Chapter 15

Performance Metrics and Software Architecture

Jeremy Kepner, Theresa Meuse, and Glenn Schrader
MIT Lincoln Laboratory

15.1 Introduction

High performance embedded computing (HPEC) systems are amongst the most challenging systems in the world to build. The primary sources of these difficulties are the large number of constraints on an HPEC implementation:

Performance: latency and throughput

Efficiency: processing, bandwidth, and memory

Form Factor: size, weight, and power

Software Cost: code size and portability

This chapter will primarily focus on the performance metrics and software architectures for implementing HPEC systems that minimize software cost while meeting as many of the other requirements as possible. In particular, we will focus on the various software architectures that can be used to exploit parallel computing to achieve high performance. In this context, the dominating factors in the HPEC software architecture are

Type of parallelism: data, task, pipeline, and/or round-robin

Parallel programming model: message passing, threaded and/or global arrays

Programming environment: languages and/or libraries

The approaches for dealing with these issues are best illustrated in the context of a concrete example.

Section 15.2 gives an overview of a canonical HPEC application (synthetic aperture radar or SAR) taken from the HPEC Challenge benchmark suite (http://www.ll.mit.edu/hpecchallenge). [Note: for a more detailed description see Appendix A.] The rest of the chapter is organized as follows. Section 15.3 will describe the different types of parallelism that can be applied to the application and provide a mathematical model for exploring the performance tradeoffs. Section 15.4 will provide a quick definition of a typical programmable multi-computer that we will attempt to build the application on. Section 15.5 discusses the software impacts of different software implementations. Finally, Section 15.6 will define the key system performance, efficiency, form factor and software cost metrics that we will use for assessing the implementation.

15.2 Synthetic Aperture Radar Example Application

SAR is one of the most common modes in a radar system and one of the most computationally stressing to implement. The goal of a SAR system is usually to create images of the ground from a moving airborne radar platform. The basic physics of a SAR system (Soumekh 1999) begins with the radar sending out pulses of radio waves aimed at a region on the ground that is usually perpendicular to the direction of motion of the platform (see Figure 15.1). The pulses are reflected off the ground and detected by the radar. Typically, the area of the ground that reflects a single pulse is quite large and an image made from this raw unprocessed data is very blurry (see Figure 15.2). The key concept of a SAR system is that it moves between each pulse, allowing multiple looks at the same area of the ground from *different* viewing angles. Combining these different viewing angles together produces a much sharper image (see Figure 15.2). The resulting image is as sharp as one taken from a much larger radar with a "synthetic" aperture the length of the distance travelled by the platform.

There are many variations on the mathematical algorithms used to transform the raw SAR data into a sharpened image. In this chapter, we will focus on the variation referred to as "spotlight" SAR. Furthermore, we will look at a simplified version of this algorithm that focuses on the most computationally intensive steps of SAR processing that are common to nearly all SAR algorithms. Our example is taken from the "Sensor Processing and IO Benchmark" from the HPEC Challenge benchmark suite (see http://www.ll.mit.edu/hpecchallenge).

The overall block diagram for this benchmark is shown in Figure 15.3. At the highest level it consists of three stages:

SDG: Scalable Data Generator. Creates raw SAR inputs and writes them to files to be read in by Stage 1.

Stage 1: Front-End Sensor Processing. Reads in raw SAR inputs, turns them into SAR images, and writes them out to files.

Stage 2: Back-End Knowledge Formation. Reads in pairs of SAR images, compares them and then detects and identifies the difference.

Although the details of the above processing stages vary dramatically from radar to radar, the core computational details are very similar: input from a sensor, followed by processing to form an image, followed by additional processing to find objects of interest in the image.

15.2.1 Operating Modes

This particular SAR benchmark has two operating modes (Compute Only and System). The "Compute Only Mode" represents the processing performed directly from a continuously streaming sensor (Figure 15.4). In this mode, the SDG is meant to simulate a sensor data buffer that is filled with a new frame of data at regular intervals, T_{input}. In addition, the SAR image created in Stage 1 is sent directly to Stage 2. In this mode, the primary architectural challenge is providing enough computing power and network bandwidth to keep up with the input data rate.

In "System Mode" the SDG represents an archival storage system that is queried for raw SAR data (Figure 15.3). Likewise, Stage 1 stores the SAR images back to this archival system and Stage 2 retrieves pairs of images from this storage system. Thus, in addition to the processing and bandwidth challenges, the performance of the storage system must also be managed. Increasingly, such storage systems are the key bottleneck in sensor processing systems. Currently, the modeling and understanding of parallel storage systems is highly dependent on the details of the hardware. To support the analysis of such hardware, the SAR benchmark has an "IO Only Mode" that allows for benchmarking and profiling. The theoretical modeling and analysis of parallel file systems is beyond the scope of this chapter, which will primarily focus on the "Compute Only Mode."

15.2.2 Computational Workload

The precise algorithmic details of this particular SAR processing chain are given in Appendix A. For the purposes of mapping the algorithm onto a parallel computer, the only relevant pieces are listed below:

Core Data Structures: The array(s) at each stage that will consume the largest amount of memory and upon which most of the computations will be performed.

Computational Complexity: The total operations performed in the stage and how they depend upon the algorithmic parameters.

Degrees of Parallelism: The parallelism inherent in the stage and how it relates to the core data structures.

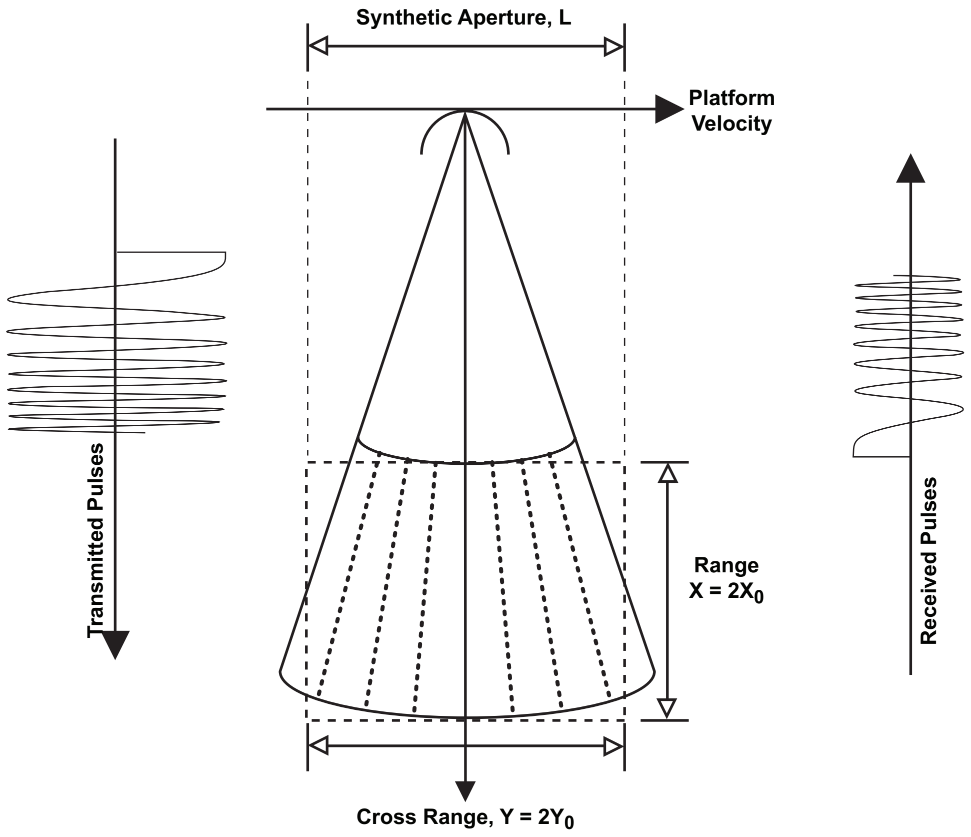


Figure 15.1: **Basic geometry of a SAR system.** A SAR system sends out pulses of radio waves aimed at a region on the ground that is usually perpendicular to the direction of motion of the platform. The pulses are reflected off the ground and detected by the radar. The direction parallel to the transmission of the pulses is referred to as the “range” or “down range” direction. The direction perpendicular to the transmission of the pulses is referred to as the “cross range” direction.

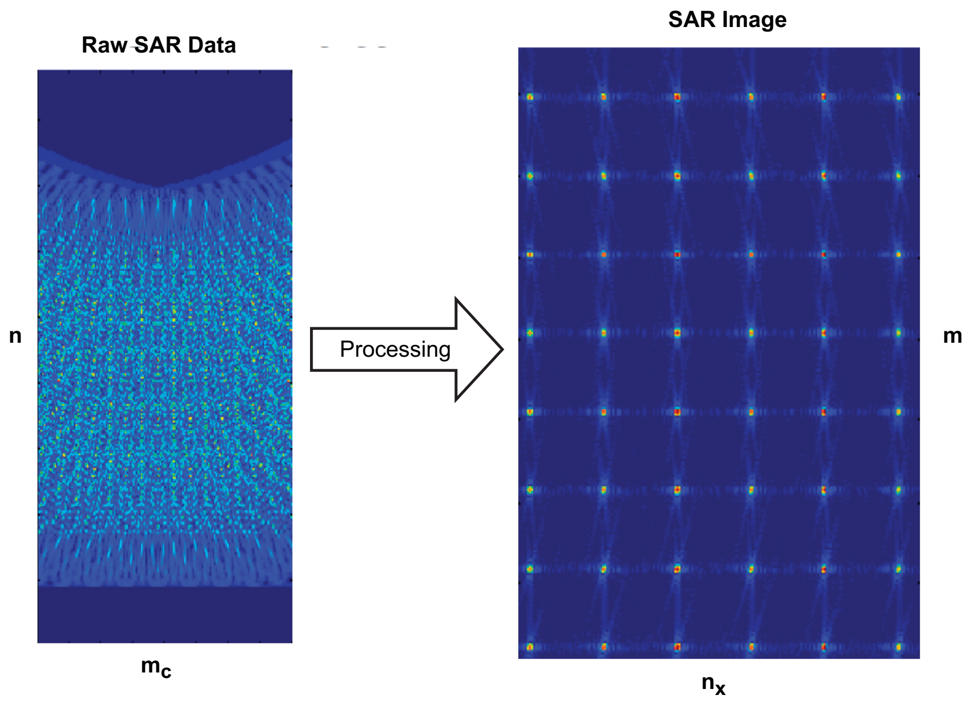


Figure 15.2: **Unprocessed (left) and processed (right) SAR data.** The area that reflects a single pulse is large and an image of this raw data is very blurry (left). A SAR system provides multiple looks at the same area of the ground from multiple viewing angles. Combining these different viewing angles together produces a much sharper image (right).

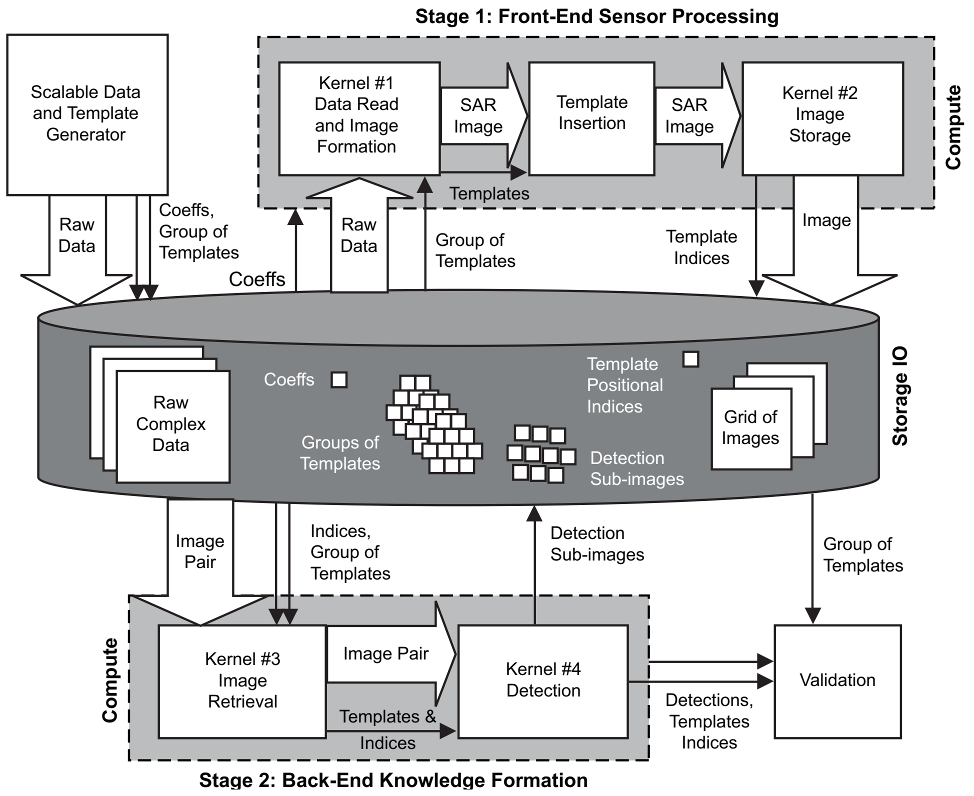


Figure 15.3: **System Mode Block Diagram.** SAR System Mode consists of Stage 1 Front-End Sensor Processing and Stage 2 Back-End Knowledge Formation. In addition, there is significant IO to the storage system.

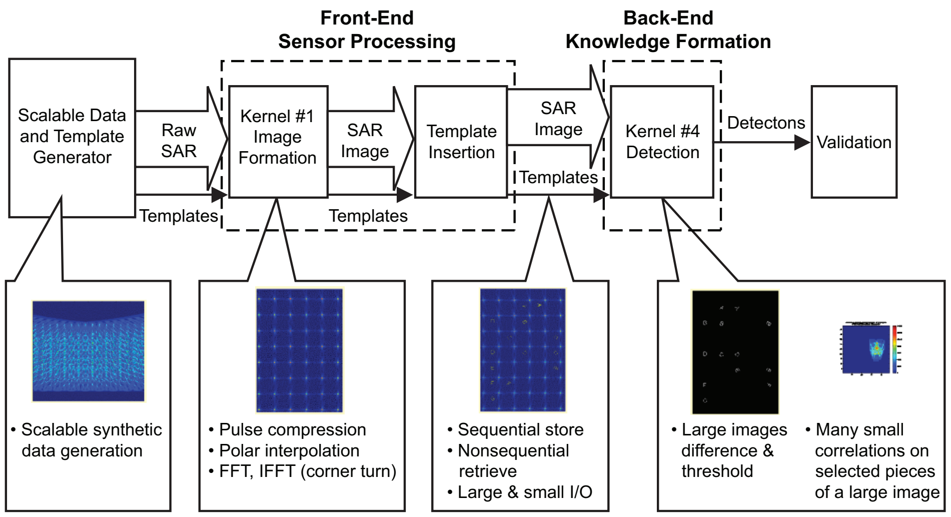


Figure 15.4: **Compute Only Mode Block Diagram.** Simulates a streaming sensor that moves data directly from front-end processing to back-end processing.

Data IO: The amount of data that needs to be moved into and out of the computational stage.

These pieces of the HPEC Challenge SAR benchmark are summarized in Figure 15.5 and Table 15.1. In Stage 1, the data is transformed in a series of steps from a $n \times m_c$ single-precision complex valued array to a $m \times n_x$ single-precision real valued array. The "degrees of parallelism" described at each stage refers to the amount of parallelism within each stage. This is sometimes referred to as "fine grain" parallelism. There is also pipeline or task parallelism, which exploits the fact that each step in the pipeline can be performed in parallel, with each step processing a frame of data. Finally, there is also coarse grain parallelism, which happens because separate SAR images can be processed independently. This is equivalent to setting up multiple pipelines.

In Stage 1, the processing is along either the rows or the columns, which defines how much parallelism can be exploited. In addition, when the direction of parallelism switches from rows to columns or columns to rows, then this means that a "corner turn" of the matrix must be performed. On a typical parallel computer a corner turn requires every processor to talk to every other processor. These corner turns often are natural boundaries along which to create different stages in a parallel pipeline. For example, it makes sense to combine Stages 1c and 1d (now referred to as 1cd) since they do not involve changing the direction of parallelism (see Table 15.1). Thus, in Stage 1 there are four steps (1a, 1b, 1cd and 1e), which require three corner turns. This is typical of most SAR systems.

In Stage 2, pairs of images are compared to find the locations of new "targets" (denoted by n_{target}). In the case of the SAR benchmark, these targets are just $n_{font} \times n_{font}$ images of rotated capital letters that have been randomly inserted into the SAR image. The region of interest (ROI) around each target is then correlated with each possible letter and rotation to determine its identity, its rotation, and its location in the SAR image. The parallelism in this stage is determined by n_{target} and can be along the rows or columns or both, as long as enough overlapping edge data is kept on each processor to do the correlations. These edge pixels are sometimes referred to as "overlap," or "boundary," or "halo," or "guard" cells.

The input bandwidth is a key parameter in describing the overall performance requirements of the system. The input bandwidth BW^i_{input} (in samples/second) for each processing stage i is given by

$$\begin{aligned} BW^1_{input} &= n m_c / T_{input} \ , \\ BW^2_{input} &= n_x m / T_{input} \ . \end{aligned} \tag{15.1}$$

A simple approach for estimating the overall required processing rate is to multiply the input bandwidth by the number of operations per sample required. Looking at Table 15.1, if we assume

$$n \approx n_x \approx 8000 \ \ , \ \ m_c \approx m \approx 4000 \ , \ \ n_{target} \approx \frac{n_x m}{8 n^2_{font}} \ ,$$

then the operations (or work) done per data input sample W^i_{sample} can be approximated by

$$\begin{aligned} W^1_{sample} &\approx 10 \lg(n) + 20 \lg(m_c) + 40 \approx 400 \ , \\ W^2_{sample} &\approx \frac{1}{8} n_{let} n_{rot} n^2_{font} \approx 1000 \ . \end{aligned} \tag{15.2}$$

Thus the rate of performance goal R^i_{goal} is approximately

$$\begin{aligned} R^1_{goal} &\approx W^1_{sample} BW^1_{input} \approx 25 \times 10^9 / T_{input} \ , \\ R^2_{goal} &\approx W^2_{sample} BW^2_{input} \approx 16 \times 10^9 / T_{input} \ . \end{aligned} \tag{15.3}$$

T_{input} varies from system to system, but can easily be much less than a second, resulting in large compute performance goals. Satisfying these performance goals often requires a parallel computing system.

The file IO requirements in "System Mode" or "IO Only Mode" are just as challenging as the computation. In this case the goal is to read and write the files as quickly as possible. During Stage 1, a file system must read large input files and write large image files. Simultaneously, during Stage 2, the image files are selected at random and read. After Stage 2 detection is performed, many very small "thumbnail" images around the targets are written. This diversity of file sizes and the need for simultaneous read and write is very stressing and often requires a parallel file system.

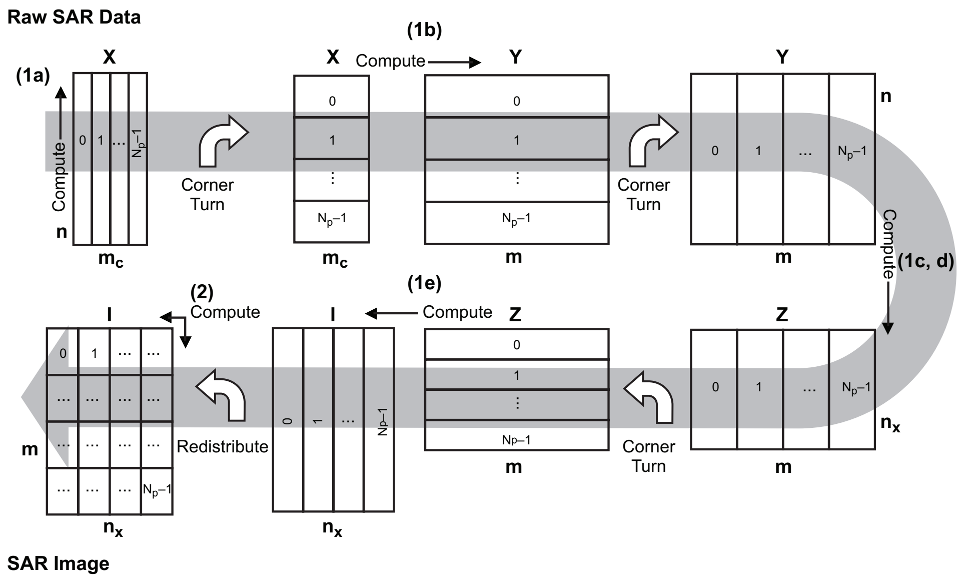


Figure 15.5: **Algorithm Flow.** In Stage 1, the data is transformed in series of steps from a $n \times m_c$ single-precision complex valued array to a $n_x \times m$ single-precision real valued array. At each step the processing is along either the rows or the columns.

Table 15.1: **Algorithm Complexity and Parallelism.** Computational complexity, degrees of parallelism, IO and operations per sample for each stage of the SAR algorithm.

Stage	Data Array	Compute Complexity	Degrees of Parallelism	Input Samples	Output Samples	Operations/ Sample
SDG	$X : C_{32bit}^{n\times m_c}$	NA	NA	NA	nm_c	NA
1a	$X : C_{32bit}^{n\times m_c}$	$5nm_c \lg(n)$	m_c	nm_c	nm_c	$5\lg(n)$
1b	$Y : C_{32bit}^{n\times m}$	$5nm_c \lg(m_c)$ $+10nm\lg(m)$	n	nm_c	nm	$5\lg(m_c)$ $+10\lg(m)$
1c	$Z : C_{32bit}^{n_x\times m}$	$40nmn_K$	m	nm	n_xm	$40n_K$
1d	$Z : C_{32bit}^{n_x\times m}$	$5n_xm\lg(m)$	m	n_xm	n_xm	$5\lg(m)$
1e	$I : R_{32bit}^{m\times n_x}$	$5n_xm\lg(n_x)$	n_x	n_xm	n_xm	$5\lg(n_x)$
2	$I : R_{32bit}^{m\times n_x}$ $T : R_{32bit}^{n_{let}\times n_{rot}\times n_{font}^2}$	$\frac{1}{8}n_{let}n_{rot}n_{font}^4 n_{target}$	n_{target}	n_xm	$n_{font}^2 n_{target}$	$\frac{n_{let}n_{rot}n_{font}^4}{8n_{target}}$

Processor Sizing. Often the first step in the development of a system is to produce a rough estimate of how many processors will be needed. This step often occurs during development, perhaps early on in the system design phase. Frequently processor sizing estimates are used to determine the type of processing technology to use (programmable or non-programmable) and approximately how much a solution will cost to implement. A basic processor sizing estimate consists of

Computational Complexity Analysis. Estimates the total amount of work that needs to be done by the application (e.g., see Equation 15.2).

Throughput Analysis. Converts the total work done into a rate at which this work must be performed (e.g., see Equation 15.3).

Processor Performance. Estimates of what the performance of the algorithm will be on a single processor.

After the above analysis has been completed, the nominal processor size (in terms of number of processors) is estimated by dividing the required throughput by the processing rate

$$N_P^{est} \approx \sum_i \frac{R^i_{goal}}{\epsilon^i_{comp} R_{peak}} = \frac{R^1_{goal}}{\epsilon^1_{comp} R_{peak}} + \frac{R^2_{goal}}{\epsilon^2_{comp} R_{peak}} \ , \tag{15.4}$$

where ϵ^i_{comp} is the efficiency at which a processor with peak performance R_{peak} (in operations/second) can perform the work in stage i. Note: this procedure only gives a rough approximation and should always be used as such.

15.3 Degrees of Parallelism

The parallel opportunities at each stage of the calculation discussed in the previous section show that there are many different ways to exploit parallelism in this application. These different types of parallelism can be roughly grouped into three categories (see Figure 15.6):

Coarse Grained. This is the highest level of parallelism and exploits the fact that each raw SAR input can be processed independently of the others. This form of parallelism usually requires the least amount of communication between processors, but requires the most memory and has the highest latency.

Task/Pipeline. This decomposes different stages of the processing into a pipeline. The output of one stage is fed into the input of another so that at any given time each stage is working on a different SAR dataset.

Data Parallelism. This is the finest grain parallelism and decomposes each SAR image into smaller pieces.

For the specific problem at hand, we will look at exploiting all of these different types of parallelism. However, our analysis will only extend to using these to one level of depth. In other words, the most complex case we will consider is multiple coarse-grained pipelines, where each pipeline has multiple data parallel steps. We will not examine the very common, but even more complex case, where individual steps within a pipeline are further broken up into more coarse grain sub-pipelines with a further number of additional data parallel sub-steps.

We will parameterize parallel implementation as follows:

$N_{coarse} \equiv$ number of different problems or pipelines,

$N_{stage} \equiv$ number of stages in the pipeline,

$N^i_{stage} \equiv$ the number of processors used at stage i in the pipeline.

These parameters must satisfy the constraint that the sum of all the processors used at every stage in every pipeline is equal to the total number of processors N_P

$$N_P = N_{coarse} \sum_{i=1}^{N_{stage}} N^i_{stage} \quad .$$

There are several important special cases of the above description. First is the pure coarse grain parallel case, where each processor is given an entire SAR dataset to process and neither fine grain parallelism nor pipeline parallelism is exploited:

$$N_{coarse} = N_P, \quad N_{stage} = 1, \quad N^1_{stage} = 1 \ .$$

Next is the pure pipeline parallel case. This case is limited in that the total number of processors can't be more than the number of stages in the pipeline:

$$N_{coarse} = 1, \quad N_{stage} = N_P, \quad N^i_{stage} = 1 \ .$$

Finally, there is the pure data parallel case where only fine grain parallelism is exploited:

$$N_{coarse} = 1, \quad N_{stage} = 1, \quad N^1_{stage} = N_P \ .$$

15.3.1 Parallel Performance Metrics (no communication)

In the absence of any communication and IO cost, all of these parallel approaches will take the same time to process a frame of data (where we assume that there are N_{frame} instances)

$$T_{comp}(N_P) = \frac{W^{tot}}{\epsilon_{comp} R_{peak} N_P} \propto \frac{W^{tot}}{N_P} \ ,$$

where W^{tot} is the total computational operations or work required to process a frame of data at each stage (see Table 15.1)

$$W^{tot} \equiv \sum_{i=1}^{N_{stage}} W^i = W^{1a} + W^{1b} + W^{1cd} + W^{1e} + W^2 \ .$$

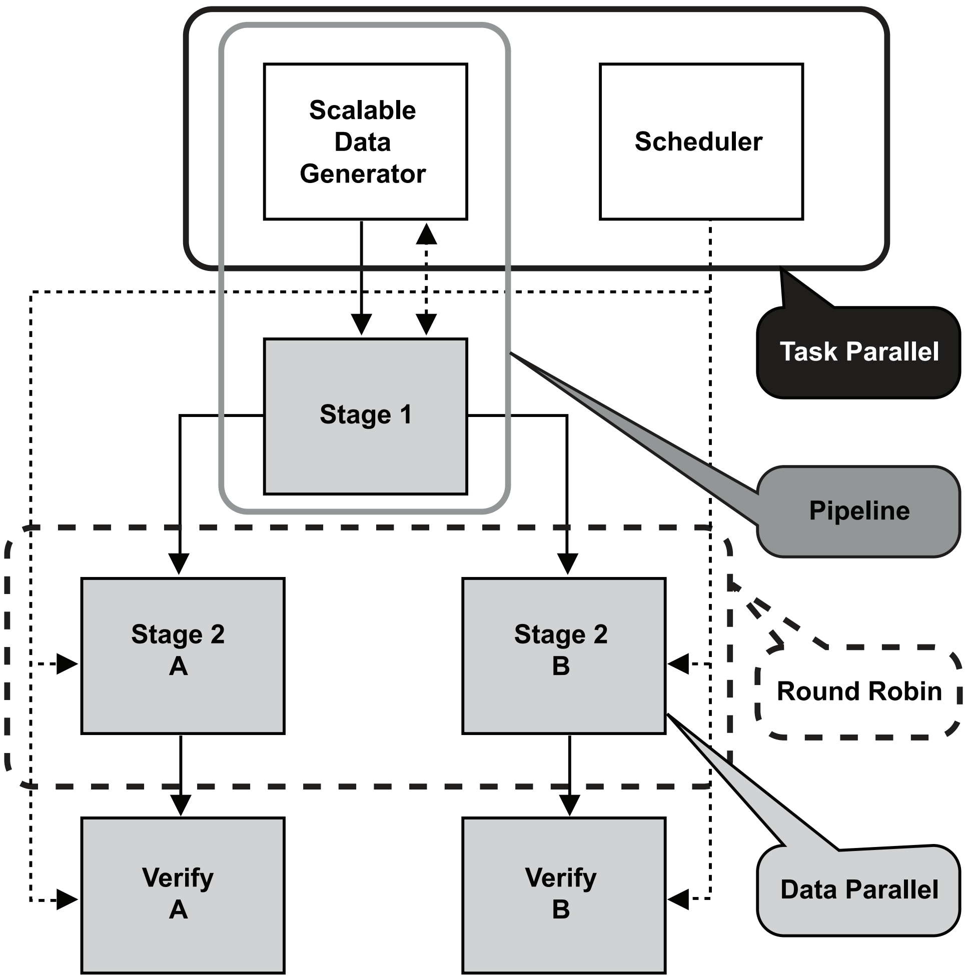


Figure 15.6: **Types of Parallelism.** Within the SAR benchmark there are many different types of parallelism that can be exploited: task (coarse grain), pipeline, round-robin (coarse grain), and data (fine grain).

The computational speedup will be linear in the number of processors:

$$S_{comp}(N_P) \equiv \frac{T_{comp}(1)}{T_{comp}(N_P)} = N_P \ .$$

Likewise, in the zero communication case, the latency (i.e., the time to get the first answer) is the number of stages times the longest time it takes to process any one stage

$$T_{latency}(N_P) = N_{stage} \max_i (T^i_{latency}(N^i_{stage})) \ ,$$

where

$$T_{latency}(N^i_{stage}) = \frac{W^i_{stage}}{\epsilon_{comp} R_{peak} N^i_{stage}} \ .$$

This is because no one stage can progress until the previous stage has also finished. The latency speedup in this case is

$$S_{latency}(N_P) \equiv \frac{T_{latency}(1)}{T_{latency}(N_P)} \ .$$

The total memory required for the different approaches is

$$M(N_P) = N_{coarse} \sum_{i=1}^{N_{stage}} \frac{M^i_{stage}}{N^i_{stage}} \ .$$

Perhaps more importantly is the max amount of memory required by any particular processor

$$M_{cpu-max}(N_P) = \max_i (M^i_{stage}/N^i_{stage}) \ .$$

15.3.2 Parallel Performance Metrics (with communication)

Of course, assuming no communication and storage costs is a bad assumption and essentially the entire goal of designing a parallel implementation is to minimize these costs. Nevertheless, the above models are useful in that they provide upper bounds on performance. In addition, the speedup and memory footprint definitions are unaltered when communication is included.

In the case of fine grain parallelism, communication is required between steps where the directionality of the parallelism changes, which in turn requires that a cornerturn is performed to reorganize the data to exploit the parallelism. The time to communicate data between steps i and $i+1$ is given by

$$T^{i \rightarrow i+1}_{comm}(N^i_{stage} \rightarrow N^{i+1}_{stage}) = \frac{D^{i \rightarrow i+1}}{BW_{eff}(N^i_{stage} \rightarrow N^{i+1}_{stage})} \ ,$$

where $D^{i \rightarrow i+1}$ is the amount of data (in bytes) moved between steps i and $i+1$ and $BW_{eff}(N^i_{stage} \rightarrow N^{i+1}_{stage})$ is the effective bandwidth (in bytes/second) between the processors, which is given by

$$BW_{eff}(N^i_{stage} \rightarrow N^{i+1}_{stage}) = \epsilon^{i \rightarrow i+1}_{comm} BW_{peak}(N^i_{stage} \rightarrow N^{i+1}_{stage}) \ .$$

$BW_{peak}(N^i_{stage} \rightarrow N^{i+1}_{stage})$ is the peak bandwidth from the processors in stage i to the processors in stage $i+1$. A more detailed model of this performance requires a model of the processor interconnect, which is given the next section.

The implication of the communication on various cases is as follows. For the pure pipeline case (assuming that computation and communication can be overlapped), we have

$$\begin{aligned} T_{tot}(N_P) &= \max_i(\max(T^i_{latency}(1), T^{i\rightarrow i+1}_{comm}(1 \rightarrow 1)))/N_{coarse} \ , \\ T_{latency}(N_P) &= 2N_{stage} \max_i(\max(T^i_{latency}(1), T^{i\rightarrow i+1}_{comm}(1 \rightarrow 1))) \ . \end{aligned}$$

If communication and computation cannot be overlapped, then the inner max(,) function is replaced by addition. In this case, we see that the principal benefit of pipeline parallelism is that it is determined by the max (instead of the sum) of the stage times. The price for improved performance is increased latency, which is proportional to the number of computation and communication stages. There is a nominal impact on the memory footprint.

For the pure data parallel case, we have

$$\begin{aligned} T_{tot}(N_P) &= \frac{T_{comp}(N_P) + T_{comm}(N_P)}{N_{coarse}} \ , \\ T_{latency}(N_P) &= T_{comp}(N_P) + T_{comm}(N_P) \ , \end{aligned}$$

where

$$T_{comm}(N_P) = \sum_i T^{i\rightarrow i+1}_{comm}(N_P \rightarrow N_P) \ .$$

In general, in a pure data parallel approach, it is hard to overlap computation and communication. The principal benefit of data parallelism is that the compute time is reduced as long as it is not offset by the required communication. This approach also reduces the latency in a similar fashion. This approach also provides a linear reduction in the memory footprint.

For the combined pipeline parallel case, we have

$$\begin{aligned} T_{tot}(N_P) &= \max_i(\max(T_{comp}(N^i_{stage}), T^{i\rightarrow i+1}_{comm}(N^i_{stage} \rightarrow N^{i+1}_{stage})))/N_{coarse} \ , \\ T_{latency}(N_P) &= 2N_{stage} \max_i(\max(T_{comp}(N^i_{stage}), T^{i\rightarrow i+1}_{comm}(N^i_{stage} \rightarrow N^{i+1}_{stage}))) \ . \end{aligned}$$

This approach provides an overall reduction in computation time, allows for overlapping computation and communication, and a reduction in the memory footprint. As we shall see later, this is often the approach of choice for parallel signal processing systems.

15.3.3 Amdahl's Law

Perhaps the most important concept in designing parallel implementations is managing overhead. Assume the total amount of work that needs to be done can be broken up into a part that can be done in parallel and a part that can only be done in serial (i.e., on one processor):

$$W_{tot} = W_{||} + W_{|} \ .$$

The execution time then scales as follows:

$$T_{comp}(N_P) \propto W_{||}/N_P + W_{|} \ ,$$

which translates into a speedup of

$$S_{comp}(N_P) = \frac{W_{tot}}{W_{||}/N_P + W_{|}} \ .$$

If we normalize with respect to W_{tot}, this translates to

$$S_{comp}(N_P) = \frac{1}{w_{||}/N_P + w_{|}} \ ,$$

where $w_{||} = W_{||}/W_{tot}$ and $w_{|} = W_{|}/W_{tot}$. In the case when N_P is very large, the maximum speedup achievable is

$$S_{max} = S_{comp}(N_P \rightarrow \infty) = w_{|}^{-1} \ .$$

For example, if $w_1 = 0.1$, then the maximum speedup is 10 (see Figure 15.7). This fundamental result is referred to as Amdahl's Law and the value $w_{|}$ is often referred to as the "Amdahl fraction" of the application. Amdahl's Law highlights the need to make every aspect of a code parallel. It also applies to other overheads (e.g., communication) that cause no useful work to be done. Finally, Amdahl's Law is also a useful tool for making trade-offs. Half the maximum speedup is achieved at $N_P = w_{|}^{-1}$

$$S_{comp}(N_P = w_{|}^{-1}) = w_{|}^{-1}/(w_{||} + 1) \approx \frac{1}{2} S_{max} \ .$$

If $w_{|} = 0.1$, then using more than 10 processors will be of marginal utility. Likewise, if an application needs a speedup of 100 to meet its performance goals, then $w_{|} < 0.01$.

15.4 Standard Programmable Multi-Computer

Embedded computers are tightly coupled to their applications and are physically connected to their input sensors. A canonical embedded system is shown in Figure 15.8. It consists of control processor, interface, and network, and a signal processor, interface and network. Typically the control processor, interface, and network are selected for programmability, reliability, and flexibility. In contrast, the signal processor, interface, and network are often driven by performance considerations. If just one processor cannot process the input data

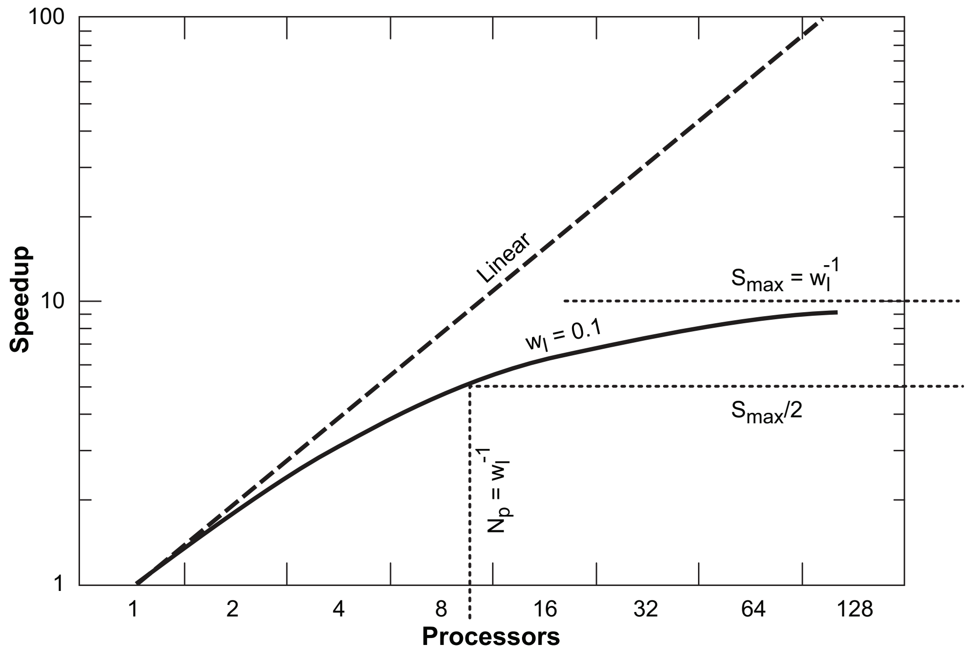


Figure 15.7: **Amdahl's Law.** Plot showing speedup of a program with an Amdahl fraction of $w_| = 0.1$. Dotted lines show the max speedup $S_{max} = w_|^{-1} = 10$ and the half speedup point $N_P = w_|^{-1}$.

stream, then the signal processor must incorporate multiple processors. Sometimes this is referred to as an embedded multi-computer. If a programmable signal processor is selected (as opposed to one implemented in hardware), then parallel computing is the only approach for increasing performance to meet the timing requirements.

The canonical parallel computer architecture consists of a number of nodes connected by a network. Each node consists of a processor and memory. The memory on the node may be further decomposed into various levels of cache, main memory, and disk storage (see Figure 15.9). There are many variations on this theme. A node may have multiple processors sharing various levels of cache, memory, and disk storage. In addition, there may be external global memory or disk storage that is visible to all nodes. A important implication of this canonical architecture is the "memory hierarchy" (Figure 15.9). The memory hierarchy concept refers to the fact that the memory "closer" to the processor is much faster (the bandwidth is higher and latency is lower). However, the price for this performance is capacity. The canonical ranking of the hierarchy is typically as follows:

- Processor registers
- Cache L1, L2, ...
- Local main memory
- Remote main memory
- Local disk
- Remote disk

Typically, each level in the hierarchy exchanges a 10x performance increase for a 10x reduction in capacity. The "steepness" of the memory hierarchy refers to the magnitude of the performance reduction as one descends down the memory hierarchy. If these performance reductions are large, we say the system has a very steep memory hierarchy.

One benchmark specifically designed to probe the memory hierarchy is the HPC Challenge Benchmark suite (Luszczek et al. 2006). HPC Challenge benchmarks have been chosen to cover a range of memory access patterns and stress different parts of the memory hierarchy. Top500 performance is mostly dominated by local matrix multiply operations. STREAM requires no communication, is dominated by local vector operations, and stresses local processor to memory bandwidth. The fast Fourier transform (FFT) is also dominated by all-to-all communications, but for very large messages. RandomAccess is dominated by all-to-all communications of very small messages.

The concept of the memory hierarchy is probably the most important idea to keep in mind when developing a high performance implementation. More specifically, a high performance implementation of a sensor processing system is one that best mitigates the performance impacts of the memory hierarchy.

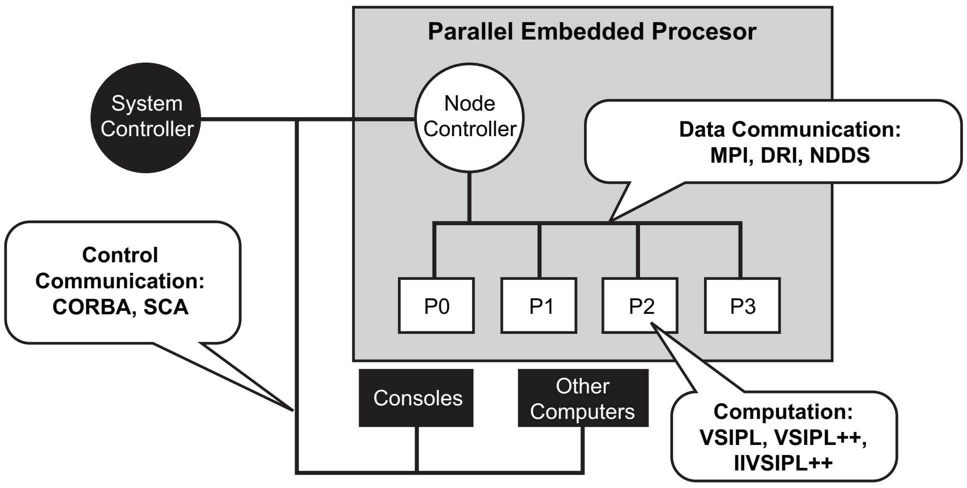

Parallel Embedded Procesor
System Controller
Node Controller
Data Communication: MPI, DRI, NDDS
Control Communication: CORBA, SCA
P0
P1
P2
P3
Consoles
Other Computers
Computation: VSIPL, VSIPL++, IIVSIPL++

Figure 15.8: **Canonical Embedded System Architecture.**

This requires programmers to have a clear picture of the system in their minds so that they can understand the precise performance implications. The basic rule of thumb is to construct an implementation that minimizes both the total volume and number of data movements up and down the hierarchy.

15.4.1 Network Model

The communication network that connects the processors in a parallel system is often the most critical piece in designing a parallel implementation of an application. The starting point of modeling a communication network is point-to-point performance. This can be effectively measured by timing how long it takes two processors to send messages of different sizes to each other. The result is a standard curve (15.10), which can mostly be characterized by a single function $T(m)$, which is the time it takes to send a message of size m. The function $T_{comm}(m)$ is typically well described by the following simple two-parameter formula

$$T_{comm}(m) = \text{Latency} + \frac{m}{\text{Bandwidth}} \quad . \tag{15.5}$$

Bandwidth, which is typically measured in bits or bytes per second, is the maximum rate at which data can flow over a network. Bandwidth is typically measured by timing how long it takes to send a large message between two processors and dividing the message size by the total time

$$\text{Bandwidth} = \lim_{m\to\infty} \frac{T(m)}{m} \quad . \tag{15.6}$$

Latency is how long it takes for a single bit to be sent from one processor to the next. It is usually measured by timing a very short message and is typically

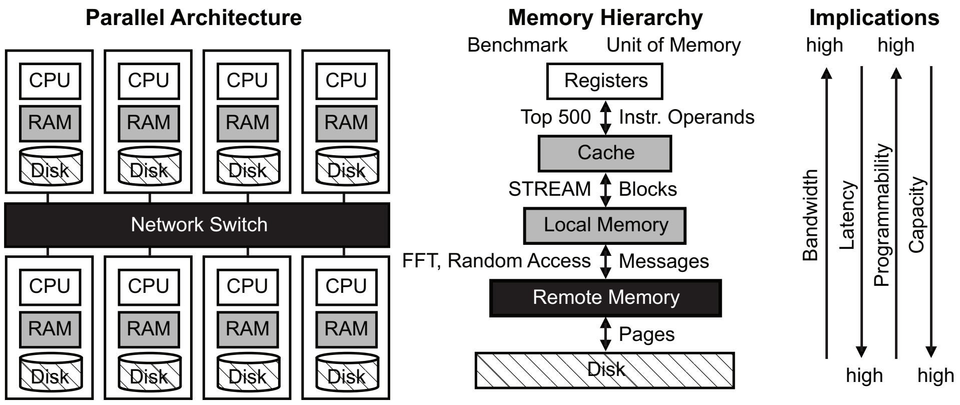


Figure 15.9: **Canonical Parallel Architecture and Memory Hierarchy.**

computed from the formula

$$\text{Latency} = \lim_{m \to 0} T(m) \quad . \tag{15.7}$$

These two parameters are important because some programs require sending a few very large messages (and are limited by bandwidth), and some programs require sending lots of very small messages (and are limited by latency).

Figure 15.10 shows the latency and instantaneous bandwidth for a typical cluster network as a function of message size. The important features of this curve, which are typical of most networks, are the leveling off in latency at small messages and the leveling off in bandwidth that occurs at large messages. While knowing the bandwidth and latency of a system is helpful, what it is even more helpful is comparing these parameters with respect to the computing power of the processor. More specifically, dividing the processor speed (typically measured in floating-point operations per second—FLOPS) by the bandwidth (measured in 8 byte floats per second) yields the number of FLOPS that can be performed in the time it takes to send a message of a given size (see Figure 15.10). This "inverse bandwidth" number provides a good guide to the minimum number of operations that need be performed on a value to amortize the cost of communicating that value.

In this case, for small messages nearly 10,000 operations need to be performed on each 8 byte element. At large messages, only 100 operations need to be performed. These data clearly illustrate a key parallel programming design paradigm: it is better to send fewer large messages than many smaller messages.

What is particularly nice about describing the network relative to the processor performance is that these numbers are relatively constant for most of the systems of interest. For example, these values (10^5 for small messages and 10^2 for large messages) are typical of many systems and do not vary much over time. The absolute processor and network speeds can change dramatically over time, but these ratios remain relatively constant. For example, while processor and networks speeds have improved by almost 10^3 over the last several decades, the relative performance has changed by only a factor of 10.

These parameters suggest that if a parallel program uses large messages, it should be performing >100 operations on each number before communicating the number over the network. Likewise, if a program sends small messages, it should be performing >10,000 operations on each number before sending. Doing less than these amounts will tend to result in a parallel program that doesn't perform very well because most of the time will be spent sending messages instead of performing computations.

Using the simple point-to-point model, we can predict the time for a more complex operation such as a "corner turn" (see Figure 15.11). This requires all-to-all communication where a set of processors P_1 sends a messages of size m to each of a set of processors P_2

$$T_{CornerTurn} = \frac{P_1 P_2 (\text{Latency} + m/\text{Bandwidth})}{Q},$$

where B is the bytes per message and Q is the number of simultaneous parallel paths from processors in P_1 to processors in P_2. Total amount of data moved in this operation is mP_1P_2.

15.5 Parallel Programming Models and Their Impact

The parallel programming model describes how the software is going to implement the signal processing chain on a parallel computer. A good software model allows the different types of parallelism to be exploited: data parallelism, task parallelism, pipeline parallelism, and round-robin. In addition, a good implementation of the parallel programming model allows the type of parallelism to be exploited to change as the system is built and the performance requirements evolve.

There are a number of different parallel programming models that are commonly used. We will discuss three in particular: threaded, messaging, and global arrays, which are also called partitioned global address spaces (PGAS).

The threaded model is the simplest parallel programming model. It is used when a problem can be broken up into a set of relatively independent tasks that the workers (threads) can process without explicitly communicating with each other. The central constraint of the threaded model is that each thread only communicates by writing into a shared- memory address space that can seen by all other threads. This constraint is very powerful and is enormously simplifying. Furthermore it has proved very robust and the vast majority of parallel programs written for shared-memory systems using a small number of processors (i.e., workstations) use this approach. Examples of this technology include OpenMP (www.openmp.org), POSIX Threads or pthreads (www.pasc.org/plato/), and Cilk (supertech.csail.mit.edu/cilk/).

The message passing model is in many respects the opposite of the threaded model. The message passing model requires that any processor be able to send and receive messages from any other processor. The infrastructure of the message passing model is fairly simple. This infrastructure is most typically instantiated in the parallel computing community via the Message Passing Interface (MPI) standard (wwww.mpi-forum.org). The message passing model requires that each processor have a unique identifier and must know how many other processors are working together on a problem (in MPI terminology these are referred to as the processor "rank" and the "size" of the MPI world). Any parallel program can be implemented using the message passing model. The primary drawback of this model is that the programmer must manage every individual message in the system, which can often require a great deal of additional code and can be extremely difficult to debug. Nevertheless there are certain parallel programs that can only be implemented with a message passing model.

The PGAS model is a compromise between the two models. Global arrays impose additional constraints on the program, which allow complex programs to be written relatively simply. In many respects it is the most natural parallel programming model for signal processing because it is implemented using arrays, which are the core data type of signal processing algorithms. Briefly, the global arrays model creates distributed arrays in which each processor stores or owns a piece of the whole array. Additional information is stored in the array so that every processor knows which parts of the array the other processors have. How the arrays are broken up among the processors is specified by a Map (Lebak et al. 2005). For example, Figure 15.12 shows a matrix broken up by rows, columns, rows and columns, and columns with some overlap. The different mappings are useful concepts to have even if the global array model isn't being used. The concept of breaking up arrays in different ways is one of the key ideas in parallel computing. Computations on global arrays are usually performed using the "owner computes" rule, which means that each processor is responsible for doing a computation on the data it is storing locally. Maps can become quite complex and express virtually arbitrary distributions.

In the remainder of this section, we will focus on PGAS approaches.

15.5.1 High-Level Programming Environment with Global Arrays

The pure PGAS model presents an entirely global view of a distributed array. Specifically, once created with an appropriate map object, distributed arrays are treated the same as non-distributed ones. When using this programming model, the user never accesses the local part of the array and all operations (such as matrix multiplies, FFTs, convolutions, etc.) are performed on the global structure. The benefits of pure global arrays are ease of programming and the highest level of abstraction. The drawbacks include the need to implement parallel versions of every single function that may exist in a serial software library. In addition, these functions need to be supported for all possible data distributions. The implementation overhead of a full global arrays library can be quite large.

Fragmented PGAS maintains a high level of abstraction but allows access to local parts of the arrays. Specifically, a global array is created in the same manner as in pure PGAS; however, the operations can be performed on just the local part of the array. Later, the global structure can be updated with locally computed results. This allows greater flexibility. Additionally, this approach does not require function coverage or implementation of parallel versions of all existing serial functions. Furthermore, fragmented PGAS programs often achieve better performance by eliminating the library overhead on local computations.

The first step in writing a parallel program is to start with a functionally correct serial program. The conversion from serial to parallel requires users to add new constructs to their code. In general, PGAS implementations tend to adopt a separation-of-concerns approach to this process which seeks to make functional programming and mapping a program to a parallel architecture orthogonal. A serial program is made parallel by adding maps to arrays. Maps only contain information about how an array is broken up onto multiple processors, and the addition of a map should not change the functional correctness of a program. An example map for the pMatlab (www.ll.mit.edu/pMatlab) PGAS library is shown Figure 15.12. A pMatlab map (see Figure 15.13) is composed of a grid specifying how each dimension is partitioned, a distribution that selects either a block, cyclic, or block-cyclic partitioning, and a list of processors that defines which processors actually hold the data.

The concept of using maps to describe array distributions has a long history. The ideas for pMatlab maps are principally drawn from the High Performance Fortran (HPF) community (Loveman 1993; Zosel 1993), MIT Lincoln Laboratory Space-Time Adaptive Processing Library (STAPL) (DeLuca et al. 1997), and Parallel Vector Library (PVL) (Lebak et al. 2005). A map for a numerical array defines how and where the array is distributed (Figure 15.12). PVL also supports task parallelism with explicit maps for modules of computation. pMatlab and VSIPL++ explicitly only support data parallelism; however, implicit task parallelism can be implemented through careful mapping of data arrays.

For illustrative purposes, we now describe the pMatlab map. The PVL, VSIPL++ as well as many other PGAS implementations use a similar construct. The pMatlab map construct is defined by three components: (1) grid description, (2) distribution description, and (3) processor list. The grid description together with the processor list describes where the data object is distributed, while the distribution describes how the object is distributed (see Figure 15.13). pMatlab supports any combination of block-cyclic distributions up to four dimensions. The API defining these distributions is shown in Figure 15.14.

Block distribution is the default distribution, which can be specified explicitly or by simply passing an empty distribution specification to the map constructor. Cyclic and block-cyclic distributions require the user to provide more information. Distributions can be defined for each dimension and each dimension could potentially have a different distribution scheme. Additionally, if only a single distribution is specified and the grid indicates that more than one dimension is distributed, that distribution is applied to each dimension.

Some applications, particularly image processing, require data overlap, or replicating rows or columns of data on neighboring processors. This capability is also supported through the map interface. If overlap is necessary, it is specified as an additional fourth argument. In Figure 15.14, the fourth argument indicates that there is 0 overlap between rows and 1 column overlap between columns. Overlap can be defined for any dimension and does not have to be the same across dimensions.

While maps introduce a new construct and potentially reduce the ease of programming, they have significant advantages over both message passing approaches and predefined limited distribution approaches. Specifically, pMatlab maps are scalable, allow optimal distributions for different algorithms, and support pipelining.

Maps are scalable in both the size of the data and the number of processors. Maps allow the user to separate the task of mapping the application from the task of writing the application. Different sets of maps do not require changes to be made to the application code. Specifically, the distribution of the data and the number of processors can be changed without making any changes to the algorithm. Separating mapping of the program from the functional programming is an important design approach in pMatlab.

Maps make it easy to specify different distributions to support different algorithms. Optimal or suggested distributions exist for many specific computations. For example, matrix multiply operations are most efficient on processor grids that are transposes of each other. Column and row-wise FFT operations produce linear speedup if the dimension along which the array is broken up matches the dimension on which the FFT is performed.

Maps also allow the user to set up pipelines in the computation, thus supporting implicit task parallelism. For example, pipelining is a common approach to hiding the latency of the all-to-all communication required in parallel FFT. The following pMatlab code fragment for elegantly shows a two-dimensional pipelined FFT run on eight processors:

```
Ymap1b = map([4 1],{},[0:3]);  % Row map on ranks 0,1,2,3
Ymap1c = map([1 4],{},[4:7]);  % Col map on ranks 4,5,6,7
Y1b = complex(zeros(n,m));     % Create Y for step 1b
Y1c = complex(zeros(n,m));     % Create Y for step 1c
...                            % Fill Y with data
Y1b = fft(Y1b,{},1);           % FFT rows (ranks: 0,1,2,3)
Y1c(:,:) = Y1b;                % Cornerturn
Y1c = fft(Y1c,{}21);           % FFT cols (ranks:4,5,6,7)
```

The above fragment shows how a small change in the maps can be used to set up a pipeline where the first half of the processors perform the first part of the FFT and the second half perform the second part. When a processor encounters such a map, it first checks if it has any data to operate on. If the processor does not have any data, it proceeds to the next line. In the case of the FFT with the above mappings, the first half of the processors (ranks 0 to 3) will simply perform the row FFT, send data to the second set of processors, and skip the column FFT, and proceed to process the next set of data. Likewise, the second set of processors (ranks 4 to 7) will skip the row FFT, receive data from the first set of processors, and perform the column FFT.

15.6 System Metrics

At this point, we have described a canonical application and a canonical parallel signal processor. In addition, we have nominally parameterized how the application might be mapped onto a parallel architecture. The actual selection of a parallel mapping is decided by the constraints of the system. This section presents a more formal description of some of the the most common system metrics: performance, efficiency, form factor, and software cost.

15.6.1 Performance

Performance is the primary driver for using a parallel computing system and refers to the time it takes to process one dataset in a a series of datasets. In our application it refers to time to process a SAR image through the entire chain. Performance is usually decomposed into latency and throughput.

Latency refers to the time it takes to get the first image through the chain. Latency is fundamentally constrained by how quickly the consumer of the data needs the information. Some typical latencies for different systems are

- microseconds: Real-Time control/targetting
- milliseconds: Operator in the loop
- seconds: Surveillance systems supporting operations
- minutes: Monitoring systems
- hours: Archival systems

For the SAR application, we will pick a latency target ($T^{goal}_{latency}$) and we may choose to express it in terms T_{input}. For this example, let us set an arbitrary latency goal of

$$T^{goal}_{latency}/T_{input} \approx 10 \ .$$

Throughput is the rate at which the images can be processed. Fundamentally, the data must be processed at the same rate it is coming into the system; otherwise it will "pile up" at some stage in the processing. A key parameter here is the required throughput relative to what can be done on one processor. For this example, let us set an arbitrary throughput goal of

$$S^{goal}_{comp} = T^{goal}_{comp}/T_{comp}(1) \approx 100 \ .$$

15.6.2 Form Factor

One of the unique properties of embedded systems is the form factor constraints imposed by the application. These include the following:

Size. The physical volume of the entire signal processor including its chassis, cables, cooling and power supplies. The linear dimensions (height, width, and depth) and the total volume are constrained by the limitations of the platform.

Weight. The total weight of the the signal processor system.

Power. Total power consumed by the signal processor and its cooling system. In addition, the voltage, its quality, and how often it is interrupted are also constraints.

Heat. The total heat the signal processor can produce that can be absorbed by the cooling system of the platform.

Vibration. Continuous vibration as well as sudden shocks may require additional isolation of the system.

Ambient Air. For an air-cooled system the ambient temperature, pressure, humidity, and purity of the air are also constraints.

IO Channels. The number and speed of the data channels coming into and out of the system.

The form factor constraints are very dramatically based on the type of platform: vehicle (car/truck, parked/driving/off-road), ship (small-boat/aircraft carrier), aircraft (small unmanned air vehicle (UAV) to Jumbo jet). Typically, the baseline for these form factor constraints is what can be found in an ideal environmentally controlled machine room. For example, if the overall compute goal requires at least 100 processing nodes, then in an ideal setting, these 100 processors will require a certain form factor. If these 100 processors were then put on a truck, it might have the following implications:

Size. 30% smaller volume with specific nonstandard dimensions $\Rightarrow$ high-density nodes with custom chassis $\Rightarrow$ increased cost.

Weight. Minimal difference.

Power. Requires nonstandard voltage converter and uninterruptible power supply $\Rightarrow$ greater cost increased size, weight, and power.

Heat. Minimal difference.

Vibration. Must operate on road driving conditions with sudden stops and starts $\Rightarrow$ vibration isolators and ruggedized disk drives $\Rightarrow$ greater cost and increased size, weight, and power.

Ambient Air. Minimal difference.

IO Channels. There are precisely four input channels $\Rightarrow$ four processors must be used in the first processing step.

Processor Selection. Once it has been decided to go ahead and build a signal processing system, then it is necessary to select the physical hardware to use. Often it is the case that the above form factor requirements entirely dictate this choice. [An extreme case is when an existing signal processor is already in place and a new application or mode must be added to it.] For example, we may decide that there is room for a total of five chassis with all the required power, cooling, and vibration isolation requirements. Let's say each chassis has 14 slots. In each chassis, we need one input buffer board, one master control computer (and a spare), and a central storage device. This leaves ten slots, each capable of holding a dual processor node (see Figure 15.15). The result is

$$N_P^{real} = (5 \text{ chassis}) \ \ (10 \text{ slots/chassis}) \ \ (2 \text{ processors/slot})) = 100 \ .$$

At this point, the die is cast, and it will be up to the implementors of the application to make the required functionality "fit" on the selected processor. The procedure for doing this usually consists of first providing an initial software implementation with some optimization on the hardware. If this implementation is unable to meet the performance requirements, then usually a trade-off is done to see if scaling back some of the algorithm parameters (e.g., the amount of data to be processed) can meet the performance goals. Ultimately, a fundamentally different algorithm may be required, combined with heroic efforts by the programmers to get every last bit of performance out of the system.

15.6.3 Efficiency

Efficiency is the fraction of the peak capability of the system the application achieves. The value of $T_{comp}(1)$ implies a certain efficiency factor on one processor relative to the theoretical peak performance (e.g., $\epsilon_{comp} \approx 0.2$). There are similar efficiencies associated with bandwidth (e.g $\epsilon_{comm} \approx 0.5$) and the memory (e.g., $\epsilon_{mem} \approx 0.5$). There are two principal implications of these efficiencies. If the required efficiency is much higher than these values, then it may mean that different hardware must be selected (e.g., nonprogrammable hardware, higher bandwidth networking, or higher density memory). If the required efficiency is well below these values, then it means that more flexible, higher level programming environments can be used, which can greatly reduce schedule and cost.

The implementation of the software is usually factored into two pieces. First, is how the code is implemented on each individual processor. Second, is how the communication among the different processors is implemented. The typical categories for the serial implementation environments follow:

Machine Assembly. Such as the instruction set of the specific processor selected. This provides the highest performance, but requires enormous effort, expertise and offers no software portability.

Procedural Languages with Optimized Libraries. Such as C used in conjunction with the Vector, Signal, and Image Processing Library (VSIPL) standard. This approach still produces efficient code, with less effort and expertise and is as portable as the underlying library.

Object-Oriented Languages with Optimized Libraries. Such as C++ used in conjunction with the VSIPL++ standard. This approach can produce performance comparable to procedural languages with comparable expertise and is usually significantly less effort. Portability may be either more or less portable than procedural approaches depending upon the specifics of the hardware.

High-Level Domain-Specific Languages. Such as MATLAB, IDL, and Mathematica. Performance is usually significantly less than procedural languages, but generally requires far less effort. Portability is limited to the processors supported by the supplier of the language.

The typical categories for the parallel implementation environment are the following:

Direct Memory Access (DMA). This is usually a processor and network-specific protocol for allowing one processor to write into the memory of another processor. It delivers the highest performance, but requires enormous effort and expertise, and offers no software portability.

Message Passing. Such as the Message Passing Interface (MPI). This is a protocol for sending messages between processors. It produces efficient code, with less effort and expertise and is as portable as the underlying library.

Threading. Such as OpenMP or pthreads.

Parallel Arrays. Such as those found in Unified Parallel C (UPC), Co-Array Fortran (CAF), and Parallel VSIPL++. This approach creates parallel arrays using Partitioned Global Address Spaces (PGAS), which allow complex data movements to be written succinctly.

Very rough quantitative estimates for the performance efficiencies of the above approaches are given in Table 15.2 (Kepner 2004; Kepner 2006). The first column (labeled ϵ_{comp}) gives a very rough relative performance efficiency of a single-processor implementation using the approach specified in the first column. The second row (labeled ϵ_{comm}) gives a very rough relative bandwidth efficiency using the approach specified in the first row. The interior matrix of rows and columns shows the combined product of these two efficiencies and reflects the range of performance that can be impacted by the implementation of the software.

Table 15.2: **Software Implementation Efficiency Estimates.** The first column lists the serial coding approach. The second column shows a rough estimate for the serial efficiency (ϵ_{comp}) of the serial approach. The first row of columns 3, 4, 5 and 6 lists the different communication models for a parallel implementation. The second row of these columns is a rough estimate of the communication efficiency (ϵ_{comm}) for these different models. The remaining entries in the table show the combined efficiencies ($\epsilon_{comp}\epsilon_{comm}$). Blank entries are given for serial coding and communication models that are rarely used together.

Serial	Serial		**Comm**	**Model**	
Code	ϵ_{comp}	DMA	Messaging	Threads	PGAS
ϵ_{comm}	-	0.8	0.5	0.4	0.5
Assembly	0.4	0.36			
Procedural	0.2	0.16	0.1	0.08	0.1
Object Oriented	0.18	0.14	0.09	0.07	0.09
High Level	0.04		0.02	0.016	0.02

The significance of the product $\epsilon_{comp}\epsilon_{comm}$ can be illustrated as follows. The overall rate of work can be written as

$$R(N_P) = W/T = W/(T_{comp}(N_P) + T_{comm}(N_P)) \ .$$

Substituting $T_{comp}(N_P) = W/\epsilon_{comp}R_{peak}N_P$ and $T_{comm} = D/\epsilon_{comm}BW_{peak}N_P$ gives

$$R(N_P) = \frac{\epsilon_{comp}\epsilon_{comm}R_{peak}N_P}{\epsilon_{comm} + \epsilon_{comm}(D/W)(R_{peak}/BW_{peak})} \ ,$$

where D/W is the inherent communication-to-computation ratio of the application and R_{peak}/BW_{peak} is the computation-to-communication ratio of the computer. Both ratios are fixed for a given problem and architecture. Thus, the principal “knob” available to the programmer for effecting the overall rate of computation is the combined efficiency of the serial coding approach and the communication model.

15.6.4 Software Cost

Software cost is typically the dominant cost of implementing embedded applications and can easily be 10x the cost of the hardware. There are many approaches to implementing a parallel software system and they differ in performance, effort, and portability. The most basic approach to modeling software cost is provided by the Code and Cost Modeling (CoCoMo) framework (Boehm et al. 1995):

$$\text{Programmer effort [Days]} \approx (\text{Total SLOC})\frac{(\text{New code fraction}) + 0.05(\text{Reused code fraction})}{\text{SLOC/Day}}$$

This formula says that the effort is approximately linear in the total number of software lines of code (SLOC) written. It shows that there are three obvious ways to decrease the effort associated with implementing a program:

Increased Reuse. Including code that has already been written is much cheaper than writing it from scratch.

Higher Abstraction. If the same functionality can be written using fewer lines of code, this will cost less.

Increased Coding Rate. If the environment allows for more lines of code to be written in a given period of time, this will also reduce code cost.

Very rough quantitative estimates for the programming impacts of the above approaches are given in Table 15.3. The first column gives the code size relative to the equivalent code written in a procedural language (e.g., C). The next column gives the typical rate (SLOC/day) at which lines are written in that environment. The column labeled "Serial" is the rate divided by the relative code size and gives the effective relative rate of work done normalized to a procedural environment. The row labeled "Expansion Factor" gives the estimated increase in the size of the code when going from a serial code to a parallel code for the various parallel programming approaches. The row labeled "Effort Factor" shows the relative increase in effort associated with each of these lines of parallel lines of code. The interior matrix combines all of these to give an effective rate of effort for each serial programming environment and each parallel programming environment. For example, in the case of an object-oriented environment, on average each line does the work of two lines in a procedural language. In addition, a typical programmer can code these lines in a serial environment at rate of 25 lines per day. If a code written in this environment is made parallel using message passing, we would expect the total code size to increase by a factor of 1.5. Furthermore, the rate at which these additional lines are coded will be decreased by a factor of two because they are more difficult to write. The result is that the overall rate of the parallel implementation would be 25 effective procedural (i.e., C) lines per day.

Figure 15.16 notionally combines the data in Tables 15.2 and 15.3 for a hypothetical 100-processor system and illustrates the various performance and effort trade-offs associated with different programming models.

Table 15.3: **Software Coding Rate Estimates.** The first column gives the code size relative to the equivalent code written in a procedural language (e.g., C). The next column gives the typical rate (SLOC/day) at which lines are written in that environment. The column labeled "Serial" is the rate divided by the relative code size and gives the effective relative rate of work done normalized to a procedural environment. The row labeled "Expansion Factor" gives the estimated increase in the size of the code when going from serial to parallel for the various parallel programming approaches. The row labeled "Effort Factor" shows the relative increase in effort associated with each of these lines of parallel lines of code. The interior matrix combines all of these to give an effective rate of effort for each serial programming environment and each parallel programming environment given by (SLOC/day)/(Relative SLOC)/(1 + (Expansion Factor - 1)(Effort Factor)).

Serial	Relative	SLOC			**Comm**	**Model**	
Code	SLOC	/Day	Serial	DMA	Messaging	Threads	PGAS
Expansion Factor	-	-	1	2	1.5	1.05	1.05
Effort Factor	-	-	1	2	2	2	1.5
Assembly	3	5	1.6	0.5			
Procedural	1	15	15	5	8	14	14
Object Oriented	1/2	25	50	17	25	45	47
High Level	1/4	40	160		80	140	148

References

[Boehm et al. 1995] Barry Boehm, Bradford Clark, Ellis Horowitz, Ray Madachy, Richard Shelby, and Chris Westland, Cost Models for Future Software Life Cycle Processes: COCOMO 2.0, Annals of Software Engineering, (1995).

[DeLuca et al. 1997] DeLuca, C. M., Heisey, C. W., Bond, R. A., Daly, J. M. A portable object-based parallel library and layered framework for real-time radar signal processing. In Proc. 1st Conf. International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE 97), Pages: 241-248.

[Haney et al. 2005] Haney, R., Meuse T., Kepner, J., and Lebak, J., The HPEC Challenge BenchmarkSuite, High Performance Embedded Computing Conference, Lexington, MA 2005.

[Kepner 2004] Kepner, J (editor). 2004. Special issue on HPC Productivity, International Journal of High Performance Computing Applications 18(4).

[Kepner 2006] High Productivity Computing Systems and the Path Towards Usable Petascale Computing: User Productivity Challenges, J. Kepner (editor), CT Watch, Vol 2, Number 4A, November 2006

[Lebak et al. 2005] Lebak, J., Kepner, J., Hoffmann, H., Rutledge, E. 2005. Parallel VSIPL++: An open standard software library for high-performance parallel signal processing. Proceedings of the IEEE 93(2).

[Loveman 1993] Loveman, D.B. High performance Fortran. Parallel and Distributed Technology: Systems and Applications, IEEE 1(1), 1993.

[Luszczek et al. 2006] P. Luszczek, J. Dongarra and J. Kepner, Design and Implementation of the HPC Challenge Benchmark Suite, CT Watch, Vol 2, Number 4A, November 2006

[Soumekh 1999] Soumekh, Mehrdad, Synthetic Aperture Radar Signal Processing with Matlab Algorithms, Wiley, New York, NY 1999.

[Zosel 1993] Zosel, M.E. High performance Fortran: an overview. Compcon Spring 93, Digest of Papers, San Francisco, CA, February 22-26, 1993.

A Synthetic Aperture Radar Algorithm

This appendix provides the algorithmic details of the HPEC Challenge SAR benchmark.

A.1 Scalable Data Generator

The mathematical details of the Scalable Data Generator (SDG) are beyond the scope of this chapter and not relevant to the subsequent processing. The SDG simulates the response of a real sensor by producing a stream of m_c x n single-precision complex data matrices X, where

$n =$ range (fast-time) samples, which roughly corresponds to the effective bandwidth times the duration of the transmitted pulses.

$m_c =$ cross range (slow-time) samples, which roughly corresponds to the number of pulses sent.

In a real sensor, these input matrices arrive at a predetermined period T_{input}, which translates into an input data bandwidth

$$BW_{input} = (8 \text{ byte})\ n\ m_c / T_{input}\ .$$

The processing challenge is to transform the raw data matrix into a sharp image before the next image arrives.

A.2 Stage 1: Front-End Sensor Processing

This stage reads in the raw SAR data (either from a file or directly from an input buffer) and forms an image. The computations in this stage represent the core of a SAR processing system. The most compute-intensive steps involved in this transformation can be summarized as follows:

Matched Filtering. This step converts the raw data along the range dimension from the time domain to the frequency domain and multiplies the result by the shape of the transmitted pulse.

Digital Spotlighting. Performs the geometric transformations for combining the multiple views.

Interpolation. Converts the data from a polar coordinate system to a rectangular coordinate system.

The result of these processing steps is a n_x x m single precision image I. The core of the matched-filtering step (labeled "1a") consists of performing a FFT on each column of the input matrix X and multiplying it by a set of pre-computed coefficients:

```
for $j = 1 : m_c$
    X(:,j) = FFT(X(:,j)) .* $c_{fast}$(:,1) .* $c_1$(:,j)
end
```

where

X = n x m_c complex single-precision matrix.

(:,j) = j^{th} column of a matrix.

.* elementwise multiplication.

FFT() performs complex-to-complex one-dimensional FFT.

c_{fast} = n x 1 complex vector of pre-computed coefficients describing the shape of the transmitted pulse.

c_1 = n x m_c complex single-precision matrix of pre-computed coefficients .

The computational complexity of this step is dominated by the FFT and is given by

$$W^{1a}_{stage} = 5\ n\ m_c\ (2 + log_2(n)) \quad \text{[FLOPS]} .$$

The parallelism in this step is principally that each column can be processed independently, which implies that there are m_c degrees of parallelism (DOP). Additional parallelism can be found by running each FFT in parallel. However, this requires significant communication. At the completion of this step, the amount of data sent to the next step is given by

$$D^{1a \to 1b}_{stage} = 8\ n\ m_c \quad \text{[bytes]} .$$

The digital spotlighting step (labeled "1b") consists of performing a FFT on each row of X, which is then copied and offset into each row of a larger matrix Y. An inverse FFT of each row of Y is then multiplied by a set of pre-computed coefficients and a final FFT is performed. Finally, the upper and lower halves of each row are swapped via the FFT_{shift} command. The result of this algorithm is to interpolate X onto the larger matrix Y.

```
for $i = 1 : n$
    X(i,:) $=$ FFT(X(i,:))
    Y(i,1:$m_c/2$)) $= m/m_c$ .* X(i,1:$m_c/2$)
    Y(i,$m_c/2 + m_z + 1 : m$) $=$ (m/mc) .* X(i,$1 + m_c/2 : m_c$)
    Y(i,:) $=$ FFT$_{shift}$(FFT( FFT$^{-1}$(Y(i,:)) .* c$_2$(i,:) ))
end
```

where

$m_z = m - m_c$.

Y $=$ n x m complex single-precision matrix.

(i,:) $=$ ith row of a matrix.

(i$_1$:i$_2$,:) $=$ sub-matrix consisting of all rows i$_1$ to i$_2$.

FFT$_{shift}$() swaps upper and lower halves of a vector.

FFT^{-1}() performs complex-to-complex one-dimensional inverse FFT.

c$_2$ $=$ n x m complex single-precision matrix of pre-computed coefficients.

The computational complexity of this step is dominated by the FFTs and is given by

$$W^{1b}_{stage} = 5\ n\ (m_c(1 + log_2(m_c)) + m(1 + 2log_2(m)))\ .$$

The parallelism in this step is principally that each row can be processed independently, which implies that there are n degrees of parallelism. At the completion of this step, the amount of data sent to the next step is given by

$$D^{1b \rightarrow 1c}_{stage} = 8\ n\ m\ \ \text{[bytes]}\ .$$

The backprojection step (labeled "1c") begins by completing the two-dimensional FFT$_{shift}$ operation from step 1b. This consists of performing aFFT$_{shift}$ operation on each column, which is then multiplied by a pre-computed coefficient. The core of the interpolation step involves summing the projection of each element of $Y(i,j)$ over a range of values in $Z(i - i_K : i + i_K)$ weighted by the $sinc()$ and $cos()$ functions. The result of this algorithm is to project Y onto the larger matrix Z.

```
for $j = 1 : m$
    Y(:,j) $=$ FFT$_{shift}$(Y(:,j)) .* c$_2^{-1}$(:,j)
end

for $i = 1 : n$
    for $i_K = -n_K : n_K$
        for $j = 1 : m$
            Z($i_{KX}(j) + i_K$,j) += Y(i,j) .*
```

```
                sinc(f$_1$(i,i$_k$,j)) .* (0.54 + 0.46 cos(f$_2$(i,k$_k$,j))
            **end**
        **end**
    **end**
```

where

Z = n_x x m complex single-precision matrix

n_K = half-width of projection region

f$_{1,2}$ are functions that map indices into coordinates to be used by sinc() and cos() functions.

The computational complexity of this step is dominated by sinc() and cos() functions used in the the backprojection step,

$$W^{1c}_{stage} = 2\ n\ m\ n_K\ (O(sinc) + O(cos)) \approx 40\ n\ m\ n_K\ .$$

The parallelism in this step is principally that each column can be processed independently, which implies that there are n degrees of parallelism. This dimension is preferred because the interpolation step spreads values across each column. If this step were made parallel in the row dimension, this would require communication between neighboring processors. The parallelism in this step is the same as the beginning of the next step, so no communication is required.

The goal of the frequency to spatial conversion step (labeled "1d" and "1e") is to convert the data from the frequency domain to the spatial domain and to reorder the data so that it is spatially contiguous in memory. This begins by performing a FFT on each column, multiplying by a pre-computed coefficient and then circularly shifting the data. Next a FFT is performed on each row, multiplied by a pre-computed coefficient and circularly shifted. Finally, the matrix is transposed and the absolute magnitude is taken. The resulting image is then passed on to the next stage.

```
for $j = 1 : m$
    Z(:,j)=cshift(FFT$^{-1}$(Z(:,j)) .* c$_3$(j),ciel($n_x$/2))
end

for $i = 1 : n_x$
    Z(i,:)=cshift(FFT$^{-1}$(Z(i,:) .* c$_4$(i),-ciel(m/2))
end

I = |Z|$^T$
```

where

cshift(,n) circular shifts a vector by n places.

c$_3$ = m complex single-precision vector of pre-computed coefficients.

c$_4$ = n_x complex single-precision vector of pre-computed coefficients.

I = n_x x m real single-precision matrix.

The computational complexity of these steps is dominated by the FFTs and is given by

$$W^{1d}_{stage} = 5\ m\ (n_x(1 + log_2(m))$$

and

$$W^{1e}_{stage} = 5\ n_x\ (m(1 + log_2(n_x))\ .$$

The parallelism for step 1d is principally that each column can be processed independently, which implies m degrees of parallelism. For step 1e, the parallelism is principally that each row can be processed independently, which implies n_x degrees of parallelism. The amount of data sent between these steps is given by

$$D^{1d \rightarrow 1e}_{stage} = 8\ n_x\ m\ \ \text{[bytes]}\ .$$

The above steps complete the image formation stage of the application. One final step, that is a negligible untimed part of the benchmark, is the insertion of templates into the image. These templates are used by the next stage. Each template is a $n_{font} \times n_{font}$ matrix containing an image of a rotated capital letter. The total number of different templates is given by $n_{let} n_{rot}$. The templates are distributed on a regular grid in the image and with an occupation fraction of 1/2. The grid spacing is given by $4n_{font}$, so that the total number of templates in an image is

$$n_{templates} = floor(m/(4n_{font}))floor(n_x/(4n_{font}))\ .$$

A.3 Stage 2: Back-End Knowledge Formation

This stage reads in two images (I$_1$ and I$_2$) of the same region on the ground and differences them to find the changes. The differenced image is thresholded to find all changed pixels. The changed pixels are grouped to find a region of interest that is then passed into a classifier. The classifier convolves each region of interest with all the templates to determine which template has a best match.

```
$I_\Delta$ = max($I_2$ - $I_1$,0)
$I_{mask}$ = $I_\Delta$ > $c_{thresh}$
i = 1
**while**( NonZeros($I_{mask}$) )        ROI(i,1:4) = PopROI($I_{mask}$,$n_{font}$)
    $I_{sub}$ = $I_\Delta$(ROI(i,1):ROI(i,2),ROI(i,3):ROI(i,4))
    ROI(i,5:6) = MaxCorr(T,$I_{sub}$)
**end**
```

where

I$_\Delta$ = m x n_x single-precision matrix containing the positive difference between sequential images I$_1$ and I$_2$.

c$_t hresh$ = pre-computed constant which sets the threshold of positive differences to consider for additional processing.

I$_{mask}$ = m x n_x logical matrix with a 1 wherever the difference matrix exceeds c$_{thresh}$.

NonZeros() = returns the number of nonzeros entries in a matrix.

ROI = $n_{templates}$ x 6 integer matrix. First four values hold the coordinates marking the ROI. The final two values hold the letter and rotation index of the template that has the highest correlation with the ROI.

PopROI(I$_{mask}$,n_{font}) selects the "first" nonzero pixel in I$_{mask}$ and returns four values denoting the n_{font} by n_{font} region of interest around this pixel. Also sets these locations in I$_{mask}$ to zero.

I$_{sub}$ = n_{font} x n_{font} single-precision matrix containing a subregion of I$_{\Delta}$.

T = n_{let} x n_{rot} x n_{font} x n_{font} single-precision array containing all the letter templates.

MaxCorr(T,I$_{sub}$) correlates I$_{sub}$ with every template in T and returns the indices corresponding to the letter and rotation with the highest correlation.

The computational complexity of this stage is dominated by computing the correlations. Each correlation with each template requires 2 n_{font}^4 operations. The total computational complexity of this stage is

$$W_{stage}^2 = 2\ n_{template}\ n_{let}\ n_{rot}\ n_{font}^4\ .$$

In the previous stage, at each point in the processing, either a row or a column could be processed independently of the others. In this stage, two-dimensional regions are involved in the calculation. Assuming the preferred parallel direction is that coming out of the previous stage (i.e., the second dimension), then this would imply n_x degrees of parallelism. However, the computation of column depends upon n_{font} neighboring columns. To effect this computation requires that overlapping data is stored on each processor. This effectively limits the degrees of parallelism to n_x/n_{font}. The amount of data sent between Stage 1 and Stage 2 is what is needed implement this overlap

$$D_{stage}^{1\rightarrow 2} = 8\ m\ N_P\ n_{font}\ [\text{bytes}]\ .$$

An alternative to the above approach is to break up the data in both dimensions, which exposes more parallelism ($m\ n_x/n_{font}^2$). However, this requires more communication to set up (8 m n_x [bytes]).

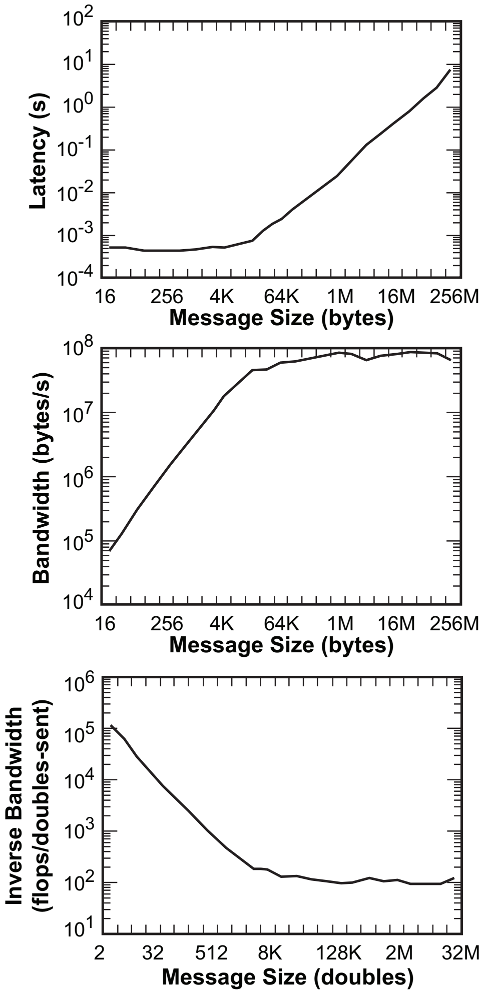


Figure 15.10: **Network performance.** Latency, bandwidth, and “inverse bandwidth” as a function of message size for a typical cluster network. Inverse bandwidth is shown in terms of the number of FLOPS/doubles-sent.

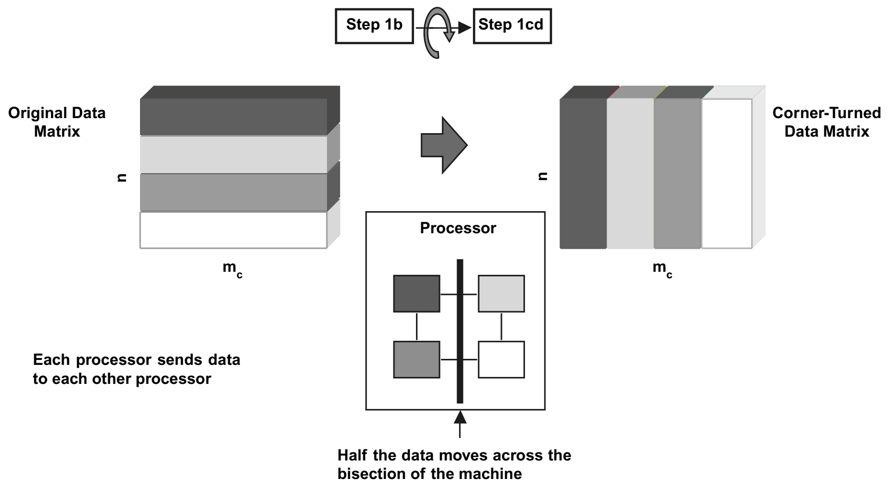

Step 1b
Step 1cd
Original Data Matrix
n
m_c
Corner-Turned Data Matrix
n
m_c
Processor
Each processor sends data to each other processor
Half the data moves across the bisection of the machine

Figure 15.11: **Corner Turn.**

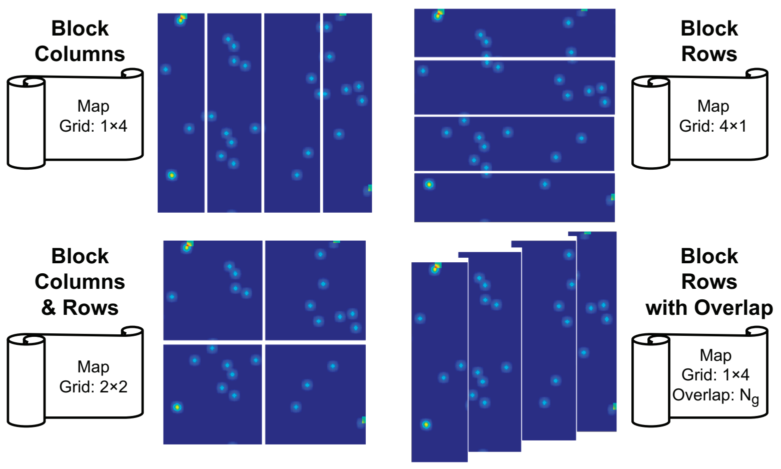


Figure 15.12: **Global array mappings.** Different parallel mappings of a two-dimensional array. Arrays can be broken up in any dimension. A block mapping means that each processor holds a contiguous piece of the array. Overlap allows the boundaries of an array to be stored on two neighboring processors.

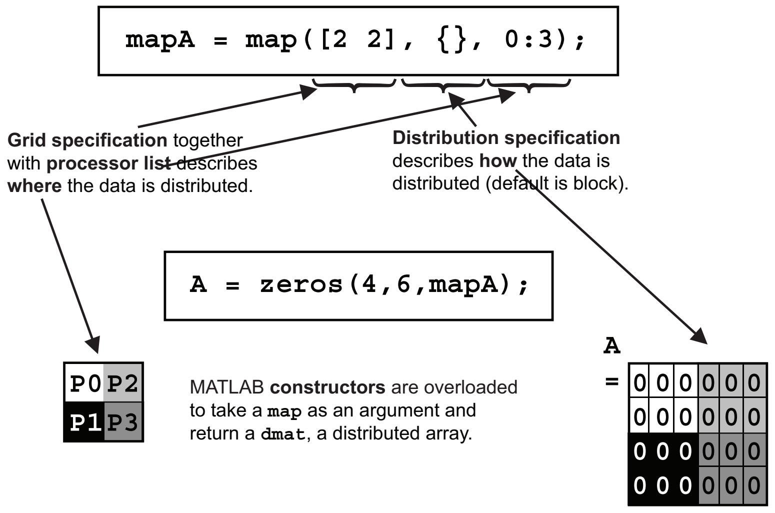
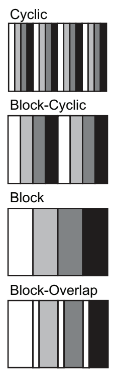

mapA = map([2 2], {}, 0:3);
Grid specification together with processor list describes where the data is distributed.
Distribution specification describes how the data is distributed (default is block).
A = zeros(4,6,mapA);
P0 P2
P1 P3
MATLAB constructors are overloaded to take a map as an argument and return a dmat, a distributed array.
A =
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0
0 0 0 0 0 0

Figure 15.13: **Anatomy of a Map.** A map for a numerical array is an assignment of blocks of data to processing elements. It consists of a grid specification (in this case a 2 x 2 arrangement), a distribution (in this case {} implies that the default block distribution should be used), and a processor list (in this case the array is mapped to processors 0, 1, 2, and 3).



```
Np = pMATLAB.comm_size;                       % Set number of processors.
N = 16;                                       % Set size of row vector.

dist_spec.dist = ‘c’;                         % Define cyclic distribution.



dist_spec.dist = ‘bc’;                        % Define block-cyclic distribution.
dist_spec.size = ‘2’;                         % Set block size = 2.


dist_spec.dist = ‘b’;                         % Define block distribution.
Amap = map ([1 Np], dist_spec,0:Np-1);        % Create a map.


% Map with overlap of 1.
Amap = map ([1 Np], dist_spec,0:Np-1,[0 1]);

A = zeros (1,N,Amap);                         % Create a distributed array.
```

Figure 15.14: **Block Cyclic Distributions.** Block distribution divides the object evenly among available processors. Cyclic distribution places a single element on each available processor and then repeats. Block-cyclic distribution places the specified number of elements on each available processor and then repeats.

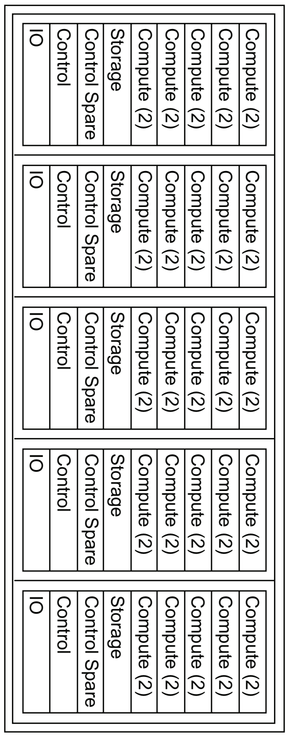


Figure 15.15: **Example Computing Rack.** A canonical signal processing rack. Each rack contains five chassis. Each chassis has 14 slots. Four of the slots may need to be reserved for IO, control (and spare), and storage. The result is that 100 processors can be fit into the entire rack.

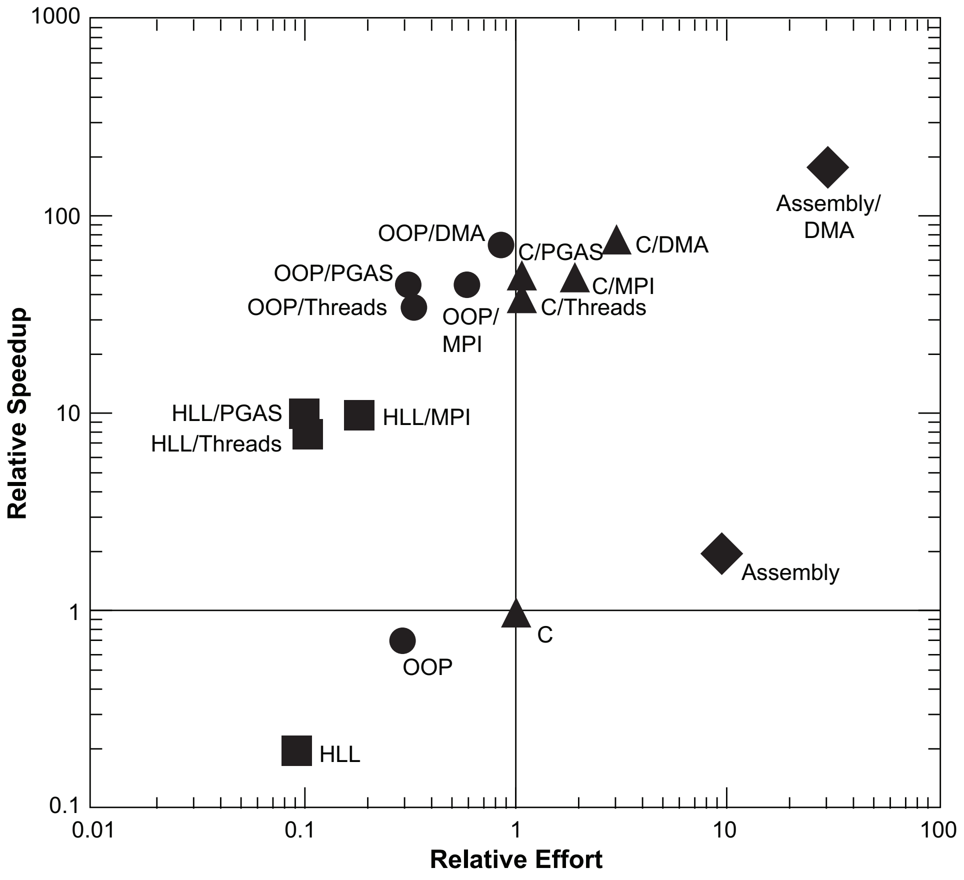


Figure 15.16: **Speedup vs. Effort.** Estimated relative speedup (compared to serial C) on a hypothetical 100-processor system plotted against estimated relative effort (compared to serial C).