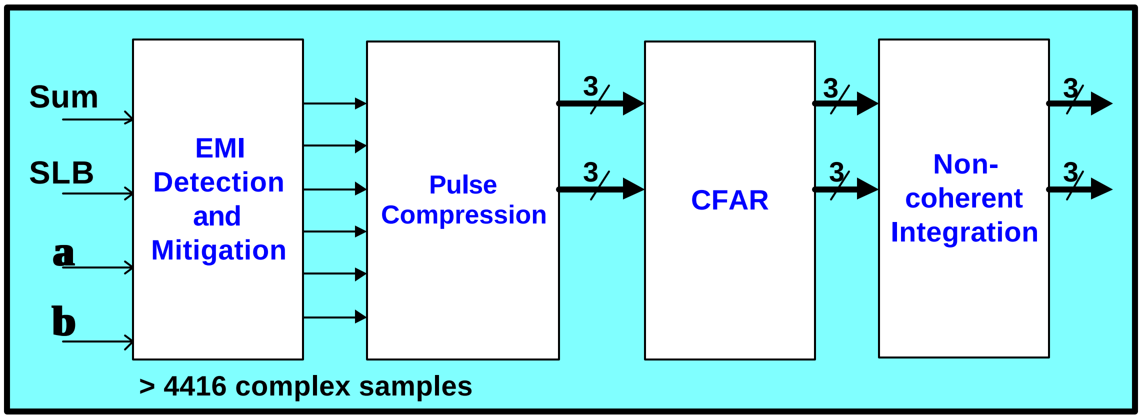

Figure 1. Core radar signal processing problem.

Another parameter that is introduced is FFT length. Pulse compression requires convolution of two complex signals. A significant amount of research and experimentation has been done to show that using modern processors, frequency domain computation of a convolution using FFTs is more computationally efficient and results in faster execution. However, the choice of FFT size is not obvious. Using an overlap-and-save convolution with the FFT size smaller than the data length can improve performance, especially when the FFT size is chosen to be a power of 2. Depending on the vectors to be convolved and the system architecture, the optimal FFT length varies. We show that for the vectors under consideration, an FFT of length 2048 is optimal on the IBM and HP systems.

The next parameter introduced is the floating point precision. Traditionally, fixed point arithmetic is used in embedded architectures for signal processing. With general-purpose processors, floating point performance is good enough to be competitive with fixed point solutions while affording greater flexibility. While the 12-bit data received from the radar A/D converters is less than 16-bit IEEE single precision floating point, the internal computations nevertheless may benefit from double precision floating point. On older 32-bit processors, the performance penalty is usually too great to allow double precision to be used. However, on newer 64-bit processors such as MIPS R12000, there is no performance degradation when using double precision. Therefore, there is no cost for the additional precision.

The communication parameter allows us to choose varying combinations of MPI and shared memory to set up communication between processors. We start with an explanation of the pure MPI implementation. The hybrid implementation uses shared memory to communicate information from the different channels within a waveform pulse. The use of shared memory eases programming without sacrificing performance when looking at fine grain parallelism while the MPI allows the use of multiple computational nodes for coarse grain parallelism within the problem. Next, we discuss in more detail each of these implementations.

The MPI implementation dynamically generates a map of pulse and channel to each processor in the available pool. Each pulse is assigned a processor group. Each channel is then assigned to a processor within the groups. Processor group and channel assignment is done in a striping fashion if the number of processors is less than the number of pulses or channels. A central housekeeper processor is assigned the task of receiving incoming pulses and distributing them to the appropriate processors. The housekeeper also gathers results needed for reporting and returns them to the radar control computer. Persistent communicators are set up to assist in distributing data to processor groups. The processor map is generated on each processor so that inter-channel and inter-pulse communication can be set up locally. This model works well for the core processing. The post-processing requires sections of the data stream from each channel to be pooled. The communication setup is complicated by the location and size of data that can only be determined once core processing is completed on all waveform pulses. There are critical tradeoffs between trying to send as much data as possible and trying to send only the bare minimum data necessary to complete post-processing.

A hybrid implementation that uses MPI and shared memory allows the programmer to take advantage of special hardware that dramatically improves the performance of the communication network when using shared memory. Architectures that use several nodes with anywhere from 8 to 16 processors per node are typically designed to execute shared memory code within the node optimally. Node-to-node communication is either degraded or impossible using the same memory constructs, necessitating either pure MPI or hybrid implementations. The IBM SP3, for example, has 8 processors per node and performance benchmarks have shown that shared memory outperforms equivalent MPI implementations. Unfortunately, there is no concept of shared memory between nodes on this system. Our hybrid implementation sets up shared memory processing groups for each pulse and distributes data to the processing groups using MPI. Post-processing is done on a separate, dedicated shared memory group of processors. As in the MPI case, a housekeeper is used to distribute data and collect post-processing data for reporting.

The pure MPI and the hybrid implementations presented meet the real-time processing requirements. The hybrid solution obtains better performance than the pure MPI solution, but sacrifices some portability. While efforts have been made to adopt a common language for shared memory coding, vendor-specific directives are still needed to obtain optimum performance on the machines we used. In addition, there are still no shared memory implementations for the Intel platforms.