Master's Thesis:

Simulation of Communication Time for a Space-Time Adaptive Processing Algorithm on a Parallel Embedded System

Jack M. West

Department of Computer Science

Texas Tech University

August 1998

Committee Members:

Dr. John K. Antonio (Chairperson)

Dr. Donald L. Gustafson

Dr. Noe Lopez-Benitez

Dr. Milton L. Smith

ACKNOWLEDGMENTS

Many people have contributed to the completion of this research effort, and I am grateful to them all. I would particularly like to acknowledge the following people for their contributions:

To my chairperson, Dr. John K. Antonio, for his patience, support, guidance, and technical expertise throughout this research effort. He is an excellent example of professionalism and extraordinary assistance.

Dr. Donald L. Gustafson, Dr. Noe Lopez-Benitez, and Dr. Milton L. Smith for their support and ceaseless encouragement.

Members of the High Performance Computing Lab at Texas Tech University, Jeff Muehring, Nikhil Gupta, and Tim Osmulski, for their countless hours of brainstorming and technical enlightenment.

The direct encouragement and support of H. Paul Haiduk who first introduced me to Computer Science. Without him, I would not be here. Thank you.

This research was supported by Rome Laboratory under grant F30602-96-1-0098 and by the Defense Advanced Research Projects Agency (DARPA) under contract F30602-97-2-0297.

To my faithful friends who have provided continual encouragement throughout my educational endeavor.

As always, my thanks and love to my beautiful Kasey. You are my never-ending confidante and advocate. I can never describe in words how much you truly mean to me. Together, I look forward to the rest of our life, and I know that the best is yet to come.

Finally, to my parents, Johnny and Jeanette West, for sharing their wisdom and patience with me throughout my life. I can never thank them enough.

CONTENTS

ABSTRACT

This thesis involves the investigation of parallelization and performance improvement for a class of radar signal processing techniques known as space-time adaptive processing (STAP). The assumed platform, which consists of multiple DSPs, is the commercially available Mercury RACE System. The main contribution of the thesis is the design and implementation of a network simulator for the RACE system. This simulator allows for the performance of various parallel STAP algorithm implementations to be predicted for existing or future RACE system configurations.

A major challenge of implementing parallel STAP algorithms on multiprocessor systems is determining the best method for distributing the 3-D data cube across CEs of the multiprocessor system (i.e., the mapping strategy) and the scheduling of communication within each phase of computation. It is important to understand how mapping and scheduling strategies affect overall performance. The network simulator developed in this thesis is used to evaluate the performance of various mapping and scheduling strategies.

FIGURES

CHAPTER I

INTRODUCTION

1.1 Background

After taking office, the Clinton Administration launched an extensive investigation researching new methods and procedures for the procurement of federal government goods and services. In an attempt to assist in the reduction of waste and hidden costs, President Clinton, in a 1994 executive order, directed all heads of executive agencies to "increase the use of commercially available items where practicable, place more emphasis on past contractor performance, and promote best value rather than simply low cost in selecting sources of supplies and services" [1].

In addition to re-engineering the policies of governmental acquisition, the Clinton Administration drastically reduced defense expenditures. As a result of the changing, and perhaps advancing, governmental procurement methodology and military cost reductions, the Department of Defense (DoD) is moving towards commercial-off-the-shelf (COTS) products for the design and deployment of military systems. There are a number of embedded military applications such as airborne target recognition systems, undersea sonar platforms, ground processing stations, and command and control systems in which non-commercial resources are being abandoned. In particular, COTS parallel processing systems are replacing custom embedded military sonar and radar systems on ships and airborne aircraft [12].

In contrast to contemporary non-commercial products that involve costly custom engineering, ideally, COTS products offer lower cost hardware, faster development that reduces program lifecycle costs, and higher reliability while adhering to strict size, weight, and power (SWAP) requirements of many military applications. These characteristics of commercial products are achievable simply because of volume production and compatibility with a wide range of applications. Furthermore, the practice of purchasing COTS equipment creates a competitive market that stimulates both technological advancement and decreased costs [12].

As the demand for commercial embedded military parallel processing systems rise, the number of companies producing practical solutions to military-based platforms is also increasing. Mercury Computer Systems, Inc. plays a significant role in providing platforms for DoD computationally-intensive embedded applications. Mercury's primary role for such applications involves supplying very high-performance real-time computing and data I/O capability [4]. Mercury Computer Systems provides state-of-the-art embedded real-time multicomputer systems for typical digital signal processing platforms for intelligence data collection and processing.

Digital signal processing is one of the core technologies central to the operation of military-based radar systems. Digital signal processing is the application of mathematical operations on a digitally represented sequence of samples from an analog signal. Since their emergence in the late 1980s, digital signal processors (DSPs) have experienced tremendous growth rates in areas of signal processing due to reductions in costs, advances in DSP architectures, and improvements in development tools. Simply stated, a DSP is a special purpose microprocessor similar to a traditional microprocessor (e.g., Intel Pentium) that is optimized to perform mathematical operations such as multiplications, additions, and subtractions with greater efficiency. In addition to their increased performance for a class of computations, DSPs are generally silicon conservative and less expensive than general-purpose microprocessors.

Classical signal processing algorithms are characterized by the need for high-performance computing and involve repetitive, numerically-intensive tasks, which are ideally suited to DSP technology. Processing speeds of a single DSP are often insufficient to satisfy the computation demand of military-based signal processing applications. For such real-time signal processing applications, parallel processing is required to meet the necessary performance requirements.

1.2 Focus of the Thesis

This thesis involves the investigation of parallelization and performance improvement for a class of radar signal processing techniques known as space-time adaptive processing (STAP). The assumed platform, which consists of multiple DSPs, is

the commercially available Mercury RACE System [4]. The main contribution of the thesis is the design and implementation of a network simulator for the RACE system. This simulator allows for the performance of various parallel STAP algorithm implementations to be predicted for existing or future RACE system configurations.

STAP involves signal processing methods that operate on data collected from a set of spatially distributed sensors over a given time interval. Signal returns are composed of range, pulse, and antenna-element digital samples; consequently, a three-dimensional (3-D) data cube naturally represents STAP data. STAP algorithms can provide improved target detection in the presence of interference through the adaptive nulling of both ground clutter and signal jamming [18]. Typical parallel STAP involves simultaneous processing of the spatial signals received by the distinct elements of an array antenna and the temporal signals received from multiple pulses of a coherent radar waveform.

Typical processing requirements for STAP range from 10-100 giga floating point operations (Gflops), which can only be met by multiprocessor systems composed of numerous interconnected compute elements (CEs) [13]. A CE contains a processor, local memory, and a connection to the network that interconnects the CEs. In most parallel STAP implementations, there are phases of computation in which data must be exchanged among CEs. A major challenge of implementing parallel STAP algorithms on multiprocessor systems is determining the best method for distributing the 3-D data cube across CEs of the multiprocessor system (i.e., the mapping strategy) and the scheduling of communication within each phase of computation. It is important to understand how mapping and scheduling strategies affect overall performance. The network simulator developed in this thesis is used to evaluate the performance of various mapping and scheduling strategies.

The remainder of this thesis is divided eight chapters. Chapter II provides an overview of radar signal processing and a computation complexity analysis of two STAP algorithms, namely fully-adaptive STAP and a partially adaptive heuristic (element-space post-Doppler STAP) used to approximate the optimal solution. Chapter III briefly introduces the basic components of Mercury's RACE multicomputer including a description of the CEs, the RACEway interconnection network, and network contention

resolution schemes. Chapter IV illustrates the challenges associated with implementing STAP algorithms on a parallel-processing computer. Two basic paradigms for distributing the 3-D STAP data cube among CEs of Mercury's RACE system are described. Chapter V presents small-scale examples to illustrate the effects that mapping and scheduling choices can have on network performance. In Chapter VI, the design of the simulator, using the Unified Model Language (UML), is described and illustrated. Chapter VII presents some numerical studies involving timing information obtained from the simulator. Finally, Chapter VIII concludes the work with a summary of the research and results.

CHAPTER II

OVERVIEW OF STAP

Current methods of radar date back to 1924, when the height of the ionosphere was first measured [16]. By 1935, the military started developing radar-based weapon systems, and shortly after, at the outbreak of the war in 1939, military radar stations were in operation. During the war, the military concealed knowledge of radar technology for obvious strategic reasons. Consequently, detailed technical information about radar was not released to the public until after the war. Today, radar technology has become an integral part of real-time signal and image processing for defense and commercial applications. Modern airborne radar systems are required to detect smaller and smaller targets in the presence of clutter and interference. Space-time adaptive processing algorithms have been developed to extract a desired signal from potential target returns comprised of Doppler shifts resulting from radar platform motion, clutter returns, and interference including jamming. The sections below provide a brief overview of radar signal processing and STAP methods. For a thorough theoretical analysis of STAP, the reader is referred to [2, 18].

2.1 Radar Signal Processing

The basic concept of radar is relatively simple, although its practical implementation is not so trivial. In military environments, radar is used to extend the capability of human's senses for observing the environment, especially the sense of vision. The basic purpose of radar is to detect the presence of an object of interest and provide information concerning that object's range, velocity, angular coordinates, size, and other parameters [11]. Radar operates by radiating electromagnetic (EM) energy, oscillating at a predetermined frequency, f, and duration, τ, into free space through an antenna. In general, the radar antenna forms a beam of EM energy that concentrates the EM wave into a given direction [3]. By effectively rotating and pointing the antenna, the transmitted radar signal can be directed to a desired angular coordinate.

An object or target located within the path of the transmitted radar beam will intercept a portion of the EM energy. The intercepted energy will be scattered in various directions from the target depending on the target's physical characteristics. In general, some of the transmitted energy will be reflected back in the direction of the radar. This retro-reflected energy is referred to as backscatter [3]. A portion of the backscattered wave or echo return is received by the radar antenna. The echo returns, which are gathered by a set of sensors, are sampled, and the resulting data is processed to identify targets and perform parameter estimation. The distance to the target is determined by measuring the time taken for the radar signal to travel to the target and back. Furthermore, the angular position of the target may be determined by the arrival direction of the backscattered wave. If relative motion exists between the target and radar, the shift in the carrier frequency of the reflected wave, also known as the Doppler effect, is a measure of the target's relative velocity and may be used to distinguish moving targets from stationary objects [14].

The basic role of the radar antenna is to act as a transducer between the free-space propagation and guided-wave propagation of the EM wave [15]. The specific function of the antenna during transmission is to concentrate the radiated energy into a shape beam directive that illuminates targets in a desired direction. During reception, the antenna collects the energy from the reflected echo returns. Many varieties of radar antennas have been used in radar systems. The type of radar antenna selected for a certain application depends not only on the electrical and mechanical requirements dictated by the radar design specifications but also on its application. In airborne-radar applications, radar antennas must generate beams with shape directive patterns that can be scanned.

The properties offered by antenna arrays are quite appealing to airborne radar systems. Antenna arrays consist of multiple stationary elements, which are fed coherently, and use phase or time-delay control at each element to scan a beam to given angles in space [8]. The primary reason for using radar arrays is to produce a directive beam that can be repositioned electronically. An electronically steerable antenna array, whose beam steering is inertialess, is drastically more cost effective when the mission requires surveying large solid angles while tracking a large number of targets [8]. Additionally, arrays are

sometimes used in place of fixed aperture antennas because the multiplicity of elements allows a more precise control of the radiating pattern.

The purpose of moving-target indication (MTI) radar is to reject signal returns from stationary or unwanted slow-moving targets, such as buildings, hills, tree, sea, rain, and snow, and retain detection information on moving targets such as aircraft and missiles [12]. The term Doppler radar refers to any radar capable of measuring the shift between the transmitted frequency and the frequency of reflections received from possible targets [16]. Relative motion between a signal source and a receiver creates a Doppler shift of the source frequency. When a radar system intercepts a moving object that has a radial velocity component relative to the radar, the reflected signal's frequency is shifted. For example, consider a radar that emits a pulse of EM energy that is intercepted by both a building (fixed target) and an airplane (moving target) approaching the radar. As previously stated, each of the objects will scatter the intercepted radar signal, which will include a portion of backscatter energy. After the reflected radar signal returns to the radar in a certain time period, a second pulse of EM energy is transmitted. The reflection of the second pulse of energy from the building is returned to the antenna in the same time period as the first pulse. However, the reflection of the second pulse from the moving aircraft returns to the antenna in less time than the first pulse because the aircraft is moving towards the radar. This time change between pulses is determined by comparing the phase of the received signal with the phase of the reference oscillator of the radar [12]. If the phase of received consecutive pulses change, the object of interest is in motion.

2.2 STAP Algorithms

The objective of many airborne radar systems is to search the given space for potential targets. Future airborne radars will be required to detect increasingly smaller targets in the presence of interference such as clutter, jamming, noise, and platform motion. If the interference is localized in frequency and comes from a limited number of sources, targets can be detected by using adaptive spatial weighting of the data from each element of an antenna array [2]. By applying the computed weights to the data, the effects of interference can be reduced thus increasing the reception of the reflected signal. For an

airborne radar platform that is in motion, the Doppler spread of the clutter returns is significantly wider, and the clutter characteristics are highly variable due to the changing ground terrain. For this reason, the use of an antenna array provides the potential for improved airborne radar performance. Because of the added dimensionality of received data, the weights must now be adapted from the data in both the time and space dimensions. This signal processing method is referred to as STAP, which is an adaptive processing technique that simultaneously combines the signals received from multiple elements of an antenna array (the spatial domain) and from multiple pulses (the temporal domain) of a coherent processing interval (CPI) [18]. The paragraphs to follow provide a general description of the computation complexity involved in implementing two STAP algorithms, namely fully adaptive STAP and element-space post-Doppler. For a detailed theoretical foundation and computational complexity analysis of STAP algorithms, the reader is referred to [2, 18].

Consider an N element airborne radar array that transmits a coherent burst of M pulses at a constant pulse repetition frequency (PRF) $f_r = 1/T_r$, where T_r is the pulse repetition interval (PRI). The time interval over which the EM echo returns are collected is referred to as the coherent processing interval (CPI), and the resultant length of one CPI is MT_r. For each of the M pulses, L range samples are collected by each array element. With M pulses and N array channels, the return signal for one CPI is composed of LMN complex signal samples [18]. Because the signal returns are composed of L range gates, M pulses, and N antenna array samples, the data may be visually represented by the three-dimensional data set shown in Fig. 2.1. This $L \times M \times N$ data set will be referred to as the CPI data cube [18].

Let x_{nml} represent the n^{th} array element and the m^{th} pulse at the l^{th} range sample time [26]. Next, define $x_{m,l}$ to represent an $N \times 1$ column vector, or a spatial snapshot, composed of the complex return signals from each array element for the m^{th} pulse and the l^{th} range. By combining all of the spatial snapshots at a given range of interest, an $N \times M$ matrix X_l can be formed, where $X_l = \left[x_{1,l}, x_{2,l}, x_{3,l}, \cdots, x_{M,l}\right]$. The shaded plane in Fig. 2.1, referred to as a range gate, represents the X_l spatial snapshot at the l^{th} range. To detect the presence

of a target within a range gate, a linear filter or space-time processor combines the data samples from the range gate to produce a scalar output, which is then typically passed through a threshold process for target detection.

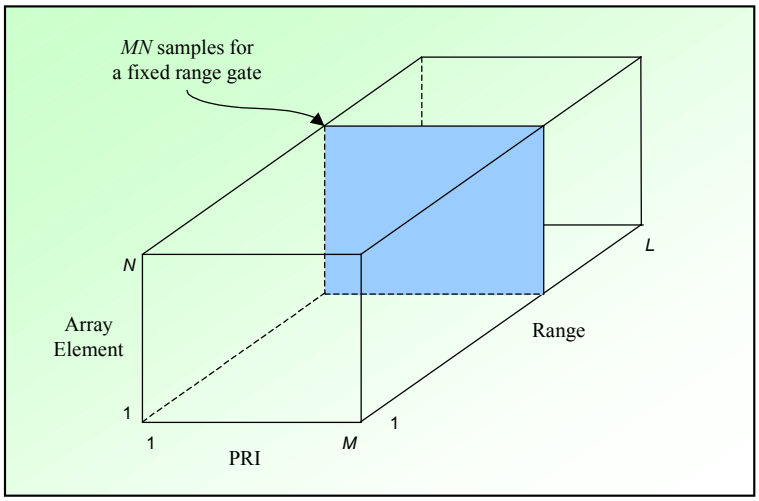

Fig. 2.1 The STAP CPI three-dimensional data cube (derived from [18]).

Three pipelined phases of processing comprise the generic space-time processor (see Fig. 2.2). First, a set of rules called the training strategy is applied to the data to estimate the interference. The objective of training strategy is to provide a good estimate of the interference at a given range gate. Because the interference is unknown, the training data is estimated data-adaptively from the STAP data cube.

The training data computed in phase one is used as input to calculate the adaptive weight vector in phase two. In general, the weight computation phase is the most computation-intense portion of the space-time processor. Typically, weight computation requires the solution of a linear system of equations [18]. Additionally, each time the training data changes, a new weight vector must be computed. The most common weight computation strategy is called sample matrix inversion (SMI). In an SMI approach, the weight vector is computed from the inverse of the covariance matrix of training data or a QR-Decomposition of the matrix of training data. After calculating a single weight vector, the final phase of weight application commences.

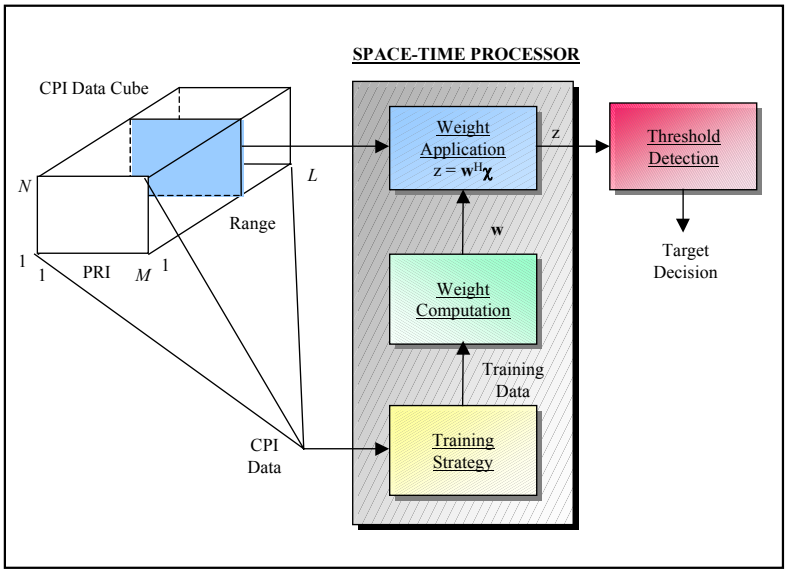

Fig. 2.2 Generic space-time adaptive processor (derived from [18]).

In the final phase, a scalar output is obtained by computing the inner product of the weight vector and range gate of interest. The scalar output is compared to a threshold value to determine if a target is present at a specified angle and Doppler [18]. Because a potential target's angle and velocity are unknown, the space-time processor computes multiple weight vectors to cover all possible target angles, ranges, and velocities at which target detection is to be queried [18].

Fully adaptive STAP refers to a space-time processor that computes and applies a separate adaptive weight to every array element and pulse. The size of the weight vector for fully adaptive STAP is MN. In order to compute the weight vector, a system of MN linear equations with dimension MN must be solved; thus, computing a single weight vector requires a $O((MN)^3)$ operations [18]. For many conventional radar systems, the product of MN may vary from several hundred to several thousand with M and N both ranging from 10 to several hundred. Furthermore, a weight vector must be calculated for each training set used. The sheer computational complexity necessary to compute the weight vectors for fully adaptive STAP, in real-time, is typically beyond the capabilities of current computing systems (especially in cases where there is limited power and space for the computing

system onboard an aircraft). This fact alone renders fully adaptive STAP impractical and provides adequate motivation for the formulation of alternative heuristic algorithms.

The goal of partially adaptive algorithms is to break the fully adaptive problem into reduced-dimension adaptive problems while maintaining near-optimal results. A partially adaptive processor gathers the large set of input signals from the CPI data cube, transforms them into a reduced number of signals, and solves the reduced-dimension filtering problem with the newly transformed data [18]. Partially adaptive algorithms are classified according to the type of preprocessing performed first. For instance, in element-space pre-Doppler STAP adaptive-processing is followed by Doppler filtering.

In element-space STAP algorithms, every array element is adaptively weighted. The advantage of element-space approaches is that they retain full spatial dimensionality while decreasing the overall problem size by reducing the number of temporal degrees of freedom prior to adaptation [18]. Algorithms belonging to the class of element-space post-Doppler STAP perform filtering on the data along the pulse dimension, referred to as Doppler filtering, for each channel prior to adaptive filtering. After Doppler filtering, an adaptive weight problem is solved for each range and pulse data vector. By using element-space post-Doppler STAP, the computational complexity is reduced to M separate N-dimensional adaptive problems. The focus of the proposed research assumes that STAP will be implemented using the element-space post-Doppler partially adaptive algorithm.

CHAPTER III

AN OVERVIEW OF THE PARALLEL SYSTEM

Since the conceptual development and implementation of serial computers in the mid 1940's, their computing speed, complexity, and reliability has steadily and drastically increased to meet the demands of emerging problems. However, the physical constraint imposed by the speed of light limits indefinite improvements in the serial computer domain. Because of the imposed physical constraints, serial computers are unable to meet the throughput requirements necessary to solve certain complex real-time applications such as embedded medical image processing and military signal processing. A natural way to circumvent this problem is to use an ensemble of processors to solve both existing and future problems. The fifth generation of computers is emphasizing scalable parallel processing machines to solve complex large-scale problems. Parallel processing has emerged as a key hardware technology in modern computers, driven mainly by the demand for higher performance, lower costs, and sustained productivity in real-time applications [5].

3.1 Parallel Architectures

In general, parallel architectures may be categorized into two fundamental classes, namely, shared-memory multiprocessors and message-passing multicomputers. Distinguishing the two taxonomies of parallel systems lies in their implementation of memory sharing and interprocessor communication. In a shared-memory multiprocessor architecture, a shared-memory address space is commonly accessible by all processors within the system. Processors communicate with each other by modifying data objects in the shared-memory address space. In a message-passing multicomputer system, each compute node is composed of a processor and its own local memory, unshared with all other compute nodes. Compute nodes are connected with each other via a common data communication fabric or interconnection network. Interprocessor communication is accomplished by passing messages through the interconnection network.

3.2 Mercury's RACE Multicomputer

In recent years, Mercury Computer Systems, Inc. has emerged as one of the leaders in the development and manufacturing of commercially available, high-performance, embedded heterogeneous message-passing multicomputer systems. Mercury's RACE multicomputer provides a foundation for parallel systems and offers a set of building blocks that provide upward scalability. A high-level diagram of a typical RACE multicomputer is illustrated in Fig. 3.1. The system's primary components include DSPs and/or reduced-instruction-set-computing (RISC) processors, I/O ports, and a network interface all connected via the RACEway interconnection network.

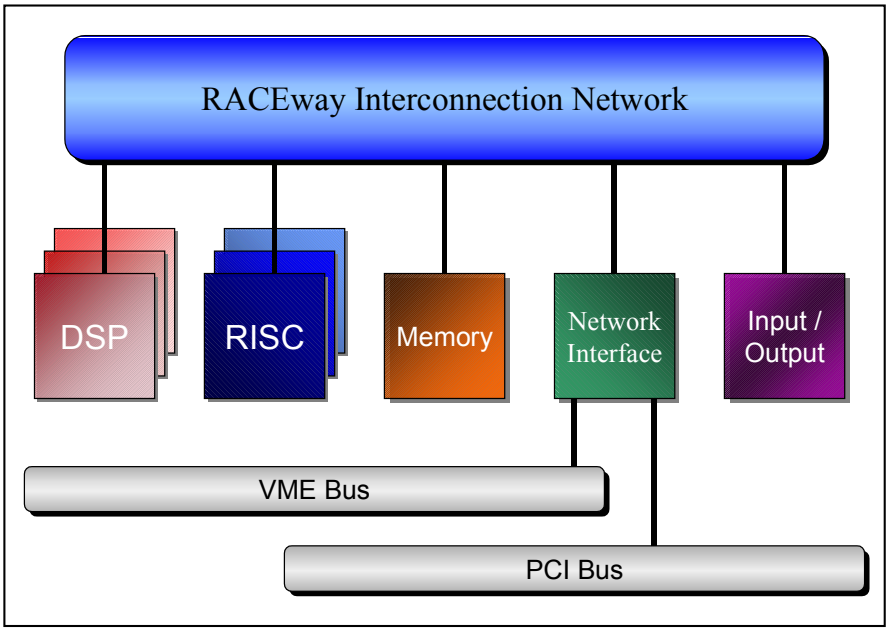

Fig. 3.1 The RACE Multicomputer (derived from [4]).

The RACEway interconnection network is used to provide high-performance communications among the interconnected processors and devices. Each node in the multicomputer interfaces the network through the RACE network chip. The network chip (see Fig. 3.2) is a crossbar with six bidirectional channels consisting of 32 parallel data lines and eight control leads [7]. Each crossbar transfers data synchronously at a clock rate

of 40-megahertz (MHz). Each channel is bidirectional but is only driven in one direction at a time at a rate of 160 megabytes per second (MB/s) [7]. Among the six ports comprising a RACE crossbar, each switch can either interconnect any three port pairs, providing and aggregate bandwidth of 480 MB/s, or can cause data to be broadcast to all or subset of the remaining five ports [4].

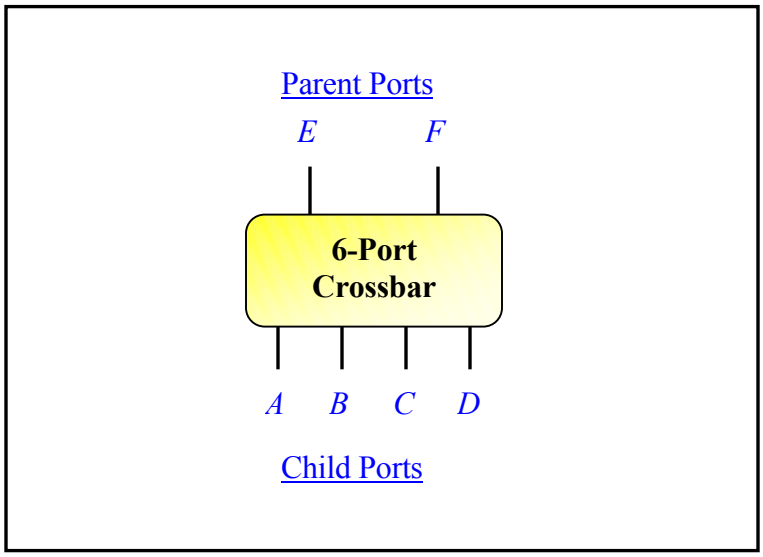

Fig. 3.2 The RACEway six-port network chip (derived from [7]).

The versatility of the RACE network chip allows the RACE multicomputer to be configured into a number of different network topologies. Possible network topologies include two-dimensional (2-D) and three-dimensional (3-D) meshes, 2-D and 3-D rings, grids, and Clos networks; however, the most common configuration is a fat-tree architecture (see Fig 3.3). For a fat-tree configuration, the crossbar switches are connected in a parent-child fashion. Each crossbar has two parent ports, *E* and *F*, and four child ports, *A*, *B*, *C*, and *D* (see Fig 3.2). The crossbars of the RACE multicomputer are connected together to form the branches of the fat-tree. The compute nodes represent the leaves of the tree.

To route a message from one processor to another, the message goes up the tree, selecting one of the two parents as it goes, until it reaches a network chip that is a common ancestor of both the source and destination node [7]. After reaching the common ancestor

network switch, the message travels down the fat tree to the destination compute node. Fig. 3.4 illustrates a message transfer from two CNs.

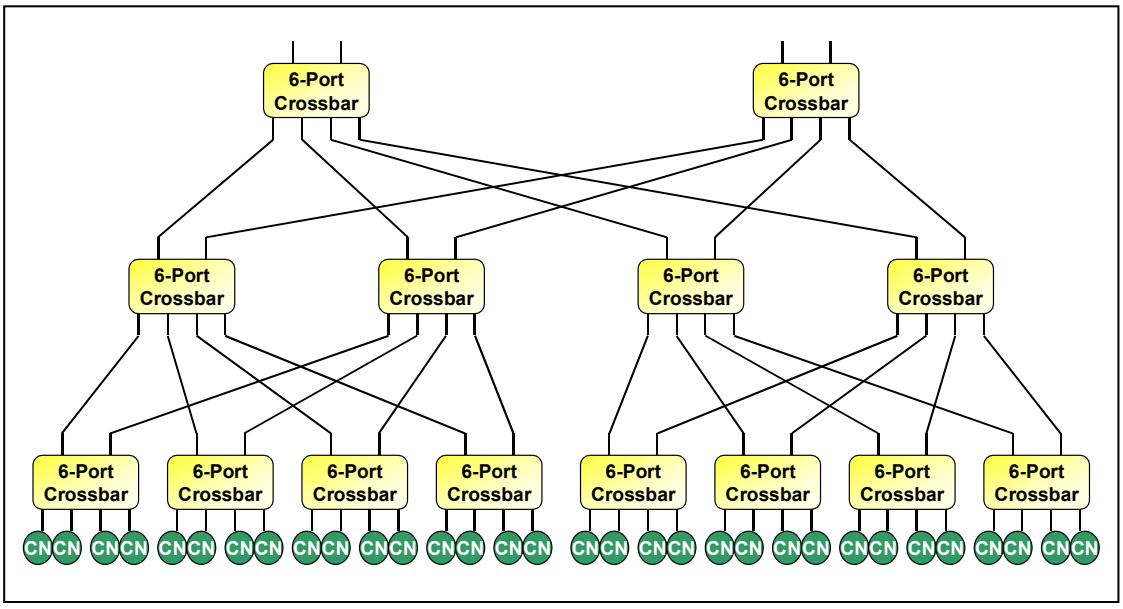

Fig. 3.3 The RACE Multicomputer fat-tree interconnection network.

In conventional tree architectures, there is only one path between any pair of processors. One major problem associated with such conventional networks is that they suffer communication bottlenecks at higher levels in the tree. For example, when several compute nodes in the left subtree communicate with compute nodes in the right subtree, the root node must handle all the messages [6]. This problem can be partially alleviated by increasing the number of effective parallel paths between compute nodes. This type of modified tree architecture is referred to as a fat-tree.

The RACE system is a circuit-switched network. In a circuit-switched network, a compute node establishes a path through the network prior to data transfer. Once the compute node has been granted a path to the destination node, the path is occupied for the duration of the data transmission. Data is transferred from one CN to another across the RACEway interconnect in packets of up to 2048 data bytes in length [20]. Each data transfer initiated by a source CN contains only a single packet consisting of up to 514 32-

bit words. The first two words of a given packet constitute the packet header. The packet header contains the information for routing the packet through the sequence of RACEway crossbars between the source and destination CN, as well as transfer control information such as the packet priority of the transfer [20]. Additionally, the destination memory address of the data transfer is included in the packet header.

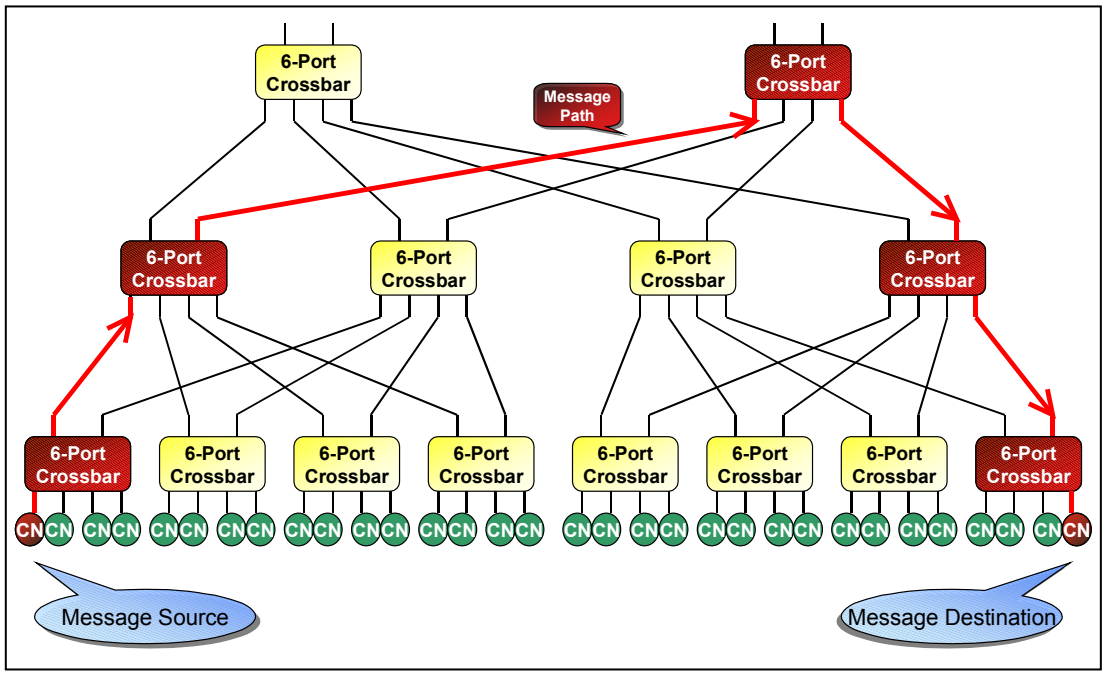

Fig. 3.4 Packet transfer between two CNs.

To send a packet through Mercury's fat-tree network, the first step is to establish a path. To establish a path, a packet header specifying a path is sent through the network along a given channel. A channel's status is categorized as either *free* or *occupied* [7]. The header makes as much progress as possible through the network until blocked. After a packet header has been blocked, it waits until a free channel becomes available. When a free channel matching the path specification (of the packet header) becomes available, the channel is flagged as occupied, and the packet header advances along that path. After establishing a path to the destination node, the packet header sends an acknowledgment to the source along the allocated path. Upon receiving the acknowledgment of a granted network path, the source node sends its packet data down the path in a pipelined fashion

[7]. During the transmission of the last byte of data, each of the occupied channel's status is set to free.

The preceding discussion of path establishment assumed that a clear and contention free path between the source and destination node existed. However, in networks that support a large number of simultaneous packet transfers, a clear path may not exist, thus contention for a crossbar port along a desired path requires arbitration. Clearly, for point-to-point transactions, a given crossbar port can only be part of one transaction per clock cycle [20]. Arbitration between two or more packets is required when the transfer paths pass through common crossbar ports.

When arbitration for a given crossbar port, or sequence of ports, becomes necessary, the arbitration is carried out on the basis of a combination of the user-programmable packet priority and a fixed hardware priority at each crossbar based on the entry and exit ports at the given crossbar [20]. Notably, the hardware priority of a given packet transaction path varies at each crossbar while the user-programmable packet priority is fixed for the duration of the packet's existence. For this work, the user-programmable packet priority is assumed equivalent for all data packets. Consequently, only the hardware priority arbitration rules associated at each crossbar will be used to resolve contention at a given crossbar port between two or more transactions.

In order to implement a fixed hardware priority at a given crossbar, each possible port-port path must be assigned a priority. In general, for an *N*-port crossbar there are $N \times (N-1)$ unique port-port pairs or possible connecting paths [20]. In this case, the RACE six-port crossbar has 30 (i.e., 6×5) possible connection paths. The assignment of the hardware priorities to each of the 30 possible paths through a RACE crossbar is complex and depends not only on the particular path, but also upon both the *status* of the contending transaction and the priority arbitration *mode* of the crossbar.

The hardware priority arbitration process performed at each crossbar to resolve contention for a given port between two or more packets depends on the following three factors. First, the directed path of a given packet through the crossbar; second, the transaction status of the two contending packets; and third, the priority arbitration mode at

the given crossbar [20]. For a more detailed description of the three hardware priority arbitration factors, the reader is referred to [20].

The assignment of hardware priorities to crossbar transaction paths is far from trivial, and the assignment of crossbar path priorities must be such as to guarantee that no *deadly embrace* conditions occur in the system [20]. A *deadly embrace* occurs when two transactions, that proceed in opposite directions along two different paths between the same two crossbars, simultaneously contend with each at both of those common crossbars, with the result that each transaction kills the other [20]. In a fat-tree architecture, the only way to prevent a *deadly embrace* situation is to implement the following rules: First, packets entering port F of any crossbar are given a different priority than those entering port E; and second, for any pair of crossbars that that can be connected via two alternative paths, the path leaving port F of one crossbar must be selected as to enter port F of the other crossbar [20]. For a more detailed analysis of the *deadly embrace* problem and solution, the reader is referred to [20].

Although the crossbars used to implement the fat-tree are physically identical, each crossbar may be configured to perform two different hardware priority arbitration algorithms [20]. The two algorithms are named Top-Level and Standard. The selection of the appropriate algorithm at a given crossbar depends upon the location of the crossbar in the network. For example, crossbars located at the top of the interconnected fat-tree are configured to implement the Top-Level algorithm while the remaining crossbars in the systems are configured to implement the Standard algorithm [20]. The reader is referred to [20] for further detail on both the Top-Level and Standard arbitration algorithms.

The priorities for hardware arbitration of crossbar port contention resolution, as a function of transaction's path through a given crossbar, are different in each transaction status case, as well as for each of the two crossbar arbitration algorithms [20]. The priorities of each of the thirty possible paths are enumerated in Fig. 3.5 and Fig. 3.6 for the case of the Standard and Top-Level algorithm, respectively, in order of priority [20]. As illustrated in the figures, there are a total of 7 different hardware priority levels, with 7 having the highest priority and 1 the lowest [20]. If two contending transactions have different hardware priority levels at a given crossbar, as defined by their respective entry

and exit ports at the crossbar and the transaction status of the exit port, the transaction having the highest hardware priority level will kill the contending lower-priority level transaction [20]. Conversely, if two or more contending transactions have the same priority level, the first one started will hold off any other contending transactions at the same level until the transmission of its data is completed [20]. The objective is to illustrate via Figures 3.5 and 3.6 the complexity of each of the two hardware priority arbitration algorithms. For a complete discussion of each algorithm, the reader is referred to [20].

	Transaction Status					
	Active		Not Yet Active			
			Port E Involved		Port E Not Involved	
Hardware Priority	Entry Port	Exit Port	Entry Port	Exit Port	Entry Port	Exit Port
7	F	A,B,C,D,E	F	A,B,C,D,E	F	A,B,C,D
6	E	F	E	F	A,B,C,D*	A,B,C,D*
5	A,B,C,D	F	A,B,C,D	F	A,B,C,D	F
4	E	A,B,C,D	E	A,B,C,D	-	-
3	*A,B,C,D	*A,B,C,D,E	A,B,C,D*	A,B,C,D*	-	-
2	-	-	A,B,C,D	E	-	-
1	-	-	-	-	-	-

* - Peer Kill Rules Apply

Fig. 3.5 Standard hardware priority arbitration algorithm [derived from 20].

	Transaction Status					
	Active		Not Yet Active		Tie	
Hardware Priority	Entry Port	Exit Port	Entry Port	Exit Port	Entry Port	Exit Port
7	F	A,B,C,D,E	F	A,B,C,D,E	F	A,B,C,D,E
6	E	A,B,C,D,F	E	A,B,C,D,F	E	A,B,C,D,F
5	A,B,C,D	A,B,C,D,E,F	A,B,C,D	A,B,C,D,E,F	D	A,B,C,E,F
4	-	-	-	-	C	A,B,D,E,F
3	-	-	-	-	B	A,C,D,E,F
2	-	-	-	-	A	B,C,D,E,F
1	-	-	-	-	-	-

* - Peer Kill Rules Apply

Fig. 3.6 Top-Level hardware priority arbitration algorithm [derived from 20].

As stated earlier in this section, the Mercury interconnection network under consideration is a fat-tree architecture comprised of multiple parallel paths. An interesting feature of the Mercury system is that it provides auto route path selection (i.e., adaptive routing) at the crossbar level, which means the multiple paths in the RACEway network may be automatically and dynamically selected by the RACE network crossbars. For instance, if one path is currently occupied with a data transfer and another path matching the path specification is free, the free path is automatically selected by the crossbar logic [10]. Adaptive routing is used to adaptively route packets that enter on either of the four child ports and exits either of the two parent ports. Auto route path selection frees the programmer from the details of path routing. Additionally, applications that require tremendous interprocessor communication such as distributed matrix transpose and corner turns often benefit from adaptive routing [4].

With the network configured as a fat-tree, the RACEway interconnection fabric provides very good scaling properties. In an p-processor system, the height of the fat-tree is $h = \lceil \log_4 p \rceil$. Thus, the network diameter or maximum number of links traversed is $D = 2h - 1$. The bisection bandwidth of a system, which is defined as the minimum number of edges (or channels) that have to be removed along a cut that partitions the network into two equal halves, assuming $p = 4^i$ processors, is $B = 160\sqrt{p}$ MB/s [7]. (Each channel in the RACEway system has a bandwidth of 160 MB/s.)

The RACE system may be configured as a heterogeneous multicomputer composed of two or more different types of processors. The potential heterogeneity of the RACE multicomputer includes various possible configurations of i860, PowerPC, and Super Harvard Architecture Computer (SHARC) DSP processors. The SHARC DSP is ideally suited for embedded vector signal processing applications such as Fast Fourier Transforms (FFTs) where physical size and power are at a premium or other similar algorithms that have a high ratio of data-to-computation. Furthermore, the Analog Devices' SHARC processor enables more than twice the physical processor density of reduced instruction set computer (RISC) based CNs. In contrast, the PowerPC and i860, both RICS processors,

are appropriate for executing scalar-type applications, with a low ratio of data-to-computation, generated by arbitrary compiled code.

The CNs in Figs. 3.3 and 3.4 are composed of three basic components: one to three processors (all of the same type), 8 to 64 MBs of dynamic random access memory (DRAM), and a Mercury-designed application specific integrated circuit (ASIC). Each ASIC is composed of address mapping logic, a direct memory access controller (DMA), processor support functions such as timers, and interfacing logic for effective RACEway transfers [4]. The address mapping logic enables local CN access to any DRAM location in any remotely located CN on the network [4]. The DMA engine provides a mechanism for rapid block-transfers between DRAM and other CNs, input/output (I/O) devices, or bridges nodes on the network. There is a unique CN ASIC for each CN processor type.

Because partially adaptive STAP is a signal processing application characterized by a high ratio of data-to-computation, the work to be completed will focus on the use of SHARC CNs. The composition of SHARC CNs includes one to three SHARC processors sharing a common DRAM and ASIC interface (see Fig 3.7). Within a CN, multiple SHARC processors are connected via a common 32-bit bus.

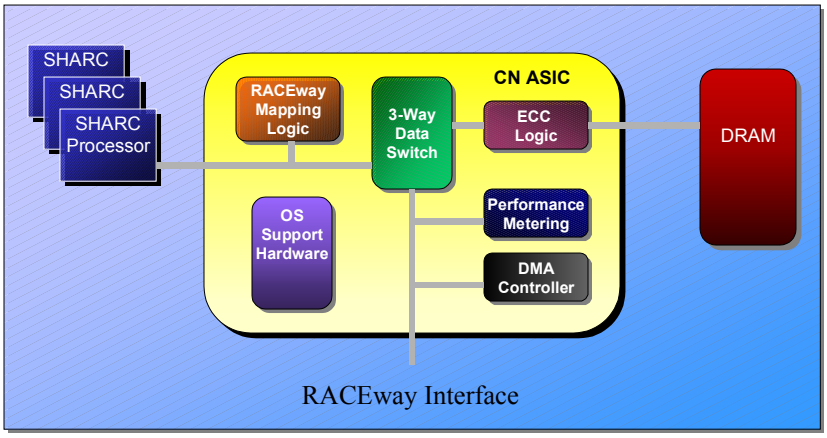

Fig. 3.7 SHARC compute node (derived from [4]).

CHAPTER IV

A PARALLELIZATION APPROACH FOR STAP

STAP refers to a class of signal processing methods that operate on a set of radar returns gathered from a set of array channels over a specified time interval. STAP is inherently three-dimensional (i.e., range, pulse, and channel), because the signal returns are composed of range, pulse, and antenna-element digital samples. Thus, a three-dimensional (3-D) data cube naturally represents STAP data. Typical processing requirements for STAP data cubes range from 10-100 Gflops, which can only be met by multicomputer systems composed of numerous interconnected CNs [6]. Imposed real-time deadlines for STAP processing restricts processing to parallel computers.

Developing a solution to any problem on a parallel system is generally not a trivial task. A major challenge of implementing STAP algorithms on multiprocessor systems is determining the best method for distributing the 3-D data set across CEs of a multiprocessor system (i.e., the mapping strategy) and the scheduling of communication within each phase of computation. Generally, STAP comprises three phases of processing, one for each dimension of the data cube. During each phase, the vectors of data along each dimension are distributed among the CNs for processing in parallel. During the processing for each dimension, the entire vector of data along the dimension of interest must reside in local memory at each CN. Additionally, each CN may be responsible for processing one or more vectors of data during each phase.

This re-distribution of data or distributed "corner-turn" requires interprocessor communication. Minimizing the time required for interprocessor communication helps maximize STAP efficiency. To assist in the minimization of interprocessor communication time during the data re-distribution phases, a paradigm for distributing the 3-D STAP data set among CNs of a multicomputer system has been proposed in [13]. Sections 4.1 and 4.2 summarize the work found in [13].

4.1 Data Set Partitioning by Planes

At each of the three phases of processing, data access is either vector oriented along a data cube dimension, or a plane-oriented combination of two data cube dimensions. Figure 4.1 illustrates the STAP flow. The three phases of processing include pulse compression, Doppler filtering, and beam weight computation and beam formation. During the first phase, pulse compression, the range dimension is processed. Next, the data cube is corner-turned to process data vectors along the pulse dimension termed Doppler filtering. After a second corner-turn, beam weight computation is performed by implementing a QR decomposition on a data matrix composed of samples from a combination of the range and channel dimensions. Finally, beam formation processing occurs along the contiguous vectors in the channel dimension.

The primary goals of many parallel applications are to reduce latency and minimize interprocessor communication (IPC) while maximizing throughput. It is indeed necessary to accomplish these objectives in STAP environments. To reduce latency, the processing at each stage must be distributed over multiple CNs in a single program multiple data (SPMD) approach. (In a SPMD approach, each CN executes the same program asynchronously.) However, prior to each processing phase, the data set must be partitioned in a fashion that attempts to equally distribute the computational load over the CNs. Furthermore, because each phase processes a different dimension of the data cube, the data cube must be re-distributed in a manner that minimizes IPC.

One approach to data set partitioning is to distribute the data set by data planes (see Fig 4.2). Each data plane is composed of two entire dimensions of the STAP data cube (and one or more elements of the third dimension). For this approach, the number of processors over which the data planes may be distributed is limited to the smallest dimension of the data cube. Shown in Fig. 4.2 is a decomposition of the N planes, one for each pulse. Data re-partitioning requires IPC between all N processors, which requires approximately N^2 data transfers.

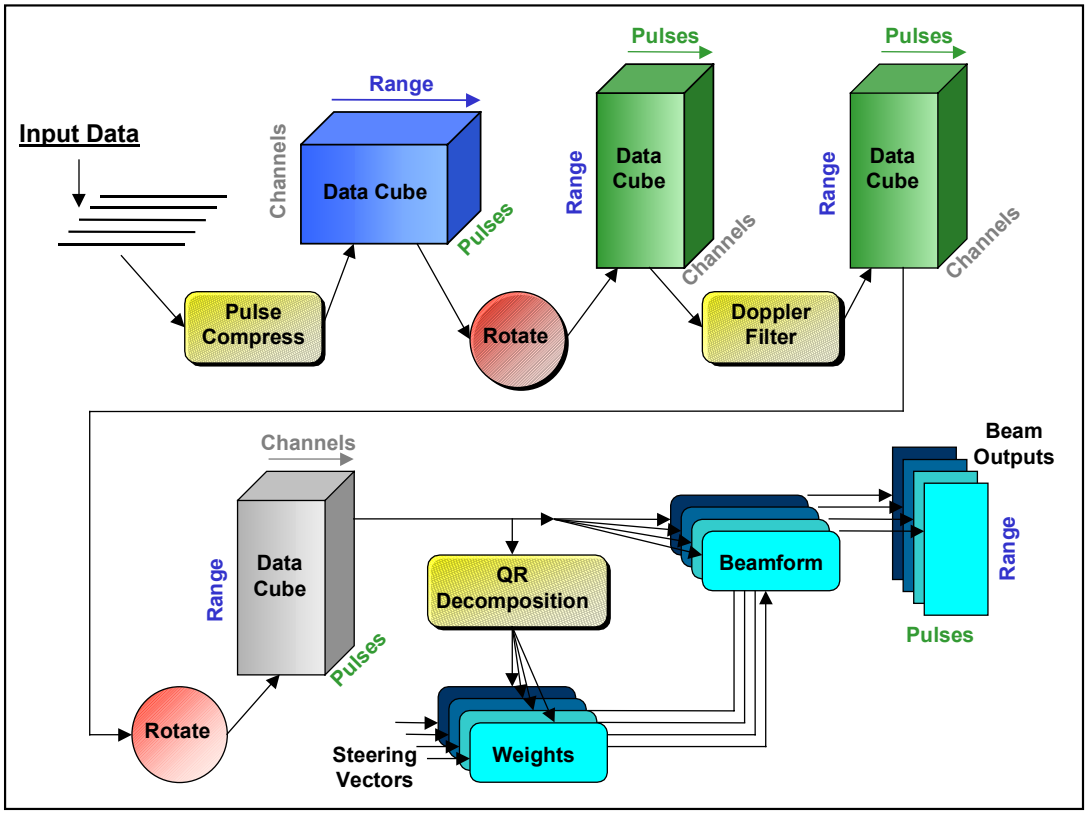

Fig. 4.1 Block diagram illustration of STAP flow (derived from [13]).

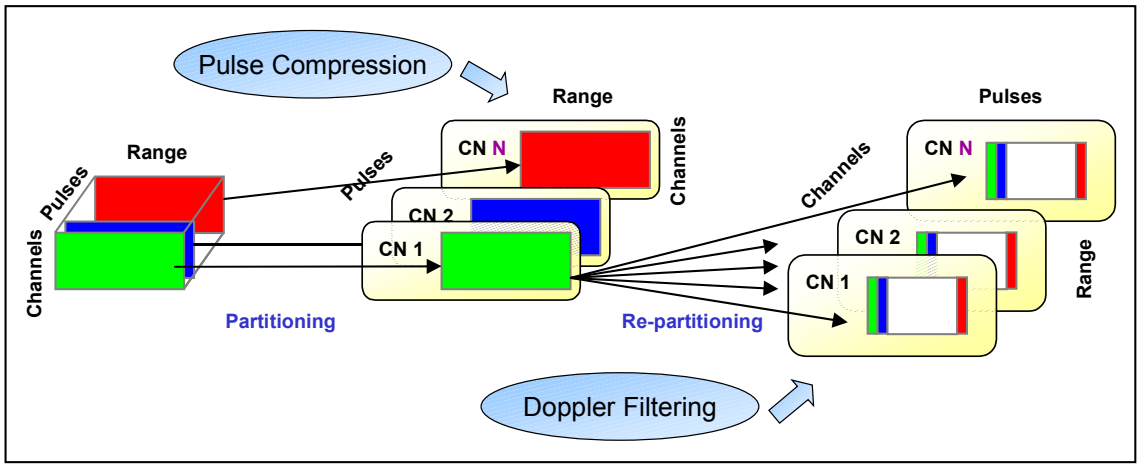

Fig. 4.2 STAP data cube partitioning by data planes (derived from [13]).

4.2 Data Set Partitioning by Sub-Cube Bars

A second approach to data set distributing in STAP applications is to partition the data cube into sub-cube bars. Each sub-cube bar is composed of partial data samples from two dimensions while persevering one whole dimension of the data cube as shown in Fig. 4.3. The maximum number of processors over which the data set may be partitioned is equal to the product of the two smallest dimensions of the data cube.

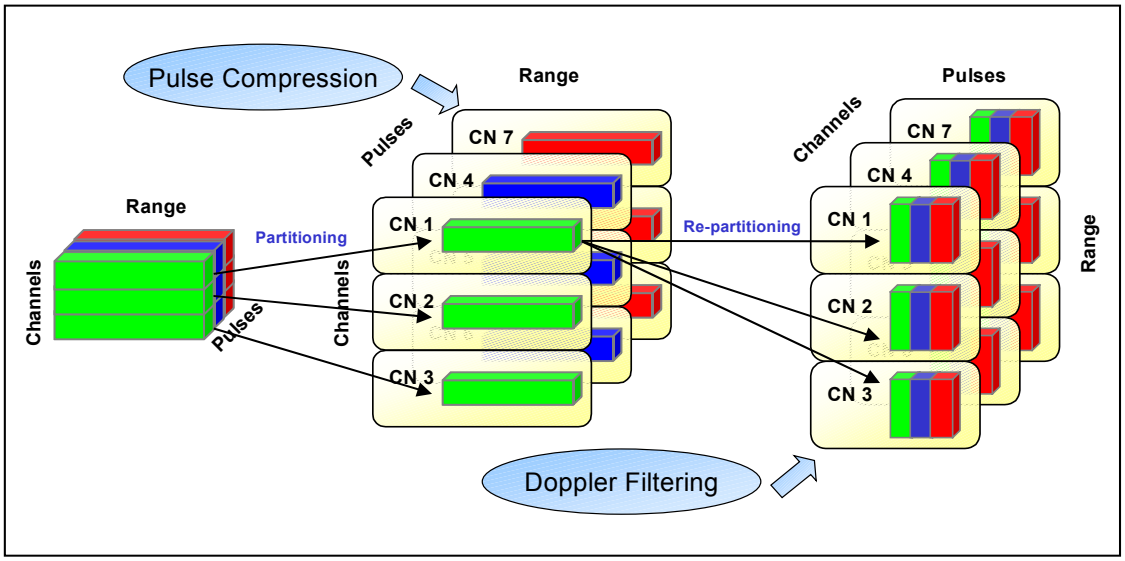

Fig. 4.3 STAP data cube partitioning by sub-cube bars (derived from [13]).

The authors of [13] proposed a five step methodology for effectively distributing and partitioning the STAP data cube across multiple processors. The first step is to partition the data cube over a two-dimensional *process set*. A process set is defined as a logical grouping of processes that can share data and synchronize with each other and other process sets. Data set partitioning is accomplished by dividing the dimensions of the data set by the dimensions of the process set. For example, in phase one of STAP, pulse compression is performed along the range data vectors. By applying a 3×4 process set to the STAP data cube, leaving the range dimension intact, the channel and pulse dimensions

become segmented (see Fig 4.4). Implementing this partitioning scheme for the first phase would require twelve processors.

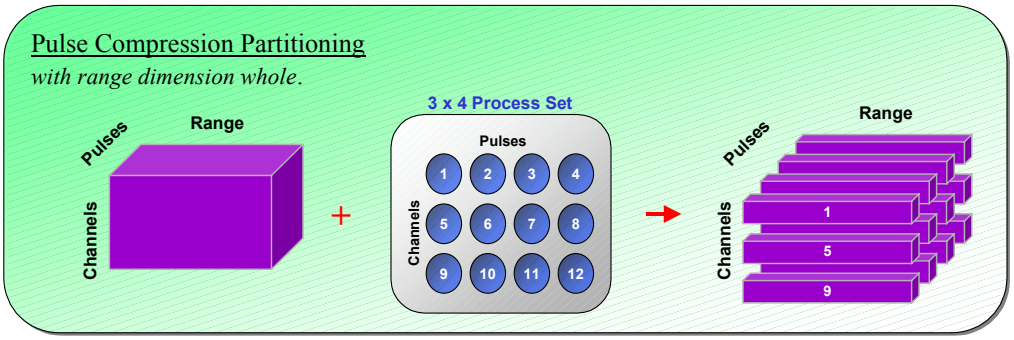

Fig. 4.4 Sub-cube bar partitioning prior to pulse compression (derived from [13]).

Applying the same 3×4 process set to the Doppler filtering phase, results in the segmentation of both the channel and range dimensions (see Fig. 4.5). In this phase, the pulse data vectors include every pulse entry for a given array channel and range.

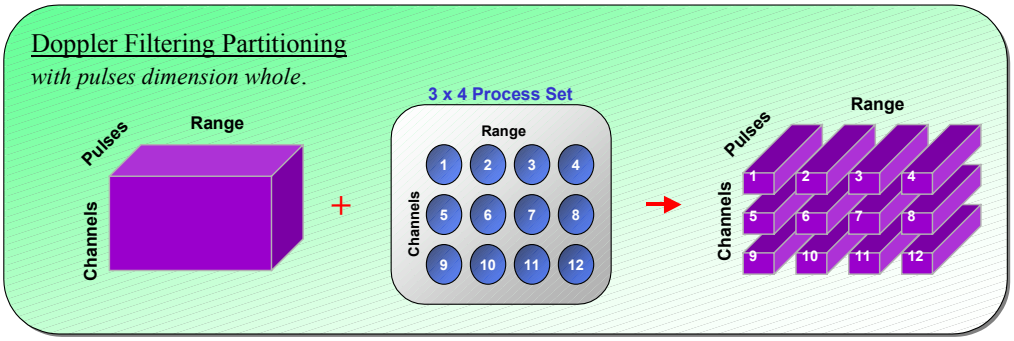

Fig. 4.5 Sub-cube bar partitioning prior to Doppler filtering (derived from [13]).

The second step involves processing in parallel the current whole dimension of the data cube (e.g., pulse compression, Doppler filtering, QR Decomposition). After performing the necessary calculations based on the current whole dimension, the third step entails re-partitioning the data before processing the next dimension. To re-partition the data, the current whole dimension must be exchanged with the next whole dimension (see

Fig. 4.6). As illustrated in Fig. 4.7, the required data exchange occurs only between processors in the same row. For example, process 1 transfers data to process 2, 3, and 4 while process 5 distributes data to process 6, 7, and 8. During this procedure, multiple phases of communication may take place in parallel. Assuming a P-processor systems, data set re-partitioning would require approximately $\sqrt{P}$ sets of P data transfers.

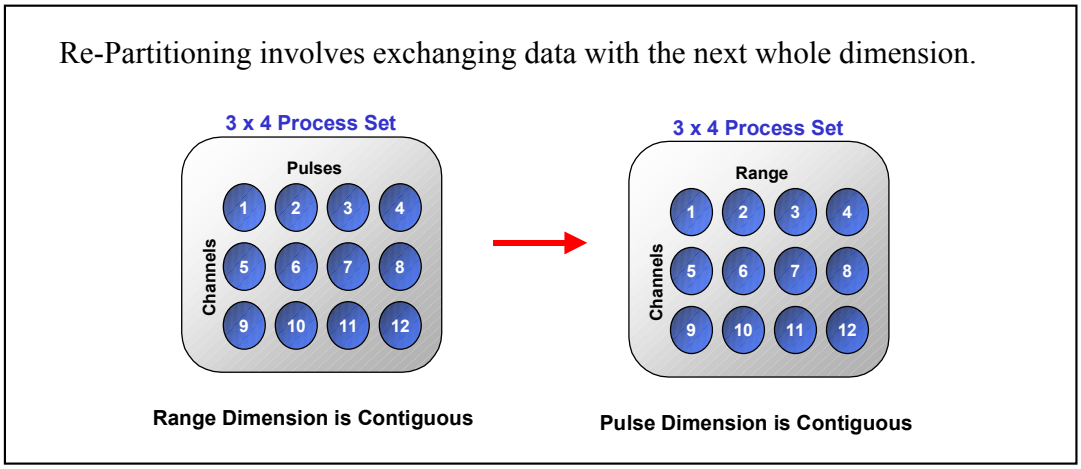

Fig. 4.6 Process set re-partitioning prior to Doppler filtering (derived from [13]).

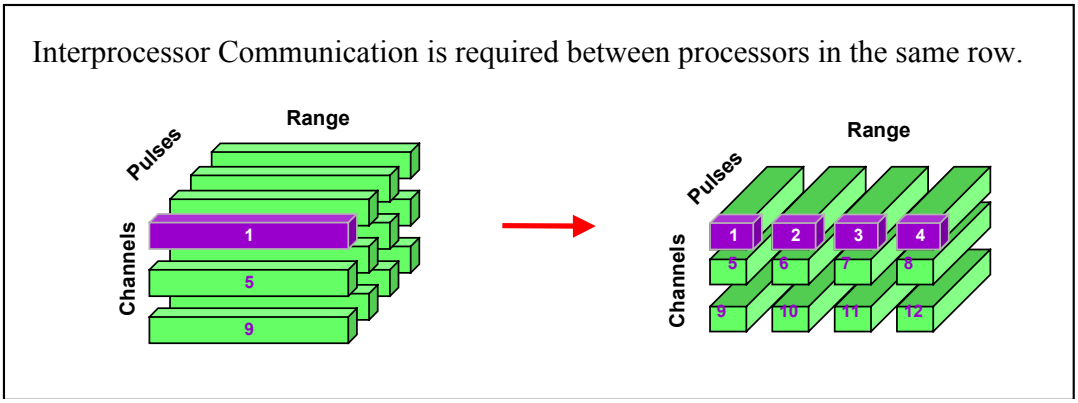

Fig. 4.7 STAP data cube re-partitioning prior to Doppler filtering (derived from [13]).

After re-partitioning the data set, completing step four involves sequentially ordering the data set in memory prior to processing. This local ordering of data, know as data set rotation, does not alter the dimension assigned to the process set nor require any IPC. It

does, however, require a rotation local to each processor. The final step is to repeat each of the first four steps on each dimension of the STAP data cube. Fig 4.8 illustrates STAP processing using sub-cube bar partitioning.

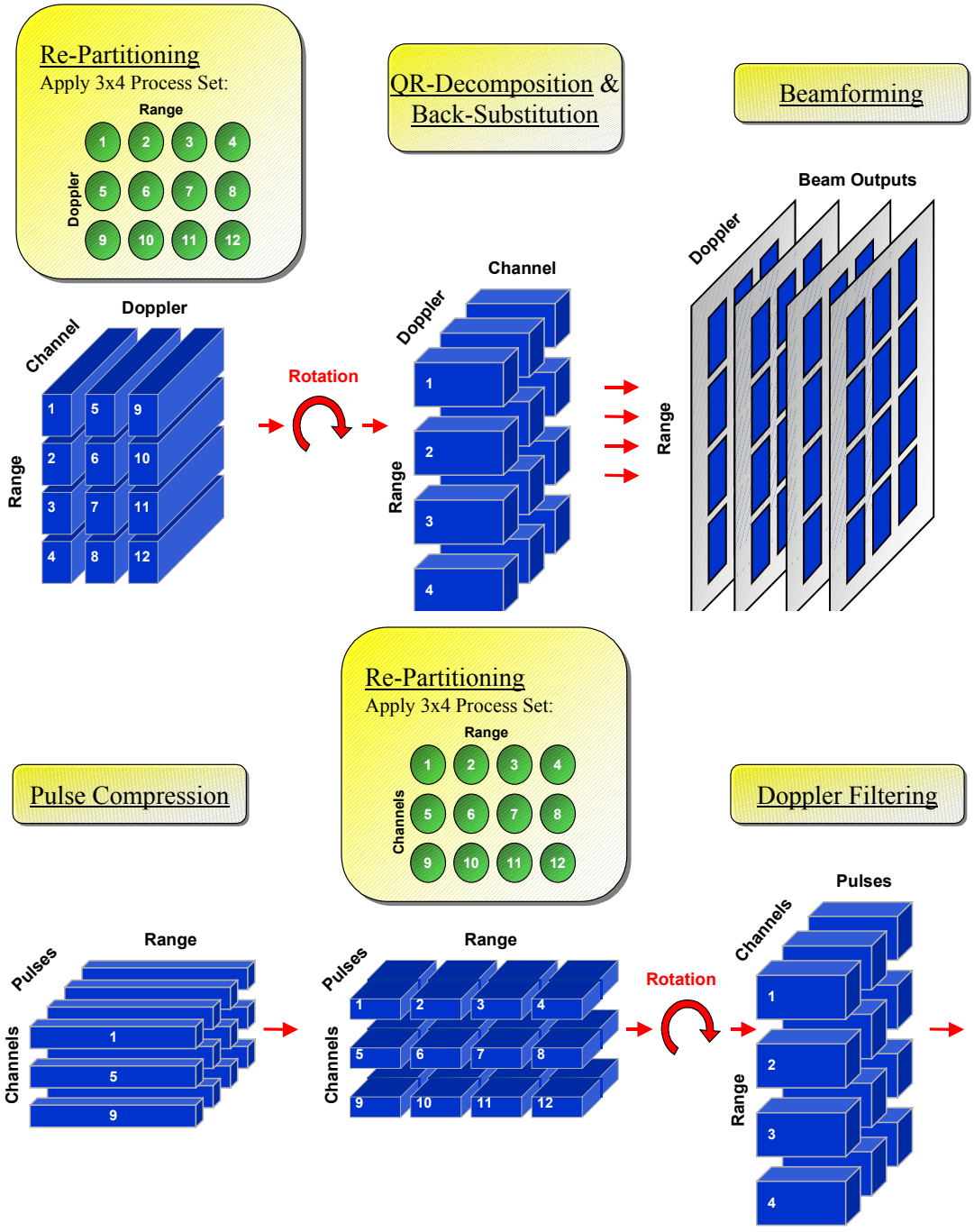

Fig. 4.8 STAP processing using sub-cube bar partitioning (derived from [13]).

4.3 Comparison of Data Plane versus Sub-Cube Bar Partitioning

Partitioning the STAP data cube along data planes has one potential advantage over sub-cube bar partitioning. If the initial data plane partitioning is performed along the channel dimension (leaving the range and pulse contiguous), both range processing and Doppler filtering may be performed without a re-partitioning phase taking place between the two steps. By implementing this scheme, only one re-partitioning step needs to take place, which occurs after Doppler filtering. Unfortunately, data plane partitioning has several disadvantages. Partitioning the data set along the smallest dimension or the channel dimension greatly reduces the number of potential processors as compared to sub-cube bar partitioning. Because the number of available processors is smaller, each processor in a data plane partitioning scheme is allocated a larger chunk of the data cube. Besides increasing the local memory requirement at each node, larger data segments demand more processing time thus increasing total completion time. Additionally, data set re-partitioning requires IPC between all processors.

In contrast, sub-cube bar partitioning of the data set provides a method whereby potential gains can be made in scalability and performance while minimizing the IPC time. Also, sub-cube bar partitioning has the potential for finer-grained parallelism because the maximum number of processors over which that data may be divided is the product of the two smallest dimensions of the data cube. Typically, the number of processors allocated to solve a sub-cube bar partitioned data cube is greater than in a data plane partitioning approach. Consequently, each processor performs fewer computations resulting in a shorter completion time. Furthermore, IPC between processing stages is isolated to only clusters of processors and not the entire system. On the other hand, sub-cube bar partition has a few disadvantages. First, this approach requires two separate re-partition and rotation phases. Secondly, scheduling data transfers during the re-partition phase is more complicated because communication is confined to groups of processors. The proposed research is to model, through simulation, the timing effects associated with how data is mapped onto the CNs and how the data transfers are scheduled.

CHAPTER V

MAPPING DATA AND SCHEDULING COMMUNICATIONS FOR IMPROVED PERFORMANCE

The overall performance of parallel computer systems can be highly dependent upon network contention. In general, the mapping of data and the scheduling of communication impacts network contention of parallel architectures. During phases of data re-distribution on parallel computers, the number of required communications is vastly impacted by the current location and future destination of the data. Determining the optimal schedule of data transfers through interconnection networks is generally an NP-hard problem [5]. Consequently, heuristics are often used to provide sub-optimal solutions. A combination of these two factors, data mapping and communication scheduling, provides the key motivation for the network simulator described in Chapter VI. The following two sections illustrate the importance of data mapping and the scheduling of communications issues that exist implementing a sub-cube bar partitioning scheme on STAP data cubes.

5.1 Mapping a STAP Data Cube onto the Mercury RACE System

As described in Section 4.2, data set partitioning by sub-cube bars entails partitioning the data cube into bars composed of two partial dimensions and one whole dimension. After partitioning, the sub-cube bars are distributed over a two-dimensional set of processors. Partitioning the STAP data cube across the Mercury System consists of an increased level of complexity because each CN is composed of three SHARC processors (i.e., CEs). (In general, a CN can contain one, two, or three CEs; three CEs are assumed here.) An important issue is how to map the sub-cube bars onto the available CNs on the Mercury System. To illustrate the impact of mapping, consider the examples of sub-cube bar partitioning prior to pulse compression in Fig. 5.1. For this example, assume that the Mercury System is composed of twelve CEs (see Fig. 5.2), and the STAP data cube is divided into twelve sub-cube bars. Additionally, the number on each sub-cube bar

indicates the CE to which the bar is assigned for processing. The left-hand portion of the figure illustrates a mapping scheme where the sub-cube bars are raster-numbered along the pulse dimension. In contrast, the right-hand portion of the figure depicts a mapping scheme whereby the sub-cube bars are raster-numbered along the channel dimension. The coloring code of each bar corresponds to its destination CN (recall that each CN is assumed to consist of 3 CEs). For instance, the three blue sub-cube bars are assigned to the blue CN, while the red CN processes the three red sub-cube bars.

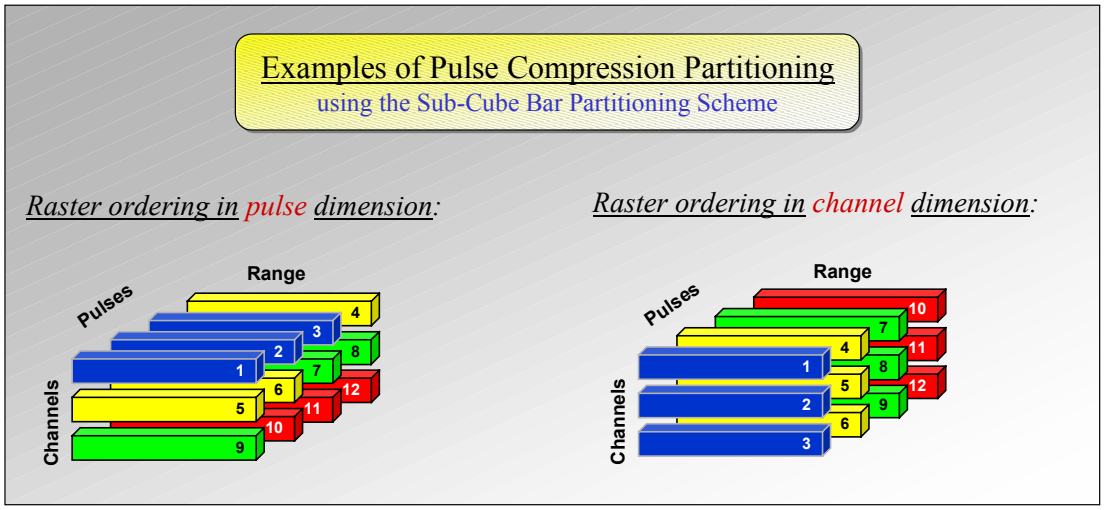

Fig. 5.1 Examples of sub-cube bar mapping schemes prior to Doppler filtering.

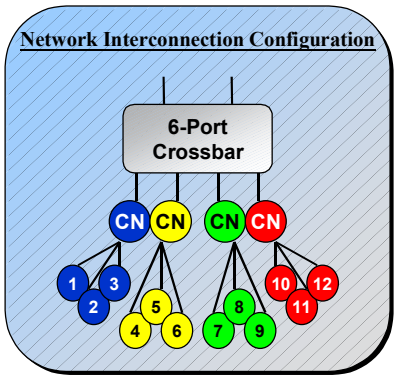

Fig. 5.2 An example configuration of a four CN (twelve CE) Mercury System.

After performing pulse compression on the data samples along the range dimension, the data set requires re-partitioning prior to Doppler filtering. The initial mapping of the data prior to pulse compression affects the number of required communications during the re-partitioning phase. In the case where the data cube is raster-numbered along the pulse dimension, six messages, totaling 20 units in size, must be transferred through the interconnection network (see Fig. 5.3).

Each CN is assumed to be configured as one master CE and two slave CEs. The master CE allocates the entire shared memory pool and distributes memory to the other two CEs. Having a master CE on each CN is advantageous during data re-partitioning phases. When two or more CEs within the same CN have one or more messages to send a common destination CN, the messages may be combined into one message by the master CE's direct memory access (DMA) controller. The newly created message may now be transferred to the destination node by the CN ASIC DMA controller. For increased efficiency, message transfers should be performed by the CN ASIC DMA controller while the CEs are concurrently processing. The reversal of this same process may be applied to message arriving at a CN. Messages arriving at a CN may be composed of one or more messages sent to one or more of its CEs. After receiving the message from the CN ASIC DMA controller, the master CE distributes the message's contents to the appropriate memory location. For this example, the blue CN (or master CE labeled 1) transfers data of size three units to the yellow CN. Furthermore, the yellow CN needs to transfer two messages, one to the blue CN of size three units and one to the green CN of size four units.

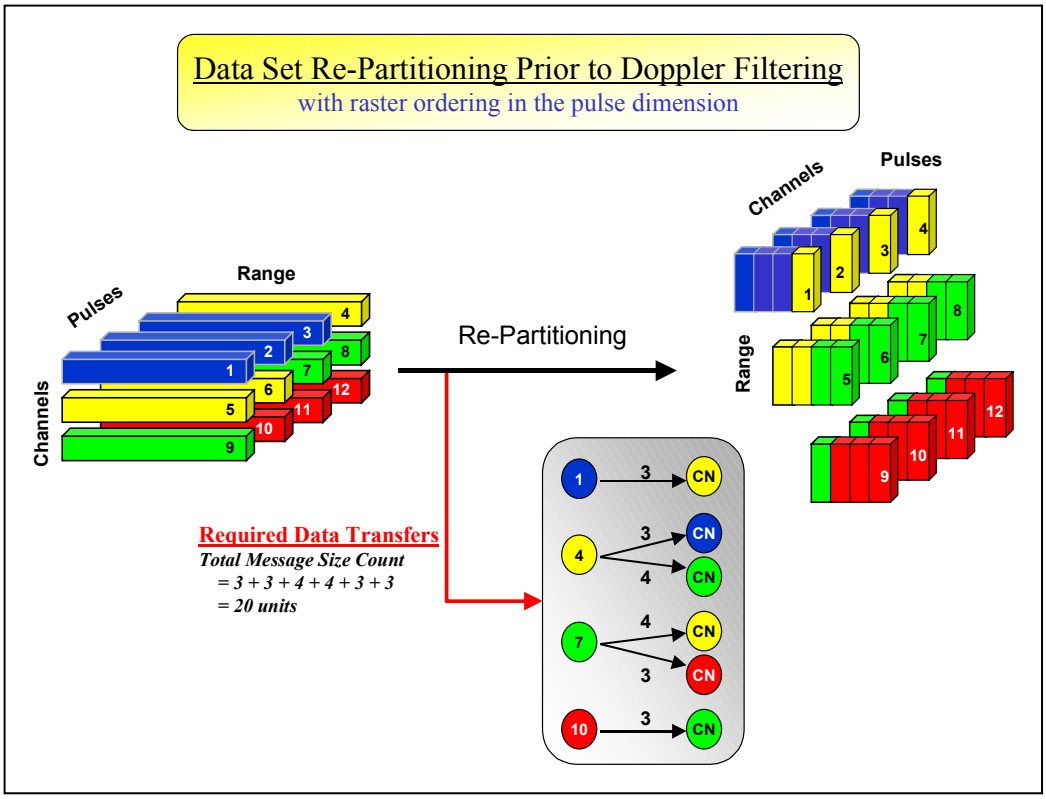

Fig. 5.3 Data set re-partitioning with raster numbering along the pulse dimension.

The second example in Fig. 5.1 shows the data cube raster-numbered along the channel dimension. Implementing this mapping scheme drastically increases the number of messages that must be communicated during the re-partitioning phase prior to Doppler filtering (see Fig. 5.4). The number of required data transfers increases from six to twelve, and the total message count also increases from 20 to 36 units. In conclusion, raster-numbering along the pulse dimension provides a smaller communication overhead than raster-numbering along the channel dimension for this example and for this communication phase.

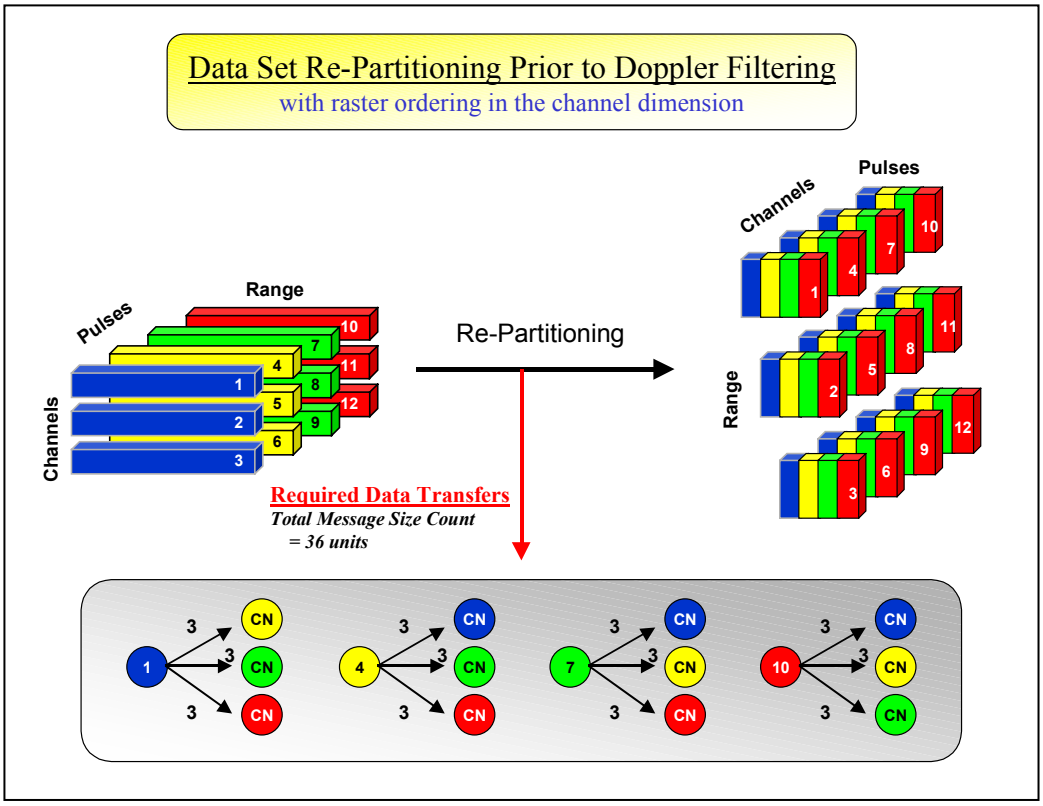

Fig. 5.4 Data set re-partitioning with raster-numbering along the channel dimension.

5.2 Scheduling Communications During Re-Partitioning Phases

After processing has been completed on the current whole dimension of the STAP data set, each master CE forms the outgoing messages necessary to replace the current whole dimension with the next processing dimension. Once constructed, the outgoing messages are placed in a queue and await transfer to their corresponding destination CNs. Determining the optimal communication schedule (i.e., ordering) of queued messages in circuit switched networks is a formidable task. Problems of this nature are generally proven to be NP-hard problems.

To illustrate the impact of message scheduling communications during data re-partition phases in partially adaptive STAP algorithms, consider the re-partitioning problem in Fig. 5.3. In this example, the re-partitioning phase involves transferring six messages

through the interconnection network. If the six messages were sequentially communicated (i.e., no parallel communication) through the network, the completion time (T_c) would be the sum of the length of each message, which totals 20 network cycles. If two or more of the messages could be sent through the network concurrently, then the value of T_c would be reduced (i.e., below 20). The purpose here is to illustrate that the order (i.e., the schedule) in which the messages are queued for transmission can impact how much (if any) concurrent communication can occur. Thus, it will be shown that scheduling choices affect the value of T_c.

An illustration of the required data transfers is depicted in Fig. 5.5. The left-hand portion of the figure shows the current location of the STAP data cube on the given CEs after pulse compression. The coloring scheme of each sub-cube bar indicates the destination CN for the next phase of processing. If part or all of the sub-cube bar is a different color than its current processor color, the data must be transferred to the corresponding colored destination node. The data cube on the right-hand of the figure illustrates the sub-cube bars of the STAP data cube after re-partitioning. Each of the sub-cube bars is composed of sample data of the whole pulse dimension, thus each sub-cube bar is a single color. After re-partitioning the data, Doppler filtering may be applied to each sub-cube bar in parallel.

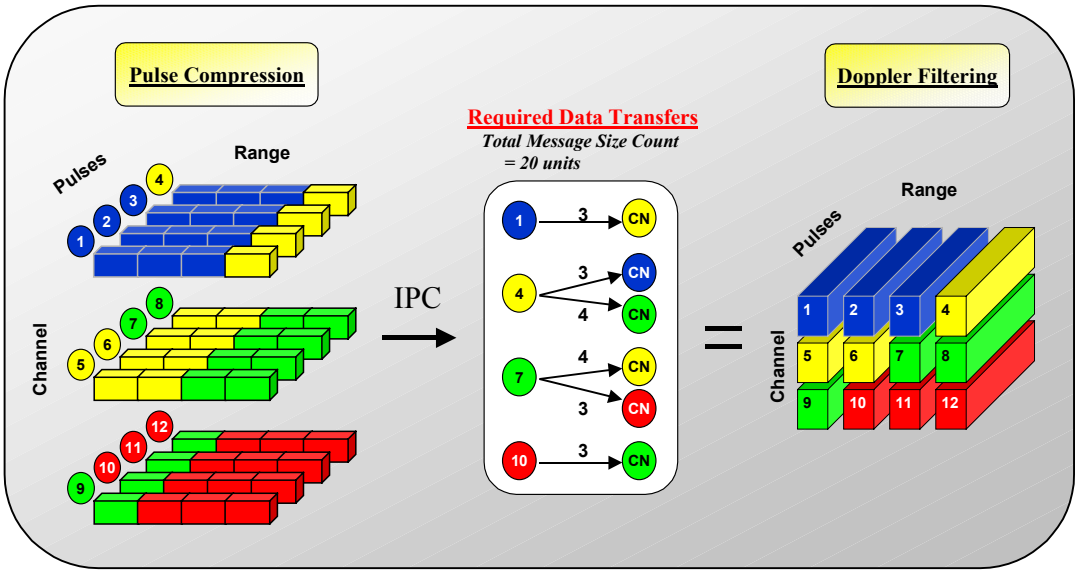

Fig. 5.5 An example of data set re-partitioning prior to Doppler filtering.

Scheduling the communication of each of the six messages through the RACEway network greatly affects the overall performance of the system. Fig. 5.6 shows the six messages, labeled A through F, in the outgoing message first-in-first-out (FIFO) queues of the source CNs. Each message's destination is indicated by its color. For instance, message A, which is colored yellow, is destined for the yellow CN. The destination of the blue message B is the blue CN, and so forth. The number in parentheses by each message label represents the relative size of the message. For example, message A's size is three units. Because of the assumed queues' FIFO implementation, message B must be transmitted before message E on the yellow CN.

The minimum achievable communication time is dependent upon the CN with the largest communication time to send/receive all outgoing and incoming messages. As shown in Fig. 5.6, the minimum possible communication time is the sum of all outgoing and incoming messages on the yellow or green CN, which equals fourteen units. Therefore, any scheduling for the six messages can complete in no less than fourteen

message units. The actual communication time, T_c, that would result for this example with the given message queue orderings (i.e., scheduling) is 17 network units.

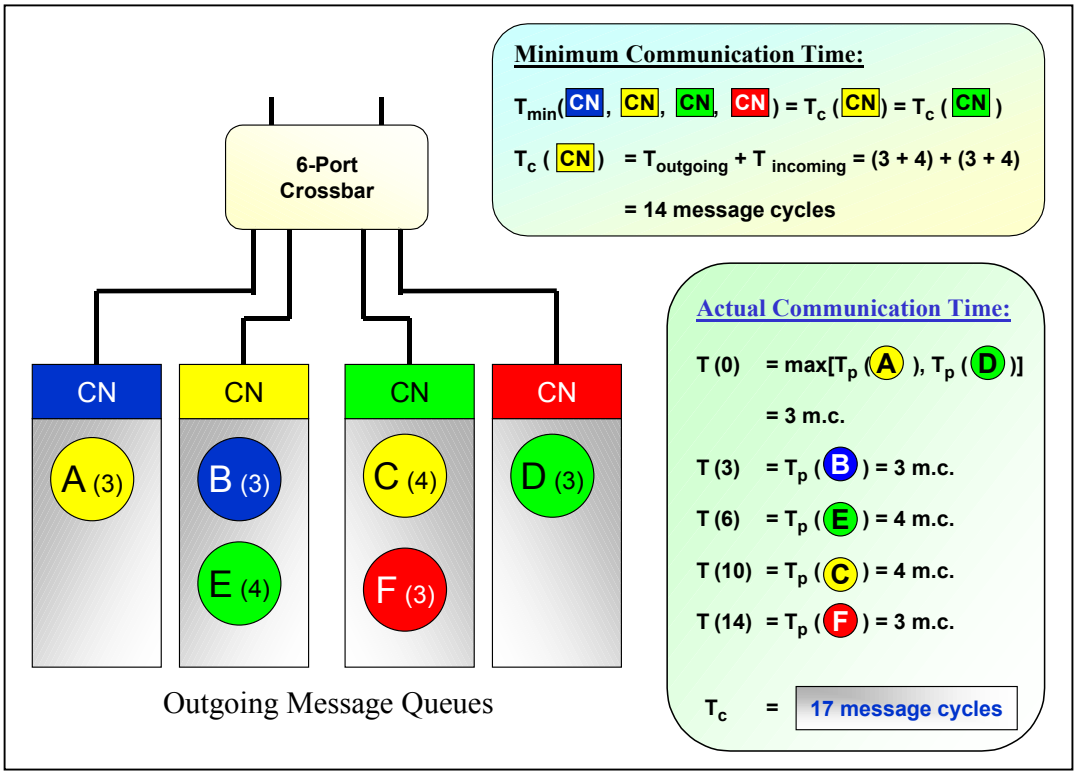

Fig. 5.6 A sub-optimal communication scheduling example.

To understand why the completion time for this example is 17 network cycles, note that at time $t = 0$, the first messages in the queue, messages A, B, C, and D, are ready for transmission. In parallel, each CN sends its first message to the six-port crossbar. All four of the incoming messages arrive at the crossbar simultaneous with each message requesting an outgoing channel. The resolution scheme based on port number resolves the network contention conflict. (In this example, each message is assumed to have the same priority.) As stated in Section 3.2, the port number is used as a tie-breaking mechanism for messages with the same priority contending for the same channel (lowest port number goes first). In this example, the lowest port number is associate with the left-most CN, and the highest port number is associated with the right-most CN (i.e., the four "children" ports of the

crossbar are numbered 0 to 3). The crossbar resolves the network contention problem by scanning the child ports from left to right. By using this contention resolution approach, message A is granted access to the link connected to the yellow CN, while message B waits in the queue. Furthermore, message D seizes the link to the green compute node, because message C is unable to establish a link to the yellow CN, which is occupied by message A.

At time $t = 3$, messages A and D complete and release the four channels occupied. After the status of the four occupied channels are set to free, queued messages B and C request links through the crossbar to their respective destination nodes. Because messages B and C both require access to the link connecting the yellow CN to the network, only message B, with a lower port number than C, establishes a path through the crossbar at time $t = 3$. After message B finishes transfer at time $t = 6$, queued messages E and C query the crossbar for a free path to their destination nodes. In this case, messages E and C are contending for the same two channels resulting in a sequential transfer of the two messages. Based on the port numbers, message C follows message E. Furthermore, messages C and F require sequential transfer because they both originate at the same CN. As a result, the remaining three messages are transferred sequential (i.e., no parallel communication) in an E $\rightarrow$ C $\rightarrow$ F ordering, totaling eleven network cycles. Compared to the minimum possible communication time of fourteen message cycles, the above message scheduling example renders a sub-optimal completion time, T_c, of 17 message cycles. However, changing the ordering of the messages in the outgoing queues, as described below, will yield an optimal scheduling of the messages.

The message queues in Fig. 5.7 are identical to those in Fig. 5.6 except messages C and F have swapped positions on the green CN. Swapping the ordering of the messages on the green compute node allows for an increase in the number of messages that can be communicated in parallel. To understand how the ordering yields improved performance, note that at time $t = 0$, the first messages in the queue, messages A, B, F, and D, are ready for transmission. In parallel, each CN sends its first message to the six-port crossbar. As before, all four of the incoming messages arrive at the crossbar simultaneous with each message requesting an outgoing channel. The crossbar resolves the network contention problem by scanning the port numbers in order, which allows messages A and F to

establish communication links through the network to their respective destination locations. Messages A and F are transferred in parallel and complete in three network cycles.

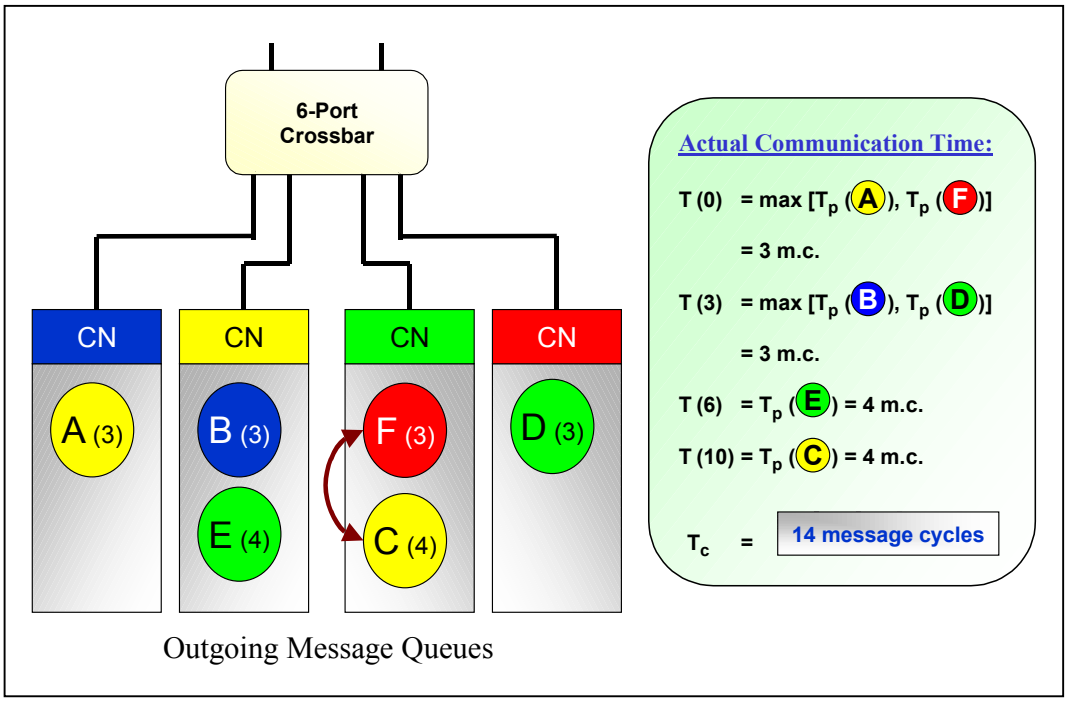

Fig. 5.7 An optimal communication scheduling example.

At time $t = 3$, queued messages B, C, and D request communication paths through the six-port crossbar, but only messages B and D are granted network access. Message C remains in the queue because messages B and C are contending for the same link (the link connecting the yellow CN to the network), and message B gains access to the channel because it originates from a lower port number. After messages B and D complete their transmission at time $t = 6$, the last two messages, E and C, query the crossbar for path establishment. In this case, messages E and C are competing for the same two channels. Consequently, the remaining two messages are transmitted sequentially, totaling eight network cycles, with E preceding C based on port number priority resolution. For this scheduling example, the actual completion time, T_c, achieves the optimal completion time of fourteen network cycles.

CHAPTER VI

DESIGN OF THE SIMULATOR

The goal of the simulator's design was to produce an accurate model or representation of the proposed system that could be implemented. As a broader understanding of the underlying details of the Mercury System was developed, the design of the simulator was further refined and modularized. The Unified Modeling Language (UML), a third generation object-oriented modeling language, was utilized to formalize the simulator's requirements in software terms [21]. UML is a language for specifying, visualizing, and constructing the artifacts of complex software systems. It simplifies the complex process of software design and provides a blueprint for construction [21]. The primary goals of UML are as follows: (1) provide an expressive visual modeling language to develop meaningful models; (2) provide specialization mechanisms to extend core concepts; (3) be independent of programming languages and development processes; (4) provide a basis for formal modeling languages; (5) encourage the growth of object-oriented tools; and, (6) integrate the best practices [21].

6.1 UML Class Definitions

The design of the simulator incorporates all the underlying components of the Mercury System. Within the structure of the design, only one definition of a class exists in the model; however, it may appear on several class diagrams. An important aspect of the simulator's object oriented design is modularity. By separating the functional components of the system into classes, the classes and their operations provide inherent modularity as well as information hiding.

The class diagram is one of the core components to a UML model. A class diagram illustrates the important abstractions in the system including relationships. The primary elements included on a class diagram are class icons and relationship icons. Fig. 6.1 shows a suppressed UML class diagram of the network simulator. The rectangular boxes represent the classes, while the lines connecting the classes signify the relationships. A

solid line with a hollow diamond at one end indicates an aggregation relation (i.e., one object is composed of another object). An association (i.e., a dependency) between objects is represented by a solid line. A line with a directed arrow at one end denotes an inheritance relationship (i.e., one object is a specialization or extension of another object). The adornments, such as 1..*, indicates the number of potential objects participating in the relationship (in this case, the * means many).

The main class, Network, is composed of a File Output class, a Clock class, a Random Scan class, a Crossbar class, and a Routing Table class. The Network class also *gets data from* the Data Cube class, and the Data Cube class *gets data from* the Process Set class. In these two cases, information relating to the data cube and the process set are passed to instantiated Network objects prior to the start of the simulation. Because both the data cube and process set may change, while the structure of the network remains the same, the two corresponding classes are not contained within the composition of the Network class. This allows a single Network object to operate on different data cubes and process sets without regeneration of the network. A more detailed description of the Network class will be presented in Section 6.2.

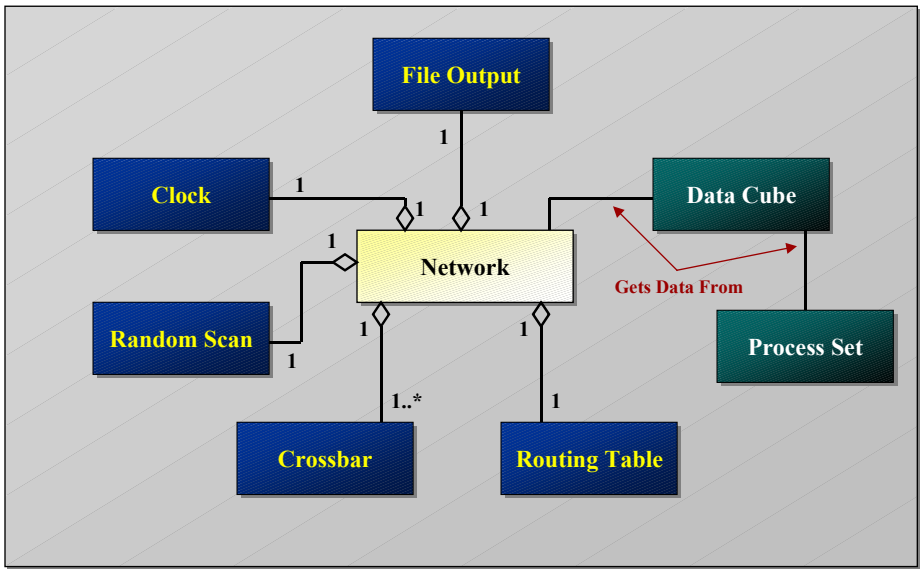

Fig. 6.1 A UML class diagram of the Network class.

A UML class diagram of the Crossbar class is illustrated in Fig. 6.2. The Crossbar class is composed of six Link objects (i.e., two parent links and four child links) and four Compute Node objects. For cases where a Crossbar object is positioned at the lowest level of the fat-tree architecture, the four Compute Node objects are enabled, and the four child Link objects are disabled. Otherwise, the four child Link objects are enabled and the four Compute Node objects are disabled for Crossbar objects not located at the lowest level in the network. Also shown in Fig. 6.2 is a UML diagram of the Compute Node class. Each Compute Node class is composed of two Message Queue objects, one outgoing and one received queue, and two Packet Stack objects, one outgoing and one received stack. A Message Queue object may be composed of zero or more Message objects, and zero or more packets may be included within each Packet Stack object. A more detailed account of each object represented in Fig. 6.2 is discussed in Section 6.2.

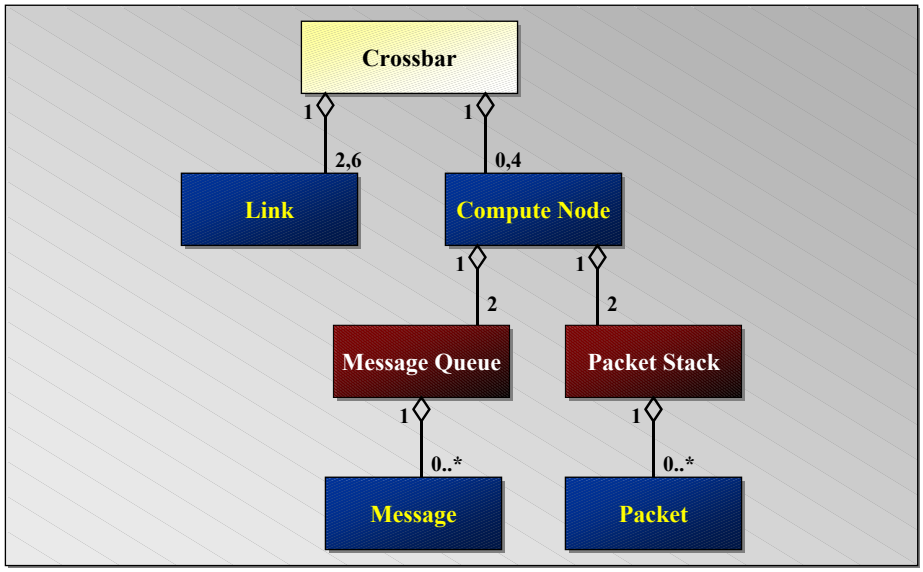

Fig. 6.2 A UML class diagram of the Crossbar class.

Both the Message Queue object and Packet Stack object are composed of data items that traverse the network links during phases of communication. Because a Packet class and a Message class contain common instance variables and operations, an abstract class, Data, was designed to collect the common components of each class (see Fig. 6.3). The

goal of the abstract class definition is to reuse as much of the data and methods as possible. In this case, both the Message class and the Packet class inherit from the abstract Data class. In addition, a Header Route List class composes each Packet class. The Route List class contains one or more Route objects that posses the information necessary to route a packet through the network to its destination.

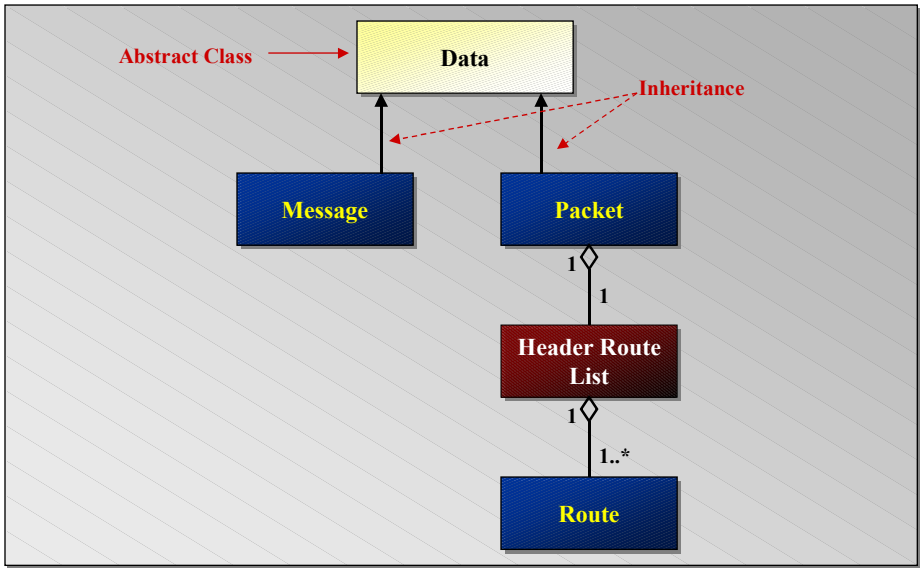

Fig. 6.3 A UML class diagram of the Data class.

6.2 Refining Class Operations

Once the classes in the solution space for the development of the simulator were defined, the next step involved formulating the operations for each class. In general, the operations defined for each class may be classified into three broad categories: (1) operations that manipulate the data; (2) operations that perform a computation; and (3) operations that monitor an object for the occurrence of an event [19].

The operations and class refinement of the Network class are shown in Fig. 6.4. Once instantiated, an instance of a Network class dynamically constructs the appropriately sized network based on the required number of CEs. After allocation of the crossbars and the generation of the connections between each level, the instantiated Network object

proceeds with the following two tasks. First, the object enables the correct number of CNs that equates to the number of required CEs. Second, a Routing Table object is dynamically constructed, based on the size of the network, that defines the routing between any two CNs in the network. This information is used to generate the source-to-destination packet header routing information for each packet prior to transmission.

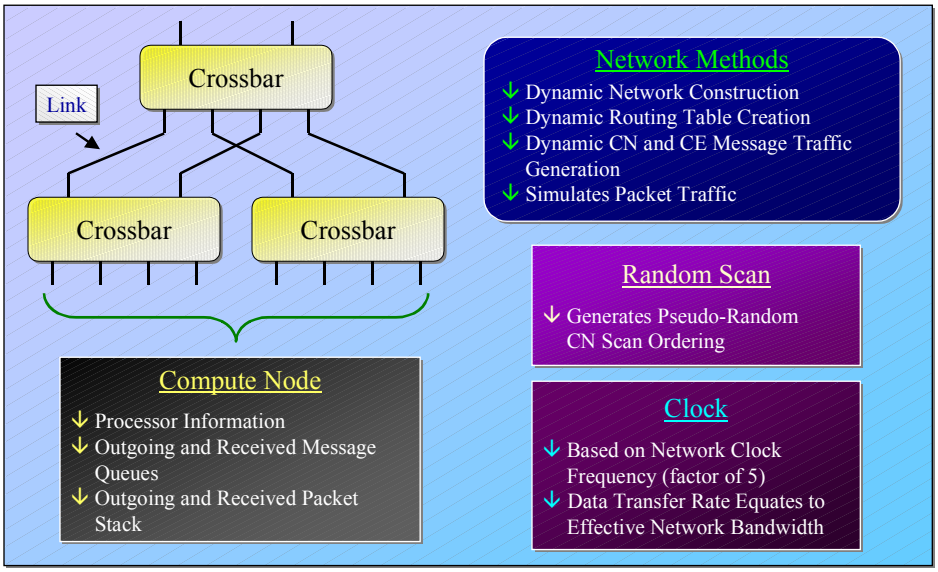

Fig. 6.4 Network class refinement and operations.

Before simulation, the outgoing messages queues of the Compute Node objects are loaded with the appropriate data messages for transmission. Recalling from Fig. 6.1, the Network object *gets data from* the Data Cube. The Data Cube object requests the configuration of the process set from the Process Set object. Using the process set configuration, the Data Cube object generates a CE message traffic matrix, which defines the required communications. The Network object requests the information in the traffic matrix. Based on the values in the matrix, the Network object generates the required message traffic for each CN or CE to accomplish either corner-turn communication pattern. To model (through simulation) the effects associated with how data is mapped onto the CNs of the Mercury system using a sub-cube partitioning approach, the messages in the

outgoing message queues at each CN are randomly ordered prior to message communication.

The complexity of simulating, in software, the message traffic of a real-time embedded parallel system requires significant management. During phases of communication in a real-time embedded system, possibly numerous data items are making connections and transmitting information simultaneously. Simulating the concurrency of such events in a single threaded software simulator is challenging. One approach to solving this problem would be to generate a separate thread of execution for each data packet that is currently transmitting data or attempting to establish a path to its respective destination in the network. Unfortunately, the overhead associated with managing the potentially high volume of currently executing threads at a given time would severely degrade the performance of the simulator. Furthermore, the crossbars and their associated connections would be a shared resource amid all the concurrently executing threads; as a result, critical sections, mutexes, or semaphores would be required to protect the shared resources by ensuring that only one thread can modify a shared resource at any given time. Implementing the necessary requirements to solve the data dependency problem would also require significant processing resources.

A second approach to simulating the real-time aspect of the network involves implementing a single thread of execution and scanning the compute nodes with current packets, during a given clock cycle, in a random order. Although this approach does not realistically simulate the exact execution of the real multicomputer, it does introduce some equality amongst the current packets. Additionally, this approach eliminates any shared resource problem that surfaced in the first approach.

To facilitate the necessity to scan the enabled Compute Node objects in random order, a Random Scan object was incorporated into the design. An instance of a Random Scan object generates a pseudo-random sequence of the enabled CNs. The simulator then proceeds, in the order designated by the Random Scan object, to evaluate and potentially alter the state of a packet at the specified CN. Prior to the execution of pass 1 of each simulation cycle, a new random scan ordering is generated by the instantiated Random Scan object. Details pertaining to the simulation cycle will be discussed later in the section.

The final object encompassing the Network Object is the Clock object. The clock object is based on the RACE multicomputer clock of 40 MHz (i.e., .025 μs period); however, the simulation clock operates at 5 times the frequency of the actual clock (i.e., .125 μs period). The reasons for selecting a multiple of the true clock cycle are three fold. First, the initial packet start-up cost is consumed in one simulation clock cycle. Second, the time required to arbitrate through a crossbar takes more than one actual clock cycle. Third, because a majority of the operations require more than one cycle to complete and implementing a simulation clock cycle of .025 μs would increase the number of required simulation cycles while degrading overall performance, an appropriate multiple of the actual clock frequency was selected for the simulation clock. Obviously, certain side effects result from the multiple-cycled simulation clock. First, because the effective data transfer rate of the actual network is 157.5 MB/s, the simulator transfers approximately 20 data bytes per simulation clock cycle. Second, during one simulation clock cycle, a packet can arbitrate through two crossbars.

A major operation of the Crossbar object entails the implementation of the hardware priority arbitration algorithms. Clearly, the RACEway architecture supports a large number of simultaneous data transactions where each of these transactions can occur along independent paths that have no crossbar ports in common [20]. However, not all data transactions occur along independent paths. Whenever two or more transactions are contending for the same port at a given crossbar, arbitration is required. Recalling from Section 3.2, a user-programmable packet priority is provided to give the user some level of control over the given data transfer transaction's priority [20]. Unfortunately, user-programmable priorities do not eliminate the need for arbitration at the hardware level. For example, the hardware priority associated with a given path through a crossbar (defined by the entry and exit ports on that crossbar) comes into play when the two or more transactions having identical user-defined packet priorities are contending for the same exit port on a given crossbar [20].

Each Crossbar object is configured to implement both the Standard crossbar priority algorithm and the Top-Level crossbar arbitration algorithm (see Fig. 6.5). The selection of the appropriate algorithm depends on the location of the crossbar in the network. Crossbars

located at the top of a hierarchy of crossbars utilize the Top-Level algorithm, and all other crossbars employ the Standard algorithm. Both the Standard and Top-Level priority arbitration algorithms are defined as a function of the transaction entry and exit ports and transaction status. The assignment of the hardware priorities to crossbar transactions paths is far from trivial. Details of these two arbitration algorithms are provided in Section 3.2.

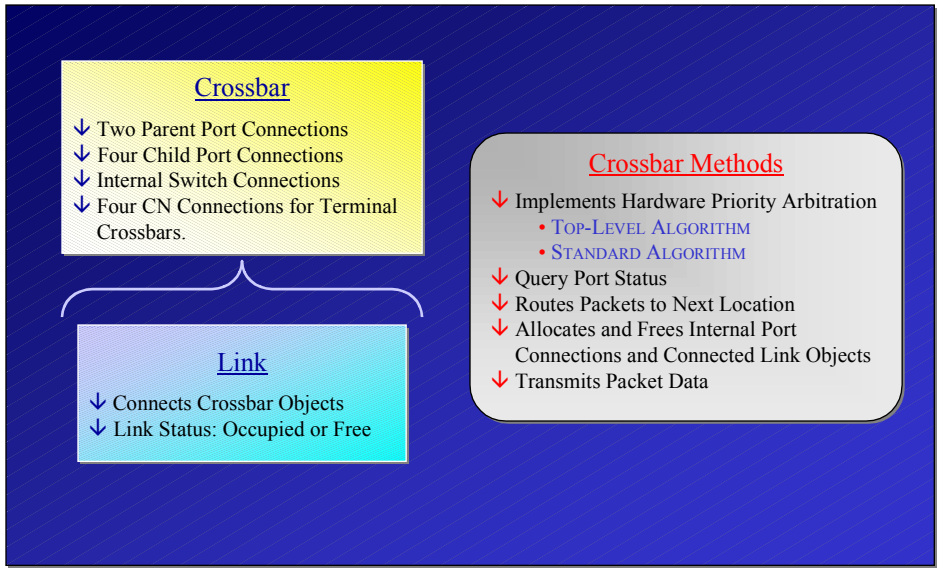

Fig. 6.5 Crossbar class refinement and operations.

In addition to the hardware arbitration, a Crossbar object exams the status of its internal and external ports and routes packets through the crossbar to the next location. A crossbar is also responsible for freeing its connections when a packet has completed or been suspended or killed. Finally, once the connection is established from the source to the destination CN, the crossbar transmits the data through the occupied connection.

The primary focus of the Compute Node class involves the management of the message queues and packet stacks (see Fig. 6.6). Because data is transferred from source to destination node across the RACEway network in packets of up to 2048 data bytes in length, each message in the outgoing message queue must be *exploded* into the appropriate number of corresponding packets. During simulation, the top message in the outgoing message queue is *exploded* into packets. After each of the packets for that message has

been transmitted to its respective destination node, the next message at the top of the queue is *exploded* into packets. This process repeats for each CN until all the outgoing messages queues are empty.

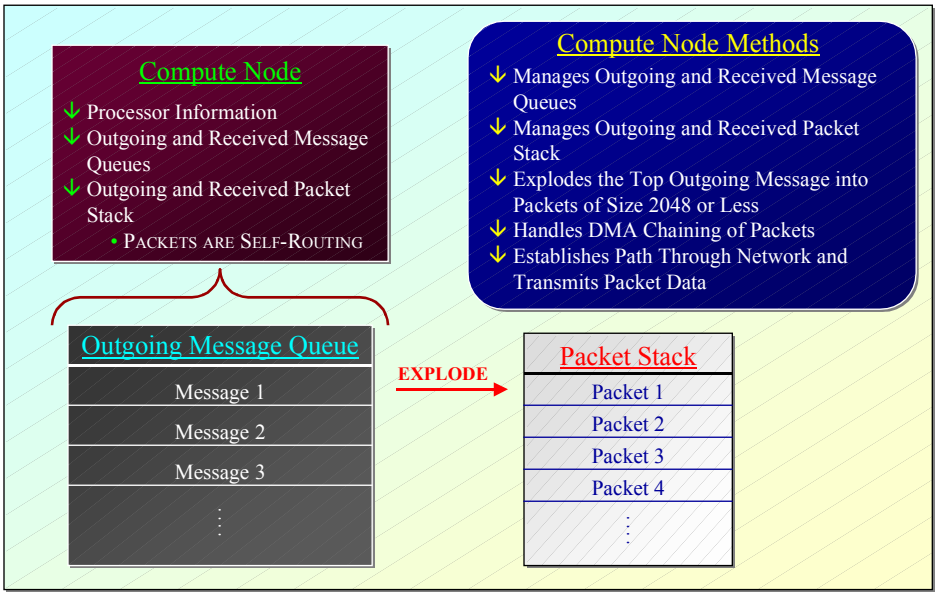

Fig. 6.6 Compute Node class refinement and operations.

During the generation of a packet, a packet header is constructed. The packet header (i.e., the Route List object) contains the information for routing a packet through the sequence of crossbars from the source CN to the destination CN. The routing information is retrieved from the Routing Table object within the given Network object. Via user selection, packets destined for the same location may be direct memory access (DMA) chained together. Essentially, DMA chaining provides a mechanism for transferring blocks of data to the same location without paying the startup cost for each packet. Furthermore, the Compute Node object is responsible for initiating the request for arbitration through the first terminal crossbar. Once access to the terminal crossbar is established, the crossbars are responsible for routing the packet through the network to the destination. Finally, when an active, transmitting packet is suspended by another packet, the Compute Node object is responsible for generating a new packet composed of the unsent packet data.

6.3 UML Statecharts and Activity Diagrams of the Simulator

The UML statechart models are based on finite state machines using an extended Harel state chart notation with modifications to make them object-oriented [2l]. A statechart diagram represents a state machine and illustrates the sequence of states that an object goes through during its life cycle. The states are represented by a rectangular box with rounded corners, and the transitions are represented by arrows connecting the states. The initial (pseudo) state is shown as a small solid filled dot representing any transition to the enclosing state [21]. A final (pseudo) state is shown as a small filled dot enclose by a circle representing the activity in the enclosing state [21]. In a state diagram, the occurrence of an event may trigger a state transition.

A UML Activity model is a variation of a state machine in which the states are activities representing the performance of operations, and the transitions are triggered by the completion of an event [21]. The purpose of an activity diagram is to focus on the flows driven by internal processing. Statecharts and not Activity Diagrams should be used in situations where asynchronous events occur.

Fig. 6.7 shows a UML Activity model of the software simulator. The ovals represent action states, and the transitions, which are triggered by the end of the activity, are depicted as lines with directed arrows. A diamond represents a decision process. After the user enters information relating to the size of the network, the size of the STAP data cube, and the size of the process set, the simulator proceeds to build the network, the data cube, and the process set. Next, the simulator enables the appropriate setting for phase 1 or phase 2 communication traffic phase (described in the following paragraph), DMA chaining, and CN or CE message traffic pattern. Once the input parameters have been initialized, the simulator simulates the designated traffic pattern and displays the timing results.

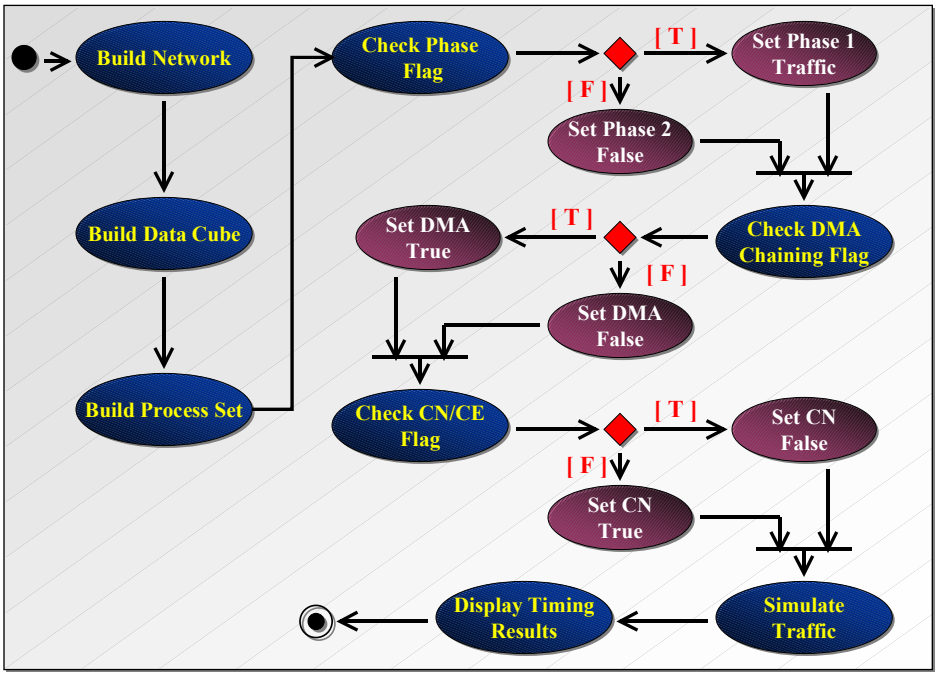

Fig. 6.7 A UML Activity Model of the Simulator.

The simulation of the network traffic is a complex and detailed process. Therefore, the low-level details of the operation of the simulator will not be discussed. Nevertheless, the important control-flow aspects of the simulation warrant explanation. The simulator's design incorporates a two-pass approach to simulating the packet traffic in the network. During one simulation clock cycle, both passes are executed. The primary objectives of the first pass are to build the packets, allocate link and crossbar resources for the packets, and transfers the packet data for each CN. The second pass performs necessary clean-up work that must be accomplished for suspended and completed packets prior to the next simulation clock cycle for each CN. The simulation process continues until all the messages have been transferred. As previously stated, the order in which the enabled CNs are scanned, during each simulation clock cycle, is random. Specifically, prior to the execution of pass 1, during a given simulation clock cycle, a new pseudo-random sequence of the CNs is generated, and CNs are scanned in that order. The CNs are also visited in

random order in pass 2, but the actual ordering of the visits in this pass has no affect on the network performance.

A combination of the Compute Node objects and the Crossbar objects are responsible for the transferring of the packets through the network. The CNs implement the two-pass simulation architecture that is required to deliver a packet from its source node to its destination node. The crossbars handle the arbitration of the connections at the switches as well as the allocation of the interconnected links. A UML statechart best illustrates the process performed by each CN object (see Fig. 6.8). First, an instantiated CN object determines if a current packet has already been removed from the packet stack. If so, the Compute Node object transitions to the Pass 1 state. In this view, the Pass 1 state is a superstate. (The states and transitions that occur during the simulation of Pass 1 will be elaborated on later in this section.) Otherwise, the Packet Stack is evaluated for the existence of an available packet. If the Packet Stack is not empty, the top packet is popped from the top of the stack and becomes the current packet. Afterwards, the CN object transitions to the Pass 1 state. An error code is generated if a failure occurs during the popping of the Packet Stack. In cases where the Packet Stack is empty, the Message Queue is evaluated for available messages. At this point, if the Message Queue is empty, the CN is tagged as completed, and control is passed back to the calling state. Otherwise, the top message is exploded into packets of size 2048 data bytes or less, and the CN transitions to the Pop the Top of Stack state. As illustrated in the figure, a CN is tagged as *done* only when the both the Message Queue and Packet Stack are empty and a current packet does not exist.

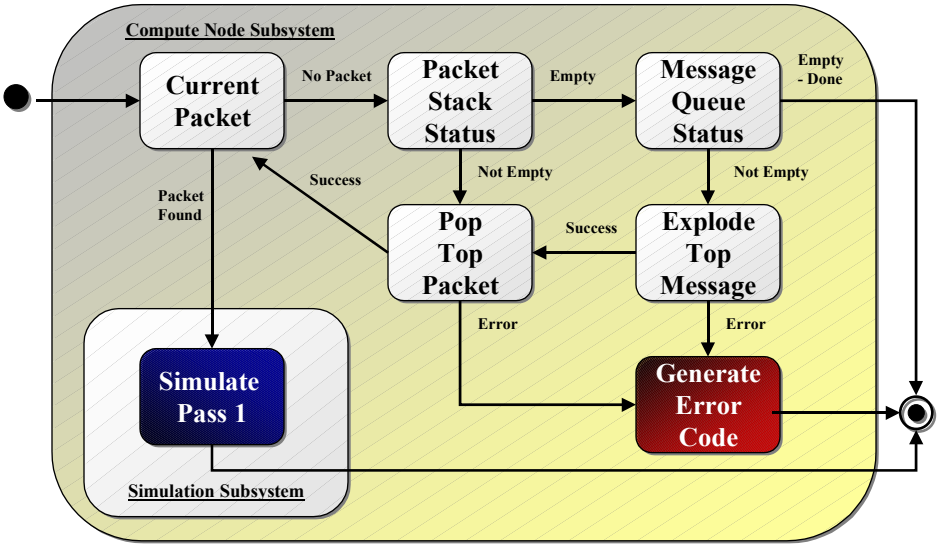

Fig. 6.8 A UML Statechart of the Compute Node class simulation Pass 1.

The Compute Node statechart diagram of the operations executed during Pass 2 of the simulation is significantly more simplistic than that of Pass 1 (see Fig. 6.9). If a current packet exist, a transition to the Pass 2 superstate takes place. On the other hand, if there is not a current packet at the current CN, a transition to the exit state occurs.

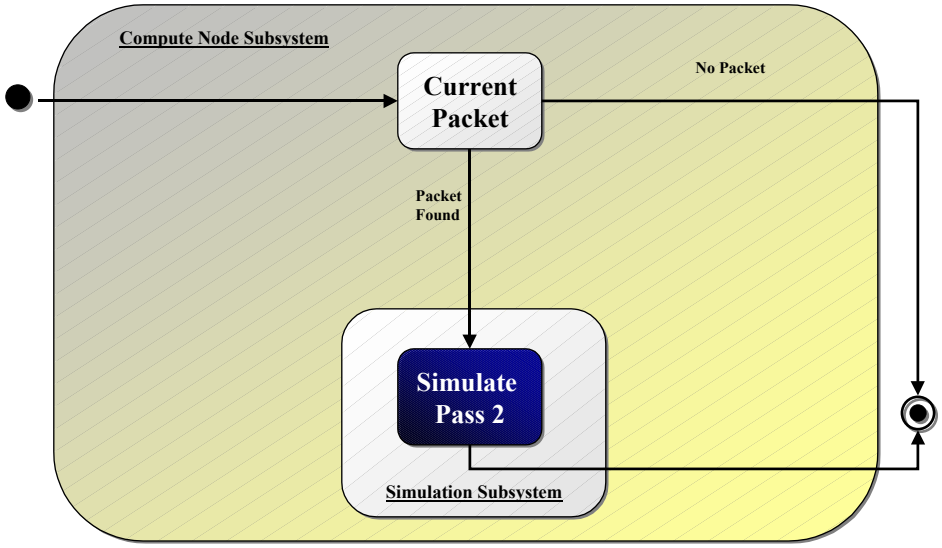

Fig. 6.9 A UML Statechart of the Compute Node class simulation Pass 2.

One of the main objectives of the software simulator is to transfer data (i.e., packets) through the network. The transitions of the Packet objects detail the underlying operation of the simulator. Fig. 6.10 illustrates the state diagram for a Packet object. The blue arrows represent transitions that can occur only during Pass 1 of the simulation, while transitions that take place during Pass 2 are indicated by red arrows. Initially, a given packet begins in either the Start Up state or Ready state. Normally, a packet begins in the Start Up state; however, for cases where DMA chaining of packets to the same destination CN is utilized, the packet's initial state is Ready. A packet in the Ready state is ready for route arbitration to the destination node. After the packet header is constructed, a packet in the Start Up state transitions to the Ready state. A Ready packet may transition to either an Active state, a Blocked state, or stay in the current state. A change to the Active state transpires only if the connection to the destination node is established. If the packet successfully acquires a partial path through the network but does not occupy a complete route to its destination, the packet transitions to the Blocked state. Finally, if the packet is unable to make any progression through the network, the packet remains in the Ready state.

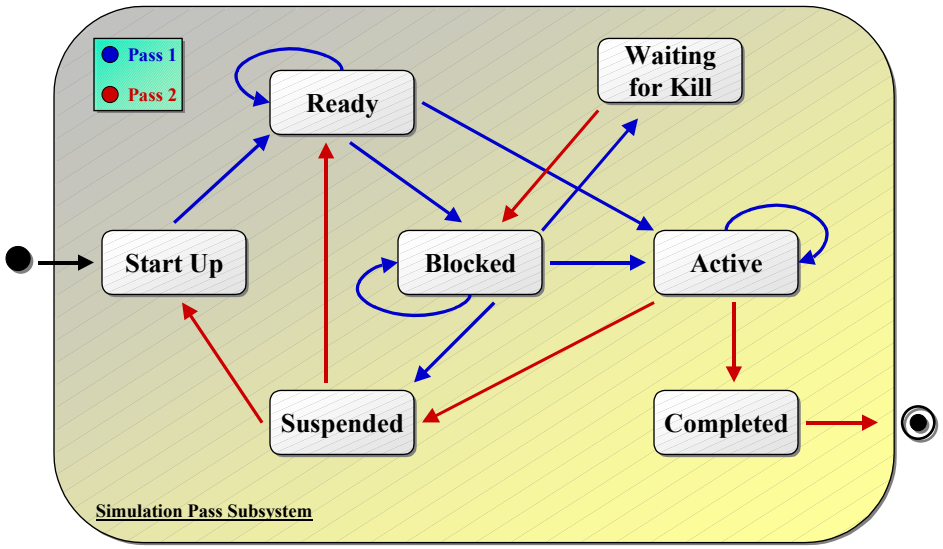

Fig. 6.10 A UML Statechart of the Packet class

A Blocked packet is categorized as a packet that occupies at least one connection in the network but has yet to make a complete connection to its final destination. A packet may be blocked for two reasons. First, the exit port that the packet requires at a particular crossbar is occupied by another packet, and the hardware arbitration algorithm does not allow for the suspension of that packet. Second, the simulation clock cycle completed before the arbitration to the destination was achieved. A Blocked packet may transition to any of four possible states. It changes to the Active state if a connection to the destination node is established. A Blocked packet may also transition to the Suspended state if it is terminated by a another packet. If the currently blocked packet suspends another packet, the current packet transitions to the Waiting for Kill state. Finally, a packet may remain in the Blocked state for the same two reasons that it first arrived in this state.

A packet that is transmitting its contents is in the Active state. Throughout the transfer of packet data, the packet remains in the Active state unless suspended. A packet in the Active state may be suspend by another packet based on the complicated hardware arbitration algorithms at the crossbar level. Once an Active packet transfers its data, a transition to the Completed state occurs.

Packets that are terminated prior to completion transition to the Suspended state. While in the Suspended state, packets that were previously Active require the construction of a new packet with the remaining data content. Additionally, the connections occupied by the packet are freed during the next simulation clock cycle, and the newly formed packet transitions to the Start Up state. Packets arriving at the Suspended state that were previously in the Blocked state are handled differently. Because none of the data was transferred, a new packet is not required; however, the packet header does require updating. After updating the packet header with new routing information, a transition to the Start Up state occurs.

Finally, packets in the Wait for Kill state are waiting for a suspended packet to release its occupied connections. During the pass 2, the waiting packet transitions to the Blocked state. Once in the Blocked state, the packet may be able to gain access to the newly freed connections in the next clock cycle, but because this is a real-time system, there is no guarantee that the connection will be available in the next clock cycle. For instance, another packet may allocate the connection before the waiting (now currently blocked) packet can occupy the connection.

6.4 Implementation

The software simulator was written in Java, although the design is language independent. Java was selected for its portability and the need for Internet access to the simulator. The actual implementation, which is based on the design described in this section, was developed in Borland's JBuilder 1.0. Although the studies on network traffic described in the next chapter were conducted based on STAP algorithms, the simulator is designed to simulate any communication pattern requirement. That is, the simulator can take as input any CE traffic matrix. After implementation, the software simulator was extensively tested prior to the collection of data.

CHAPTER VII

PRELIMINARY NUMERICAL STUDIES

Recalling that the objective of this research is the design and implementation a network simulator to model the effect data mapping and communication scheduling has on the performance of a STAP algorithm on the Mercury RACE system. Determining the optimal communication schedule of queued messages during the two phases of data re-partitioning is beyond the scope of this research. In addition to scheduling, one could consider the complexity of determining the optimal routing of the queued messages (recall that there are multiple paths connecting pairs of CNs in the RACEway system). The goal of the research is not to *solve* these types of optimizations, but to *simulate* the effects different schedules and data mapping have on performance. The scope of the research involved the investigation of the following four areas: process set configuration, CN and CE message traffic, adaptive routing settings, and DMA chaining options.

7.1 Process Set Configuration

In a sub-cube bar partitioning approach, the STAP data cube is distributed to the available CEs by partitioning the data cube into sub-cube bars by applying a two-dimensional process set to the data cube. Before processing can take place at the next phase, the data vectors must be re-distributed to form contiguous vectors of the next dimension. Five separate studies were conducted related to the size of a process set. Each simulation involved recording both the phase 1 and phase 2 completion times for fifty randomly selected schedules. After these fifty completion times for each phase were collected, the resulting data was placed in histogram format.

7.1.1 Performance Metric for a $3x12$ and $4x12$ Process Set

Fig. 7.1 shows the timing results collected from both a $3x12$ and a $4x12$ process set. For a $3x12$ process set, which includes thirty-six CEs or 12 CNs, the horizontal dimension is 3, and the vertical dimension is 12. Intuitively, a $4x12$ process set contains 16 CNs or 48 CEs, and the horizontal dimension is 4 while the 12 represents the vertical dimension. The notation above the illustrated graph, which is consistent throughout this chapter, defines the additional parameters of the simulation. The first label, CN, signifies that the message traffic pattern generated was CN traffic. The parameters of the STAP data cube are defined by the sizes of the range (R) dimension, the pulse (P) dimension, and the channel (C) dimension. For this simulation, an antenna array including sixteen channels obtained two hundred range samples from a CPI of twenty-two pulses. Additionally, adaptive routing is used to adaptively route the packets that enter the child ports to an exit parent port. In this instance, the F parent port is evaluated prior to the E parent port (i.e., adaptive routing, F first). The x-axis expresses the time line in milliseconds, and the y-axis denotes the tallied appearance of a particular time interval.

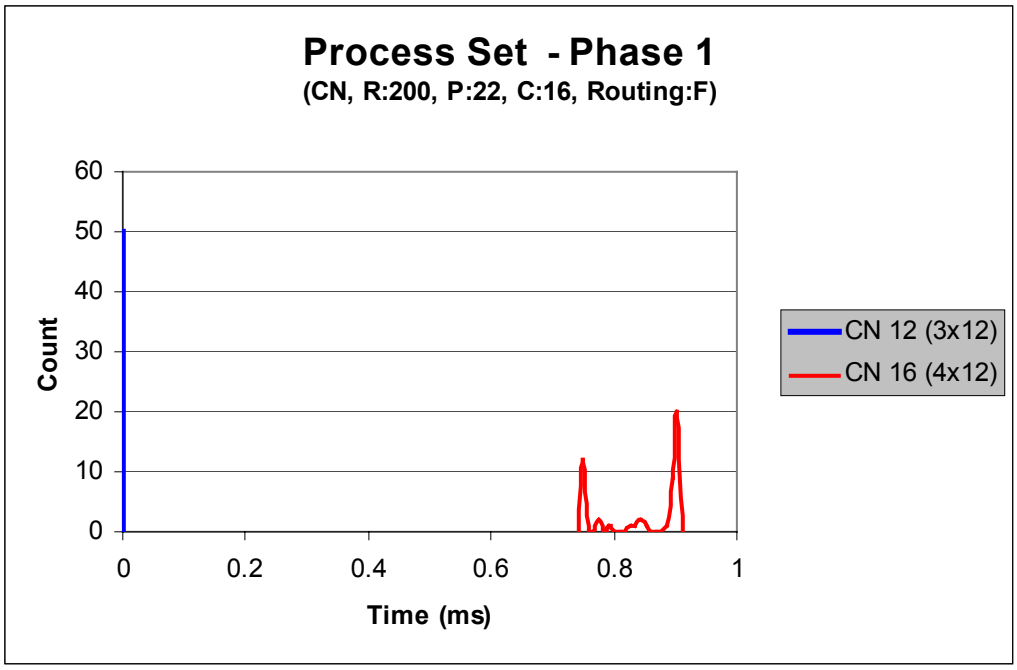

Fig. 7.1: Phase 1 performance metric for a $3x12$ and $4x12$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

Notice that the communication time for the $3x12$ process set is zero. Recall that each CN contains 3 CEs, so in this case, the data required for the second processing phase is located on the correct CN because the horizontal dimension contains exactly 3 CEs. For the CN 16 case, where the horizontal dimension contains 4 CEs, communication is required before processing of the next dimension can commence. In this instance, the $4x12$ process set is outperformed by the $3x12$ process set.

An examination of the communication times for the second corner-turn reveals a different outcome (see Fig. 7.2). In this simulation, the communication times are quite similar, although the CN 12 configuration again records the smallest time by approximately .25ms. Intuitively, the CN 12 configuration has fewer messages to communication because there are fewer processors and more data locate at each CN. However, in the CN 16 case, the STAP data is distributed to more processors, which results in a larger number of messages during the phase 2 corner-turn distribution of the data.

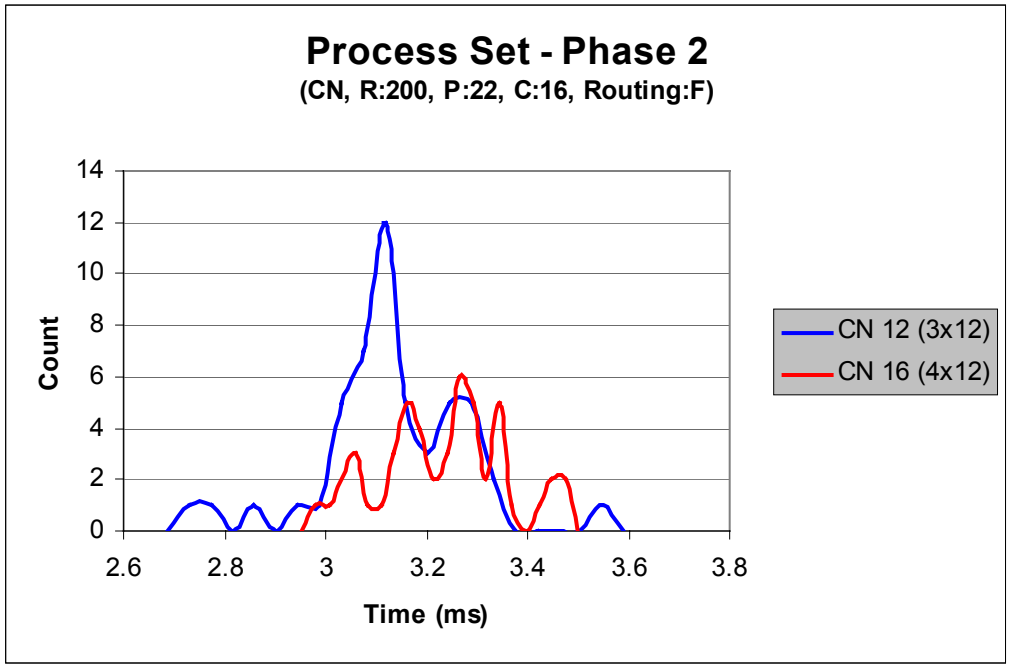

Fig. 7.2: Phase 2 performance metric for a $3x12$ and $4x12$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

7.1.2 Performance Metric for a $6x4$ and $4x6$ Process Set

Fig. 7.3 illustrates the phase 1 simulation timing results obtained for an 8 CN system configured with the STAP data cube partitioned by a $6x4$ and a $4x6$ process set. In this example, the communication pattern for the $6x4$ process set records the same time for each iteration. Because the horizontal dimension is a factor of three, the communication pattern is a more predictable. In the first phase, CEs are sending messages to other CEs in the same row (i.e., the horizontal dimension). Additionally, because the data cube size is particularly large (i.e., 800 range samples, 32 pulses, and 22 channels), the messages are of significant size, which translates to a high number of packets sent from the same source node to the same destination node. Furthermore, there is only one message in the outgoing queue of each CN, so the number of possible orderings at each CN is one. Unfortunately, this is not the case with the $4x6$ process set. In this instance, there is more than 1 message in the outgoing queues, and the messages are not uniform in size, resulting in a more diverse recording of completed simulation times.

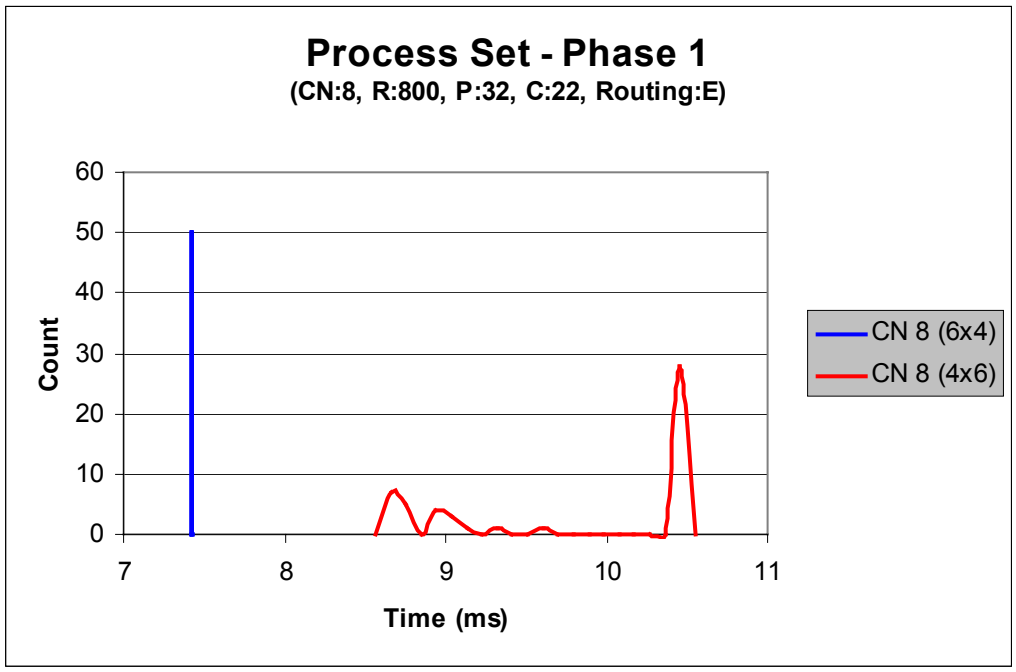

Fig. 7.3: Phase 1 performance metric for a $6x4$ and $4x6$ process set with range: 800, pulses: 32, channels: 22, and adaptive E routing.

In contrast to the different communication times in phase 1, the times in phase 2 have less variation between the two process sets (see Fig. 7.4). Furthermore, the completion times for phase 2 are a factor of 3 to 4 greater than phase 1 times because there is more data to distribute in this phase. In fact, the phase 2 communication dominates the total completion time for each case presented in this chapter. This simulation reveals that the $6x4$ process set size would yield the shortest total completion time.

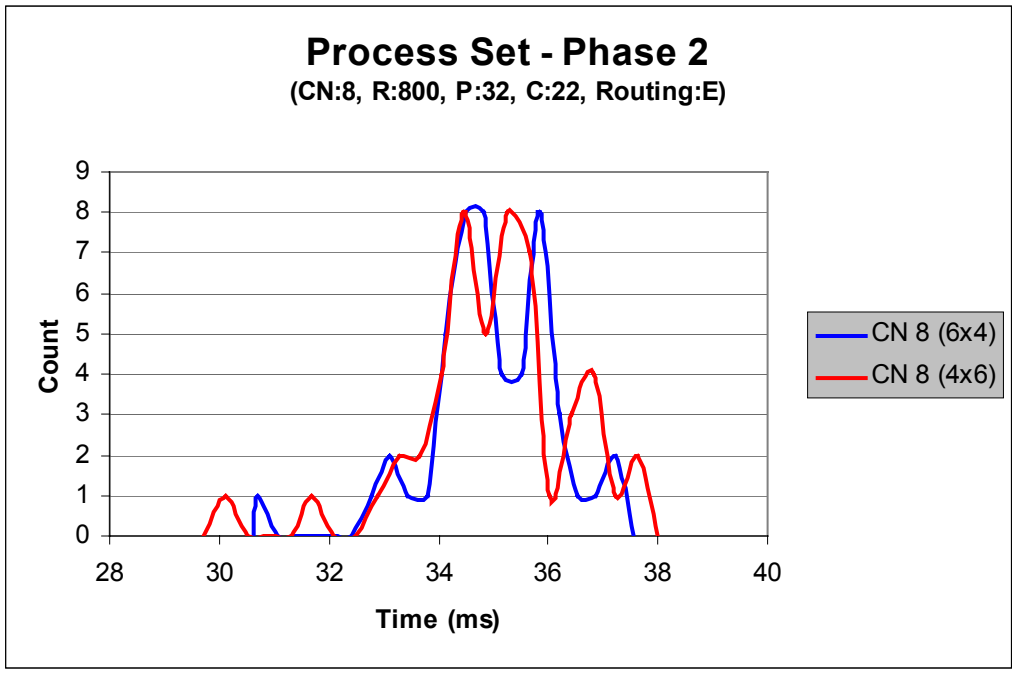

Fig. 7.4: Phase 2 performance metric for a $6x4$ and $4x6$ process set with range: 800, pulses: 32, channels: 22, and adaptive E routing.

7.1.3 Performance Metric for a $12x3$, $9x4$, $6x6$, and $4x9$ Process Set

The object of this simulation is to illustrate, for a given 12 CN system configuration, the effects the process set choice can have on performance. Figs. 7.5 and 7.6 display the simulation timing results for a $12x3$, $9x4$, $6x6$, and $4x9$ process set for communication phases 1 and 2, respectively. For a $6x6$ process set, the communication pattern for phase 1 is very regular which results in a low degree of variation of the recorded completion time.

In addition, the $4x9$ process set performs better in phase 1 than both the $12x3$ and $9x4$ process set. Furthermore, this simulation unveiled that the phase 1 communication benefited from lower horizontal dimension process set value. However, it is important to note that this may not be true for all horizontal cases (i.e., data cube sizes, routing, options, etc.).

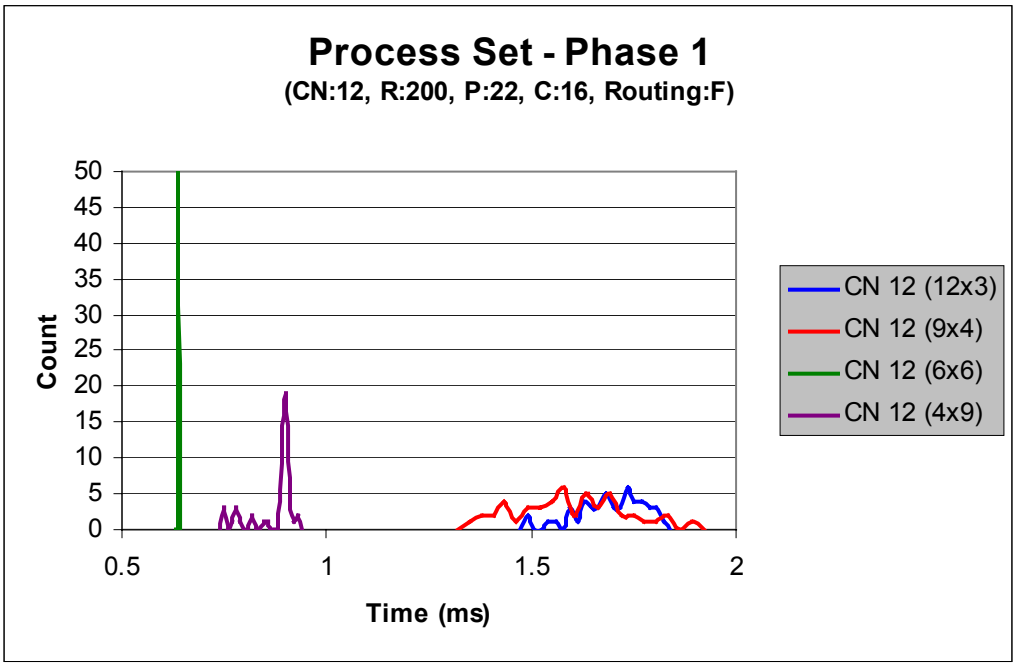

Fig. 7.5: Phase 1 performance metric for a $12x3$, $9x4$, $6x6$, and $4x9$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

In phase 2, the variation of completion times ranged from 2.9 to 3.75 milliseconds, with the $12x3$ process set on average performing poorest. The $4x9$ arrangement of compute elements registers the lowest communication time, while the remaining three process sets recorded slightly higher times.

Once a process set is selected, the dimension sizes of the process set may not change between phases 1 and 2. Recall that phase 2 communication depends on the resultant communication of phase 1. So a change in the dimension of the phase 2 process set does

not reflect the actual location of the source data. As a result, the process sets with the lowest times for corner-turn communication phases 1 and 2 must have the same process set. However, the ordering of the CEs within the structure of the process could be altered between phases.

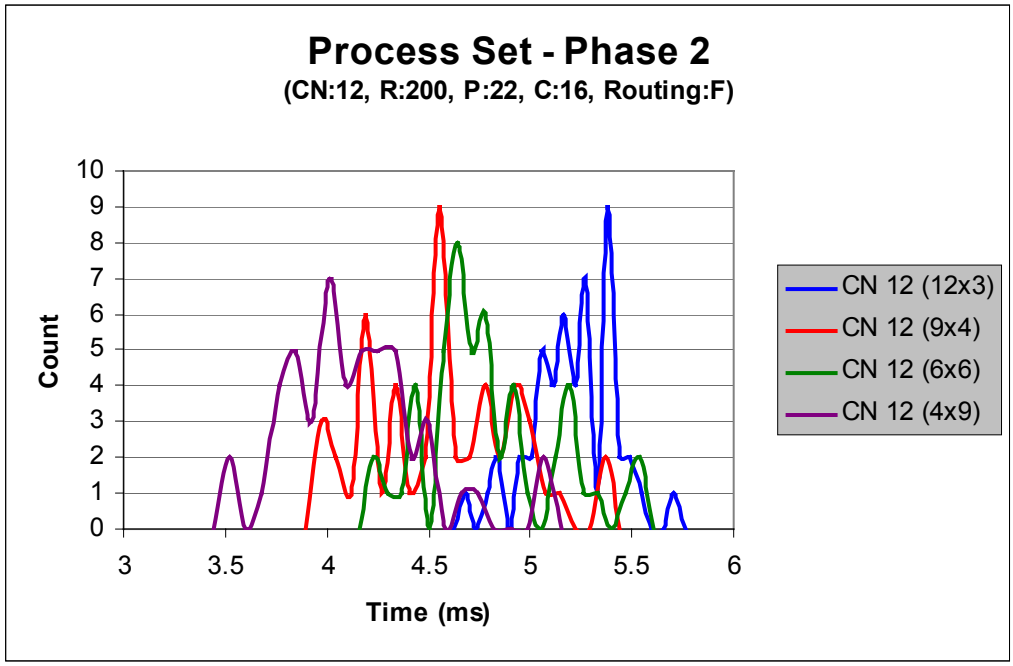

Fig. 7.6: Phase 2 performance metric for a $12x3$, $9x4$, $6x6$, and $4x9$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

7.1.4 Performance Metric for a $3x12$, $12x3$, and $4x9$ Process Set

One of the possible process sets neglected in Section 7.1.3 was the $3x12$ set. The $3x12$ process set was not included in the above section because of the limited graphing space. The $3x12$ process set requires no communication during the phase 1 communication cycle due to the dimensions of the process set (see Fig. 7.7). The horizontal dimension of the process set corresponds to the number of available CEs on a

given CN; as a result, the data required for both range compression and Doppler filtering are currently available on the same CN. In this instance, there is no data transfer requirement for phase 1. The $12x3$ and $4x9$ process set, which are elaborated on in Section 7.1.3, are include here for comparison only.

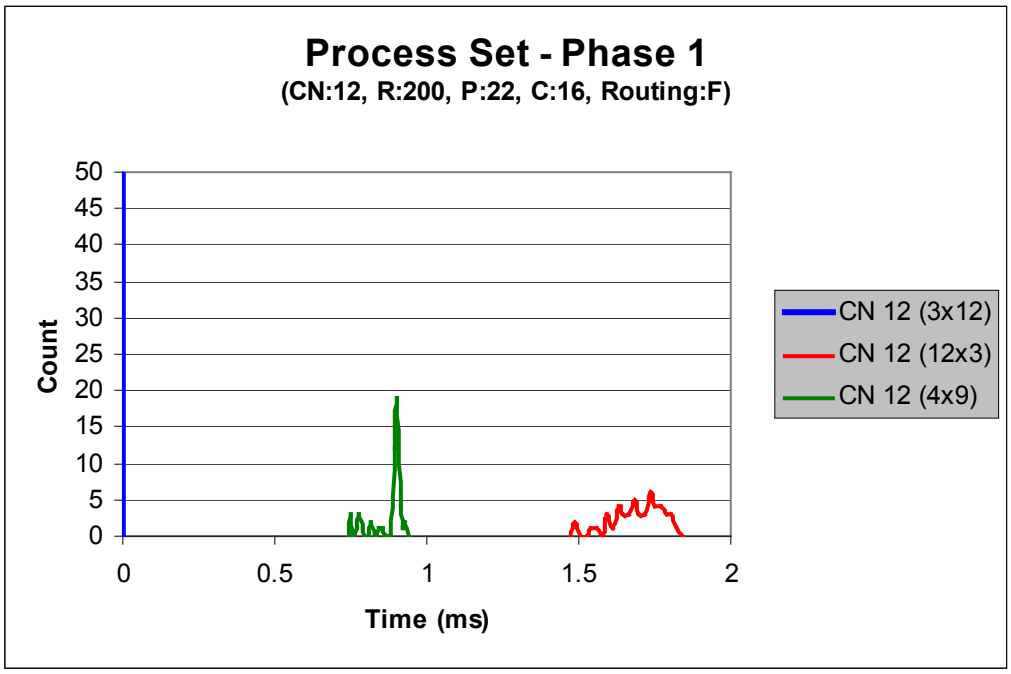

Fig. 7.7: Phase 1 performance metric for a $3x12$, $12x3$, and $4x9$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

In the second corner-turn phase, the $3x12$ partitioning of the data cube also performs the data redistribution in the shortest period (see Fig. 7.8). Recalling from Section 7.1.3, the $4x9$ process set was the best overall performer; nevertheless, the $4x9$ process set is, on average, roughly a millisecond slower compared to the $3x12$ process set in phase 2. In addition, because the $3x12$ process set does not require communication in the first phase, it is the best process set for the listed problem parameters.

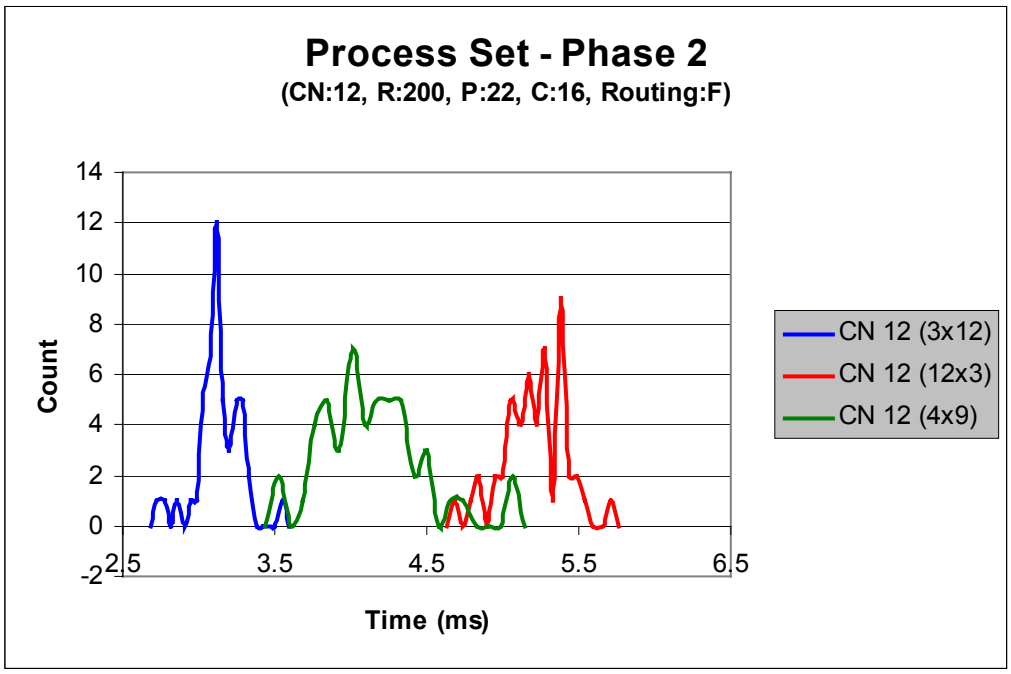

Fig. 7.8: Phase 2 performance metric for a $3x12$, $12x3$, and $4x9$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

7.1.5 Performance Metric for a $12x4$, $8x6$, and $4x12$ Process Set

The final process set performance metric compares a subset of the possible process set combinations for a 16 CN system. If the minimum sized dimension of a process set is 2, the number of possible process set size combinations is 10. For illustration purposes, only a subset of the possible process sets is presented for a set of fixed problem parameters. Fig. 7.9 shows the phase 1 communication times recorded for a $12x4$, $8x6$, and $4x12$ process set. In phase one, there is slightly less than a 50% difference in the separation between that shortest and longest time recorded. As illustrated in the graph, the $4x12$ process set is, on average, approximately 20% faster than the $8x6$ process set and 28% percent faster than the $12x4$ process set. The $8x6$ has a longer interval of recorded

completed times, which indicates that the ordering (i.e., the scheduling) of the messages impacts the performance of this process set more than the others.

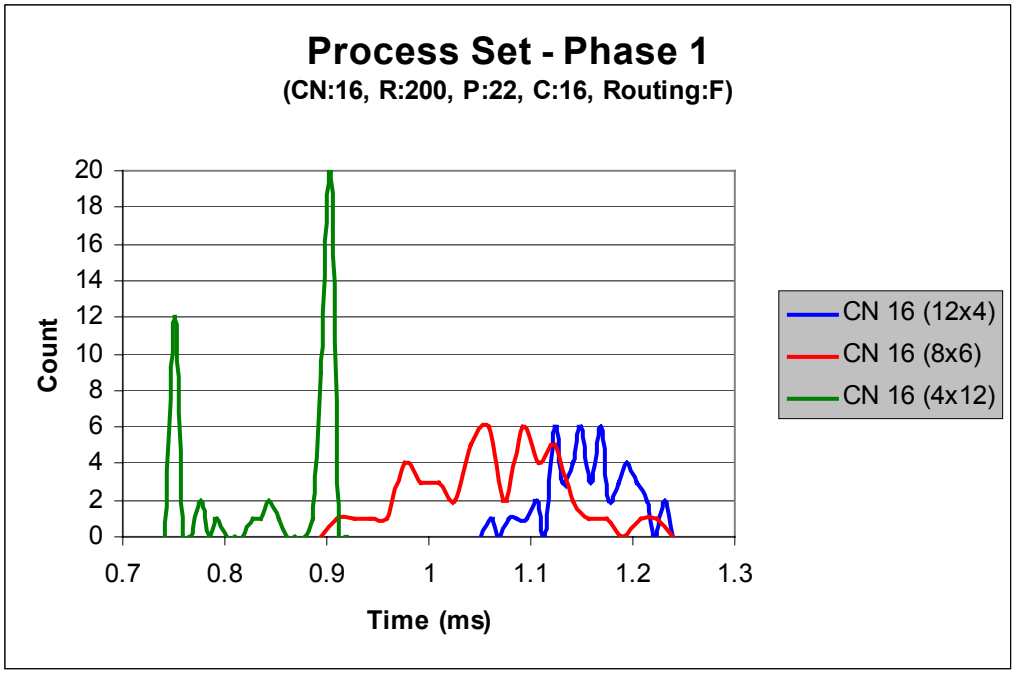

Fig. 7.9: Phase 1 performance metric for a $12x4$, $8x6$, and $4x12$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

The $4x12$ process set also produces the best completion time for the phase 2 communication pattern (see Fig. 7.10). In fact, the performance increase is almost thirty percent. The $12x4$ and $8x6$ process sets generate comparable results during phase 2, but each are roughly 1 to 1.25 ms slower in the best case. In addition, there is around a 1 ms variation in completion times which indicates message ordering affects performance.

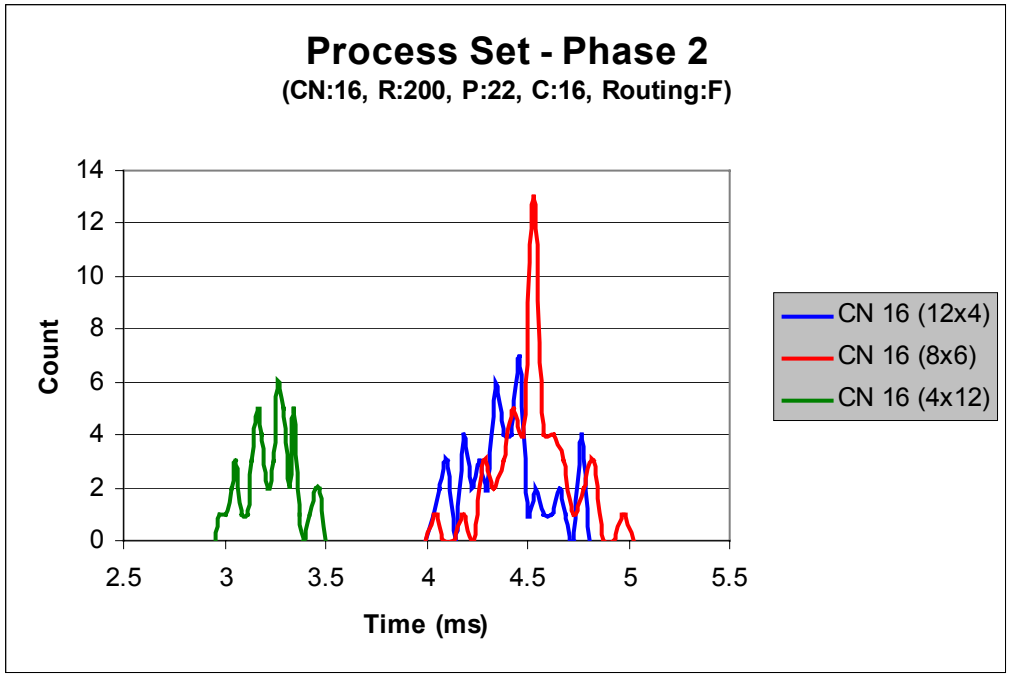

Fig. 7.10: Phase 2 performance metric for a $12x4$, $8x6$, and $4x12$ process set with range: 200, pulses: 22, channels: 16, and adaptive F routing.

7.2 Compute Node and Compute Element Traffic Investigation

The simulator is capable of generating either compute node traffic or compute element traffic. Messages from the same CN to the same destination CN are combined to form one message in a CN traffic approach to message generation. In contrast to CN traffic, each CE generates its own message to the destination CE, and messages to the same "CN destination" are not combined together in a CE traffic approach. In general, CN traffic contains larger and fewer messages than CE traffic. Simulations involving CE traffic contain more messages but each message is smaller. Three distinct investigations were conducted related CN and CE traffic. As in Section 7.1, each simulation involved recording both the phase 1 and phase 2 completion times for fifty simulations. After the fifty completion times for each phase were collected, the resulting data was placed in a histogram format.

7.2.1 Message Traffic Performance Metric for 16 CN ($12x4$) Configuration

A message traffic investigation of a 16 CN system with a $12x4$ process set configuration is provided in Fig. 7.11. From the graph, the CE traffic is approximately 10% faster than the CN traffic on average. In a CN traffic approach to message generation, the larger messages tend to allocate the same path through the network for longer back-to-back periods. However, when CE traffic is utilized, there is an increase in the number of messages, but the messages are smaller. By having more messages, the number of possible orderings in the outgoing queues increase. Additionally, because the messages are smaller and the orderings more diverse, the same CN is not necessary requesting a connection to the same destination node repeatedly. Furthermore, notice the variation in communication times for CE traffic. This indicates that the ordering of the messages in the queues affects the completion time of the communication pattern.

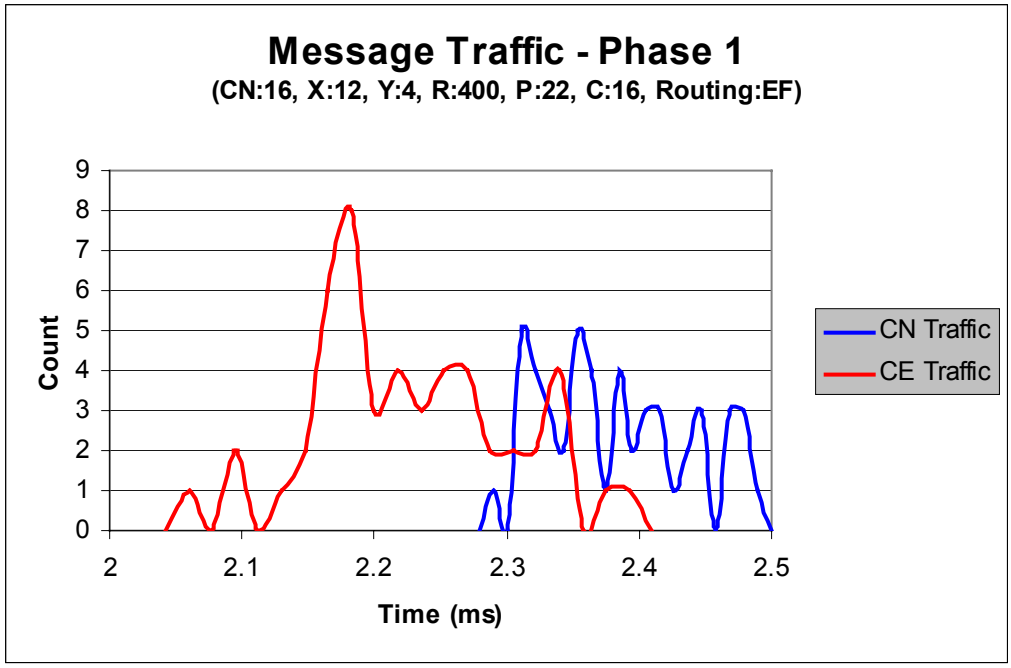

Fig. 7.11: Phase 1 message traffic performance metric for a 16 CN ($12x4$) configuration with range: 400, pulses: 22, channels: 16, and adaptive E/F routing.

The phase 2 message traffic results for the same set of parameters are opposite of phase 1 (see Fig. 7.12). In phase 2, the CN traffic appears to dominate the CE traffic. In fact, the CE traffic is approximately 25% slower than the CN traffic. For this scenario, it would be possible and advisable to employ a CE distribution of messages in phase 1, and a CN deployment of messages in phase 2. Also worth mentioning in the phase 2 CN message traffic results is the variation in completion time. This again indicates that the ordering of the outgoing messages is correlated to the completion time.

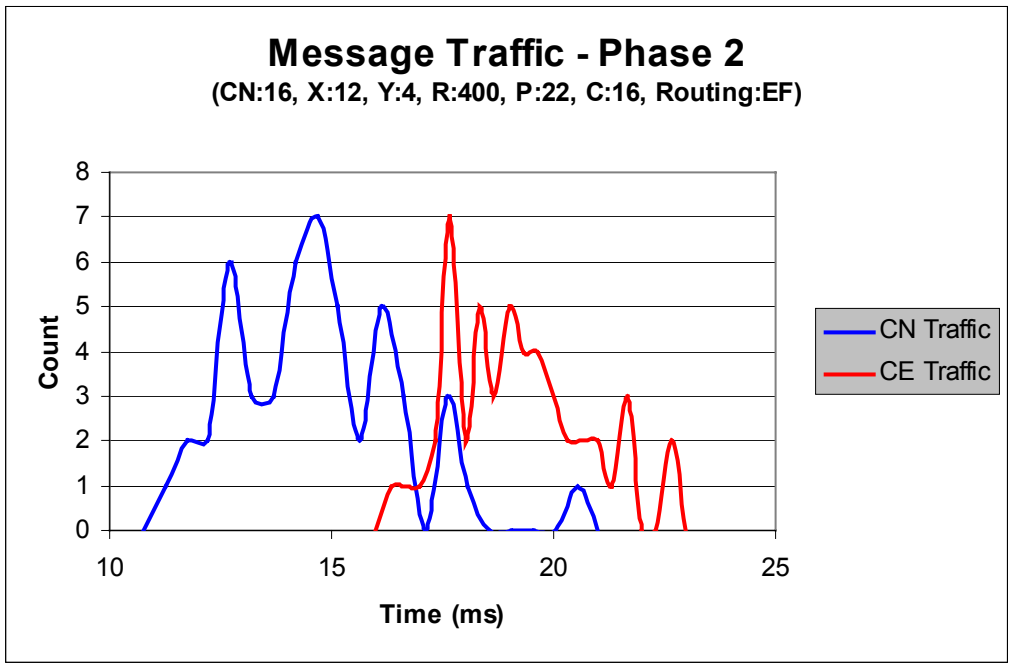

Fig. 7.12: Phase 2 message traffic performance metric for a 16 CN ($12x4$) configuration with range: 400, pulses: 22, channels: 16, and adaptive E/F routing.

7.2.2 Message Traffic Performance Metric for 16 CN ($6x8$) Configuration

In Section 7.2.1, the CN and CE traffic was examined for a 16 CN system configured with a $12x4$ process set. In this section, a 16 CN system is studied, but the process set configuration is $6x8$. Fig. 7.13 illustrates the results from the phase 1 corner-turn of the STAP data cube. In this scenario, the times for CN and CE message traffic in phase 1 are almost identical. In addition, the completion time for the $6x8$ process set is approximately 60% faster than for the $12x4$ process set in Section 7.2.1.

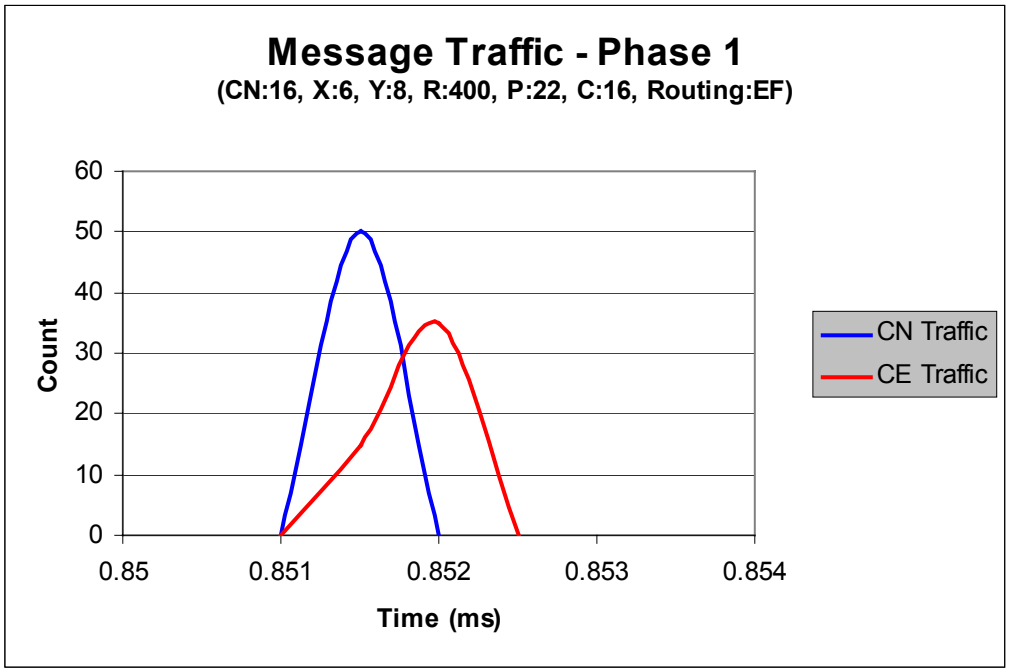

Fig. 7.13: Phase 1 message traffic performance metric for a 16 CN ($6x8$) configuration with range: 400, pulses: 22, channels: 16, and adaptive E/F routing.

As in Section 7.2.1, the CN traffic performs better than the CE traffic in phase 2 (see Fig. 7.14). On average, the CN traffic is approximately 30% quicker than the CE traffic. The variation in the CN traffic completion times also indicates that the ordering of the messages in the queues at each compute node is related to the performance of the communication pattern.

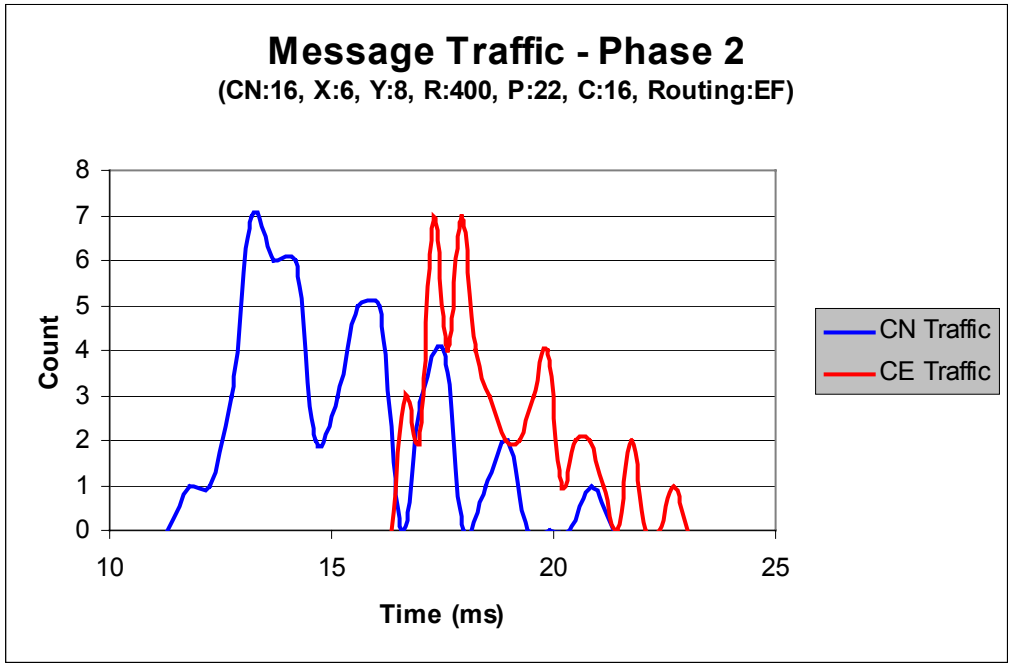

Fig. 7.14: Phase 2 message traffic performance metric for a 16 CN ($6x8$) configuration with range: 400, pulses: 22, channels: 16, and adaptive E/F routing.

7.2.3 Message Traffic Performance Metric for 12 CN ($6x6$) Configuration

In the previous two investigations, the communication phase prior to QR-Decomposition (i.e., phase 2), was best suited to CN traffic. However, this scenario will reveal that CE traffic could be best served for the phase 2 corner-turn. Fig. 7.15 illustrates the differences recorded in the completion times of both CN and CE traffic for a 12 CN system with a $6x6$ process set configuration. In this example, the CE traffic is only slightly slower than the CN traffic in phase 1. In fact, the overall difference is less than 1%.

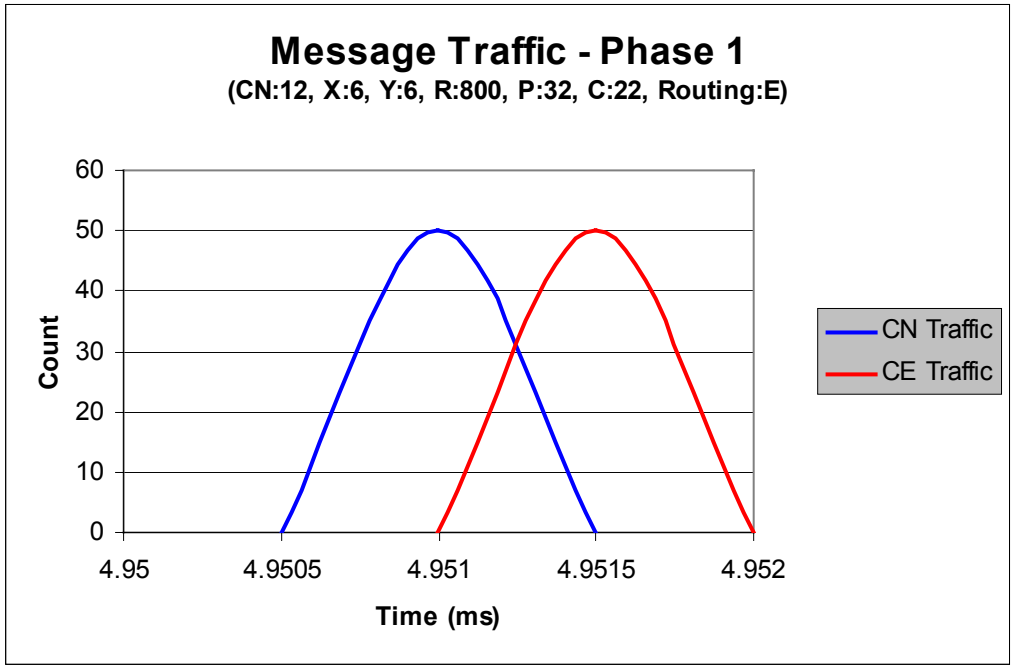

Fig. 7.15: Phase 1 message traffic performance metric for a 12 CN ($6x6$) configuration with range: 800, pulses: 32, channels: 22, and adaptive E routing.

In phase 2, the best completion times for both CN and CE traffic are approximately identically (see Fig. 7.16). In the above two examples, the CN traffic clearly out performed the CE traffic. For this example, the number of CNs, the size of the data cube, and the arrangement of the process set were altered to demonstrate that CE traffic, under the right conditions, could prove valuable during the phase 2 communication. In this simulation, the results indicate that the number of CNs, size of the STAP data cube, and the layout of the process set significantly affects the message traffic performance of communication phases on the Mercury network. In addition, there is a 10% variation in the completion times for the CN traffic. Consequently, the ordering of the messages can both improve and degrade the communication performance.

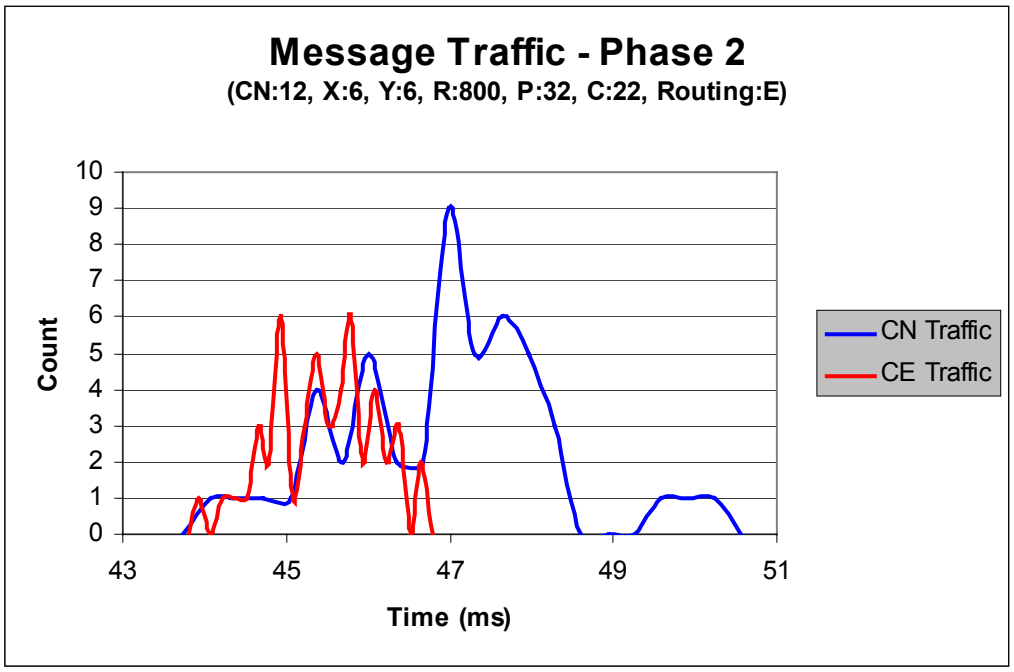

Fig. 7.16: Phase 2 message traffic performance metric for a 12 CN (6*x*6) configuration with range: 800, pulses: 32, channels: 22, and adaptive E routing.

7.3 Adaptive Routing Configurations

An adaptive routing technique may be used to route packets through the connections at each crossbar. Packets exiting one of the parent ports may be routed to the other parent port if the first port is not free and adaptive routing is used. Because each crossbar contains two parent ports, the adaptive routing option may be set to evaluate either E or F first. Additionally, a combination of both adaptive E and adaptive F could be used to arbitrate packets through the interconnection of crossbars. The following sections illustrate the effects of adaptive routing on the communication time. As before, each simulation involved recording both the phase 1 and phase 2 completion times for fifty simulations.

7.3.1 Adaptive Routing Performance Metric 1 for a 16 CN ($8x6$) Configuration

In the first simulation, a 16 CN system configured with an $8x6$ process set was studied. The STAP data cube size for this simulation was eight hundred range bins, thirty-two pulses, and twenty-two channels. For this simulation the combination of adaptive E and adaptive F routing recorded the shortest communication times (see Fig. 7.17). Additionally, the adaptive E/F configuration accounted for the smallest completion time interval. When configured with adaptive E routing only, the simulation record the widest variation in completion times. Again, this indicates that the scheduling of the messages impacts the performance of the communication pattern.

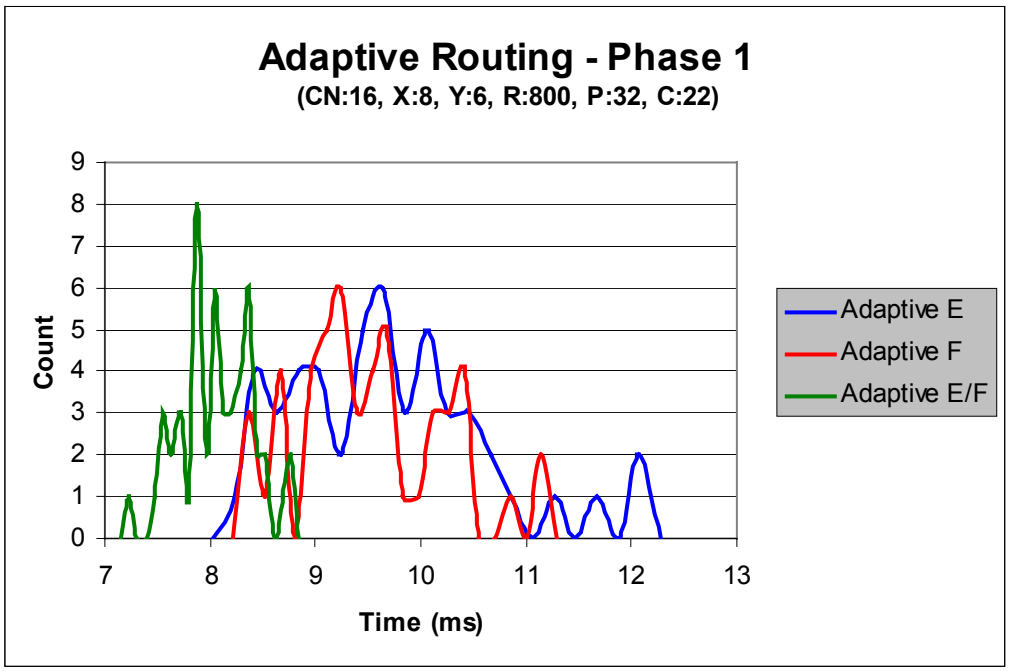

Fig. 7.17: Phase 1 adaptive routing performance metric for a 16 CN ($8x6$) configuration with range: 800, pulses: 32, and channels: 22.

In the second phase of communication, the adaptive E/F routing outperforms the other two adaptive routing options (see Fig. 7.18). The adaptive E/F routing completed, on average, 5 ms faster than adaptive F routing (i.e., approximately a 15% decrease) and 10

ms faster than adaptive E routing (i.e., approximately a 25% decrease). The adaptive E routing completed last. This is primarily due to the two hardware priority arbitration algorithms at the crossbars. Packets entering port F are given a higher hardware priority than those entering port E. For this simulation, a combination of adaptive E/F routing produces the smallest completion times for both phase 1 and 2.

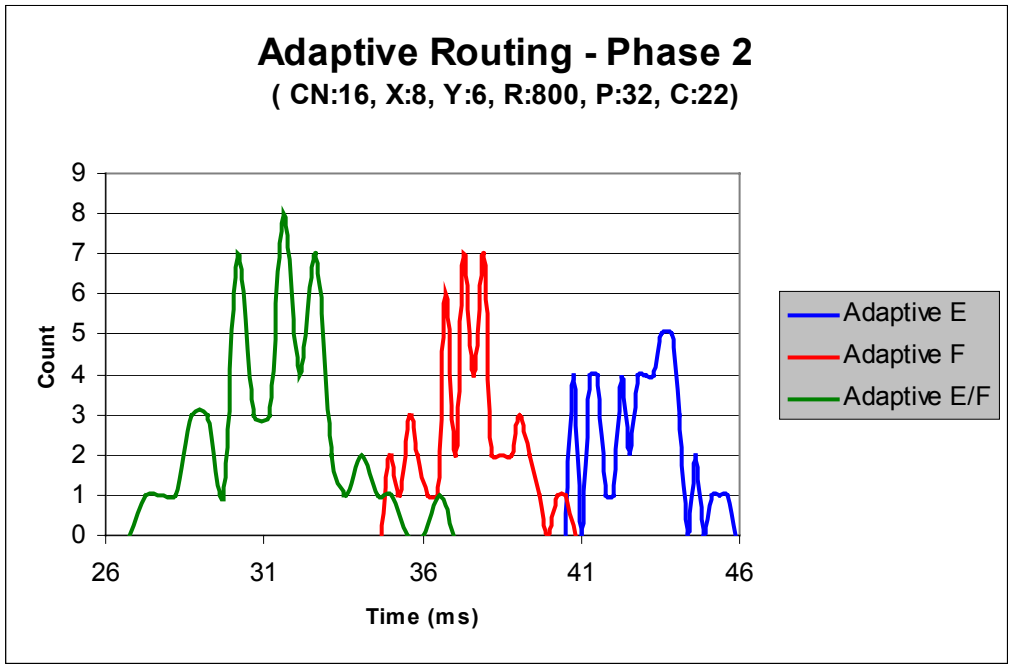

Fig. 7.18: Phase 2 adaptive routing performance metric for a 16 CN ($8x6$) configuration with range: 800, pulses: 32, and channels: 22.

7.3.2 Adaptive Routing Performance Metric 2 for a 16 CN ($8x6$) Configuration

By applying a slight modification to the simulation parameters in Section 7.3.1, different timing results were obtained. In this simulation, the input parameters of the STAP data cube were modified to produce a smaller data sample. In this case, the range samples were reduced to four hundred, the pulses to twenty-two, and the channels to sixteen. As a result of the changes, the adaptive E/F routing approaches the completion times of adaptive E and F routing in both phases (see Figs. 7.19 and 7.20), although the adaptive E/F

combination still records the shortest time for each phase. In addition, the adaptive E routing options continued to account for the longest completion times.

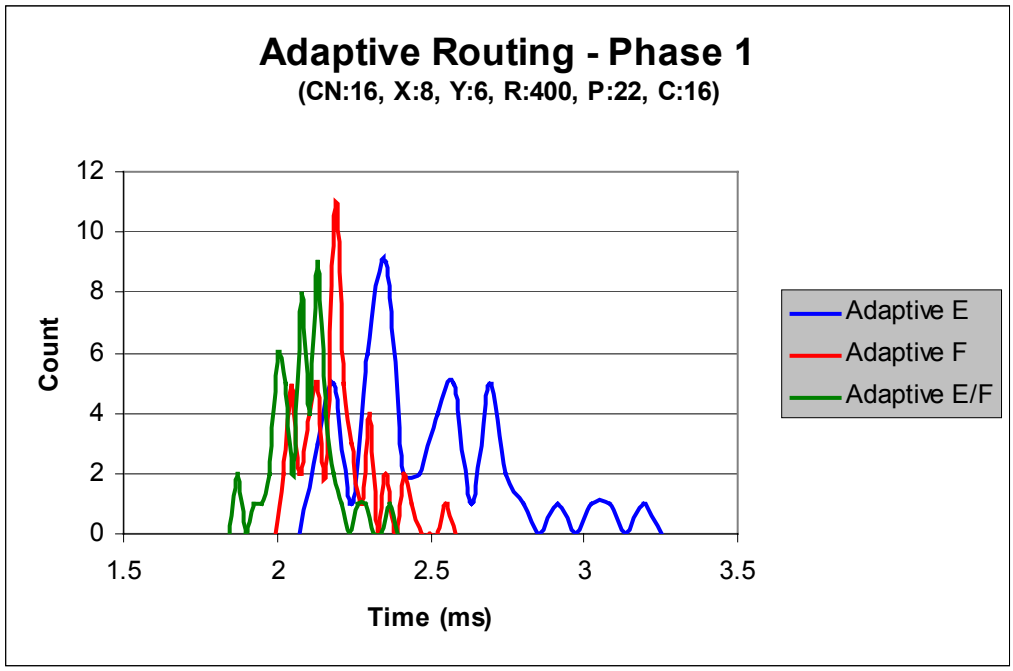

Fig. 7.19: Phase 1 adaptive routing performance metric for a 16 CN ($8x6$) configuration with range: 400, pulses: 22, and channels: 16.

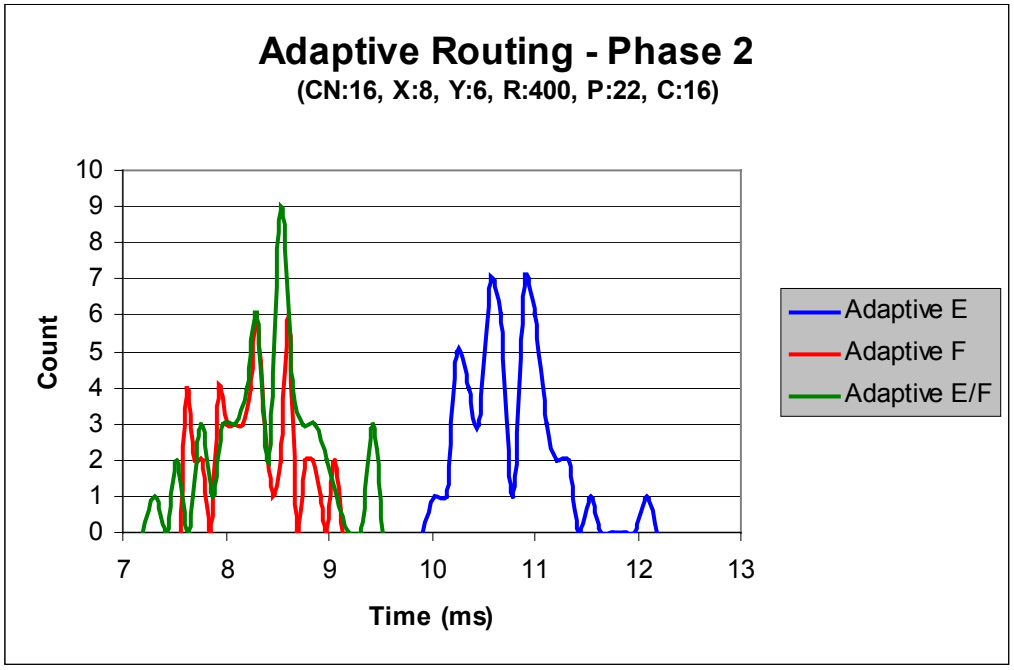

Fig. 7.20: Phase 2 adaptive routing performance metric for a 16 CN ($8x6$) configuration with range: 400, pulses: 22, and channels: 16.

7.4 DMA Chaining Options

Direct Memory Access (DMA) block transfers may be utilized to send multiple packets to the same destination CN. When packets destined for the same location are DMA chained together, only the first packet is assessed the DMA start up time, which is required to start the DMA controller. The remaining packets do not incur a start up cost, and proceed directly to the route arbitration state. Three distinct investigations were conducted related to DMA chaining. Each of the three simulations generate CE traffic. CE traffic was selected because it tends to create more, but smaller messages than CN traffic. Each simulation involved recording both the phase 1 and phase 2 completion times for fifty simulations.

7.4.1 DMA Chaining Performance Metric 1 for a 24 CE ($8x3$) Configuration

In the first DMA chaining investigation, the parameters of the system studied included twenty-four CEs, an $8x3$ process set, two hundred range samples, twenty-two pulses, sixteen channels, and adaptive F first routing. Fig. 7.21 illustrates the timing results collected from the simulator with DMA chaining enabled and disabled. Under these conditions, the DMA chaining option has only a limited effect on the timing results. Disabling the DMA chaining for this scenario achieves the shortest completion time. Furthermore, each option contains a disparity of approximately .3 ms in recorded completion times. The variation is a product of message ordering prior to communication. This alone suggests that the ordering of the messages influences the performance of the communication pattern.

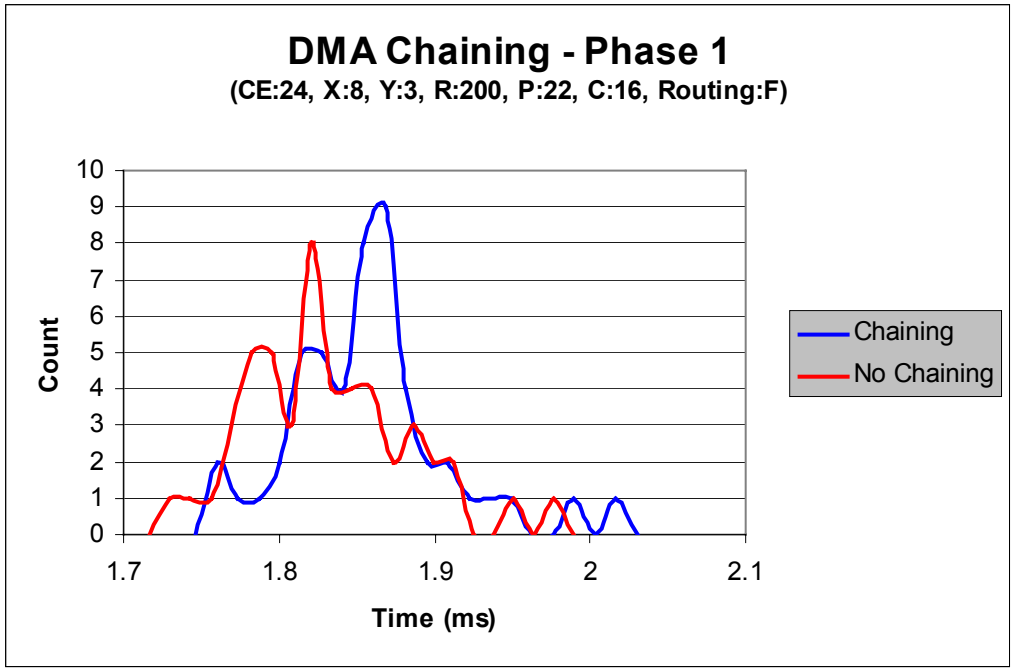

Fig. 7.21: Phase 1 DMA chaining performance metric for a 24 CE ($8x3$) configuration with range: 200, pulses: 22, channels: 16, and adaptive F routing.

The phase 2 communication details yield a similar results (see Fig. 7.22). The overall performance for both DMA chaining enabled and disabled are comparable. In this instance, the shortest possible completion time is recorded by both options. In addition, the average completion time for no chaining is approximately .2 ms better than with chaining enabled.

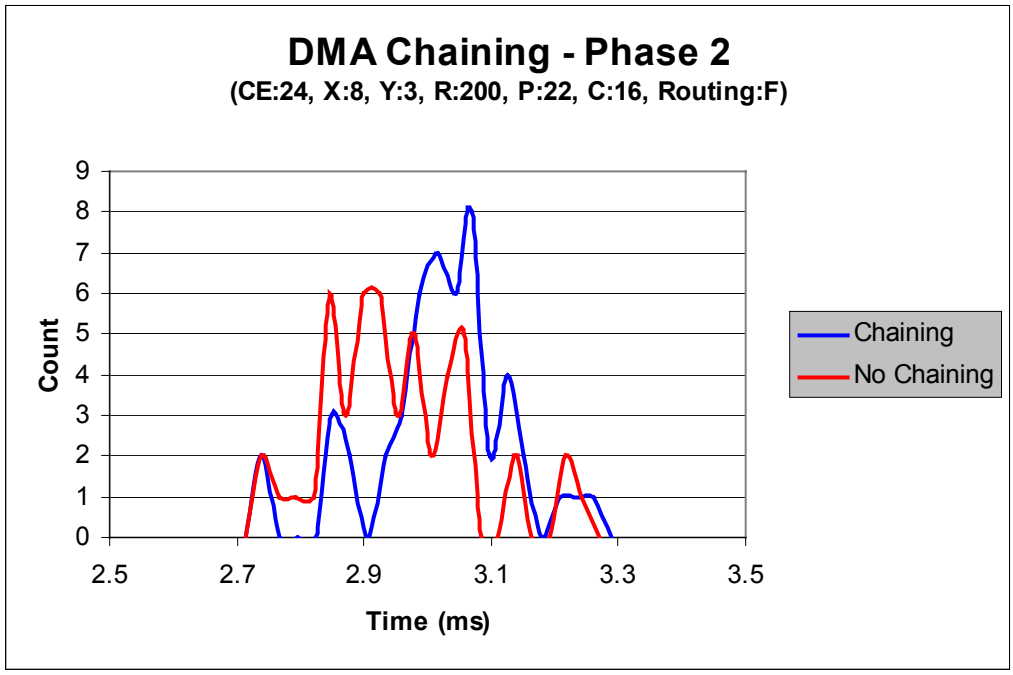

Fig. 7.22: Phase 2 DMA chaining performance metric for a 24 CE (8*x*3) configuration with range: 200, pulses: 22, channels: 16, and adaptive F routing.

7.4.2 DMA Chaining Performance Metric 2 for a 24 CE ($8x3$) Configuration

In section 7.4.1, DMA chaining had only a small effect on the performance of the communication pattern. The second performance metric involved investigating the same hardware configuration (i.e., a twenty-four CE system, an $8x3$ process set, and adaptive F first routing) but with a larger data cube. Increasing the size of the data cube equates to increasing the amount of packets transmitted during corner-turn phases. The results of increasing the data cube range parameter from two hundred to four hundred for both phases of communication are illustrated in Figs. 7.23 and 7.24. In each phase, disabling the DMA block transfers accounted for the shortest completion time. For this simulation configuration, packets chained together tended to occupy the same connection path repeatedly. Consequently, certain packets remained blocked until the entire DMA block transfer was completed. Disabling the DMA chaining yielded a greater diversity of packets successfully arbitrating through the network. Finally, the variation in recorded completion times for each phase signifies the importance of the outgoing order of messages.

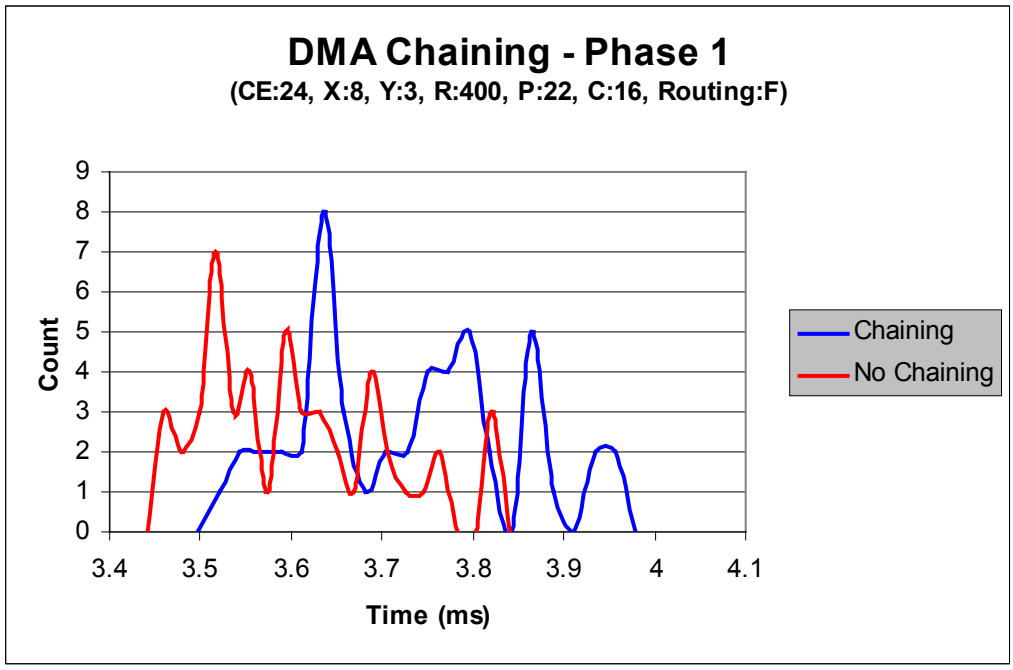

Fig. 7.23: Phase 1 DMA chaining performance metric for a 24 CE ($8x3$) configuration with range: 400, pulses: 22, channels: 16, and adaptive F routing.

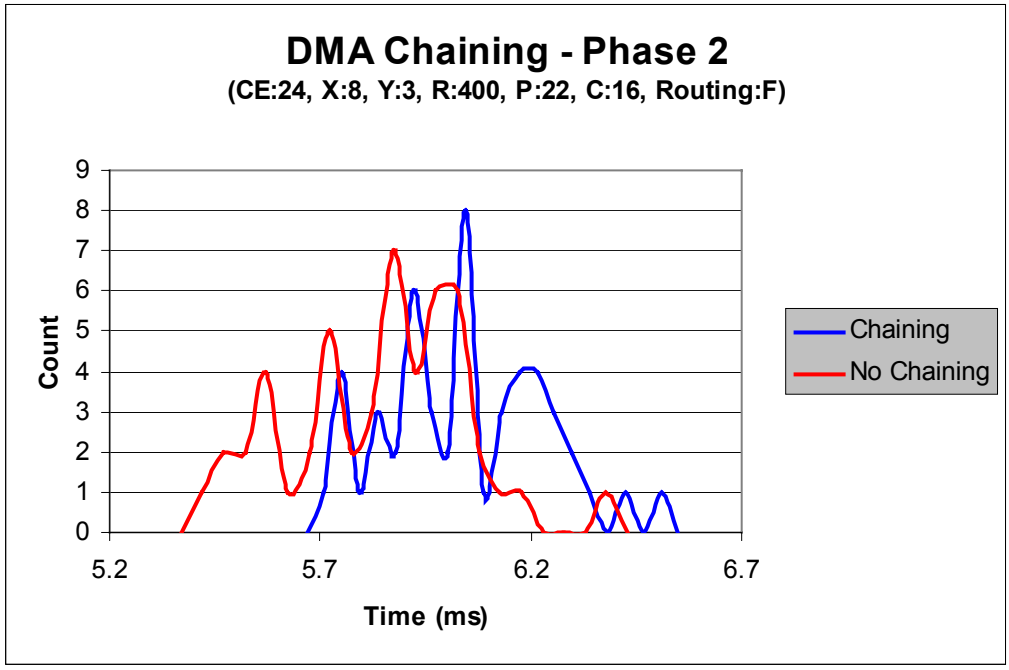

Fig. 7.24: Phase 2 DMA chaining performance metric for a 24 CE ($8x3$) configuration with range: 400, pulses: 22, channels: 16, and adaptive F routing.

7.4.3 DMA Chaining Performance Metric 3 for a 24 CE ($8x3$) Configuration

The third and final performance metric involved investigating the same hardware configuration (i.e., a twenty-four CE system, an $8x3$ process set, and adaptive F first routing) but with a larger data cube size than either of the first two investigations. Again, increasing the size of the data cube equates to increasing the amount of data communicated during corner-turn phases. The results of increasing all three of the data cube parameters for both phases of communication are illustrated in Figs. 7.25 and 7.26. In both cases, DMA block chaining significantly improved the performance of data transfers. The disparity between the completion times illustrates the fact that message order effects the communication performance whether DMA chaining is enabled or not.

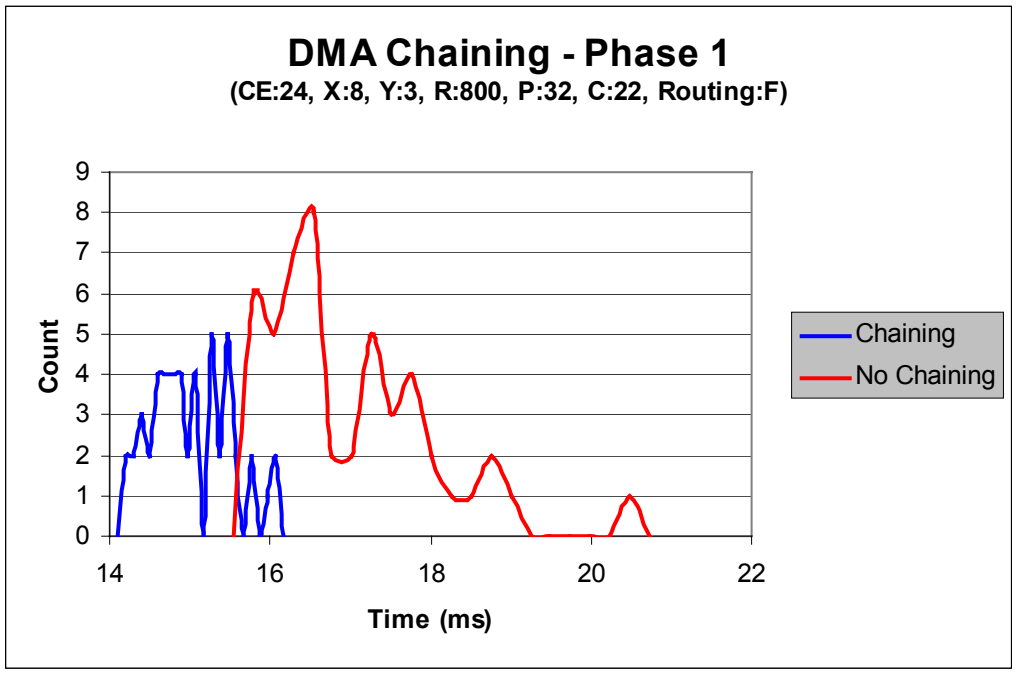

Fig. 7.25: Phase 1 DMA chaining performance metric for a 24 CE ($8x3$) configuration with range: 800, pulses: 32, channels: 22, and adaptive F routing.

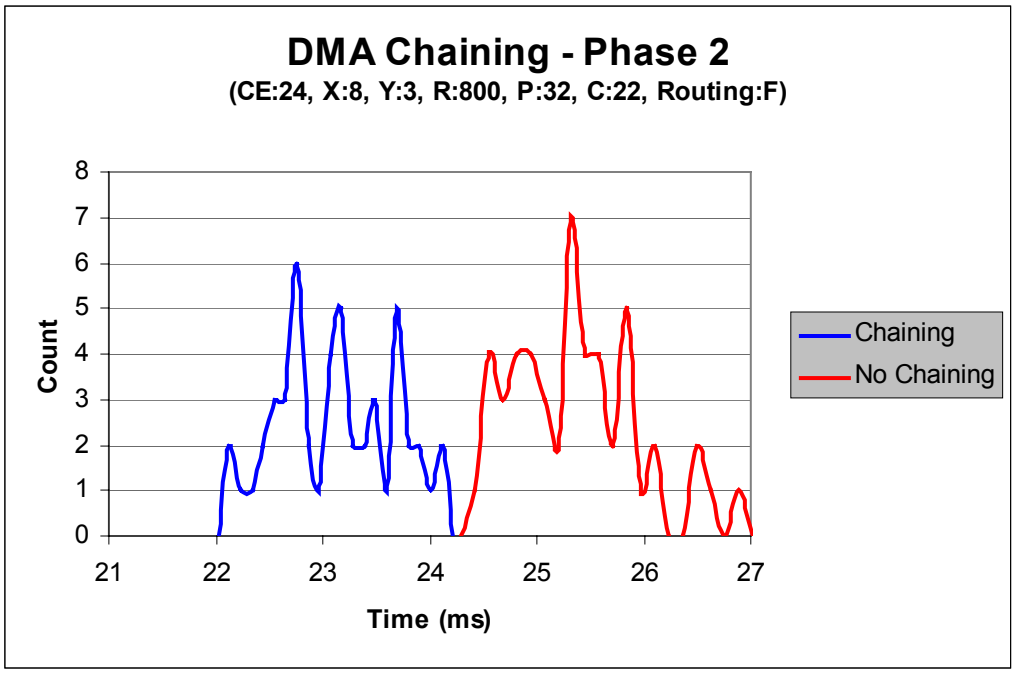

Fig. 7.26: Phase 2 DMA chaining performance metric for a 24 CE ($8x3$) configuration with range: 800, pulses: 32, channels: 22, and adaptive F routing.

CHAPTER VIII

CONCLUSIONS

Achieving real-time performance for STAP algorithms on parallel embedded systems like the Mercury RACE multicomputer, largely depends two major issues. First is determining the best method for distributing the 3-D STAP data cube across CNs, composed of multiple processors, of the multiprocessor system (i.e., the mapping strategy). Second is determining the scheduling of communications prior to Doppler filtering and weight computation and beamforming. In general, STAP algorithms contain three phases of processing, one for each dimension of the data cube (i.e., range, pulse, and channel). During each phase of processing, the vectors along the dimension of interest are distributed as equally as possible among the CNs for parallel processing. In a sub-cube bar approach, before processing can take place at the next phase, the data vectors must be re-distributed to form contiguous vectors of the next dimension.

Determining the optimal communication schedule of queued messages during the two phases of data re-partitioning may be classified as an NP-hard problem. The goal of the research was to model (through simulation) the effects associated with how data is mapped onto the CNs of the Mercury system using a sub-cube partitioning approach and how the data transfers are scheduled.

Chapter VI described the design and implementation of the network simulator for the RACE system. Chapter VII provided numerical studies of a subset of the data recorded from simulation scenarios investigated. In general, five parameters can be modified to produce different results from the simulator. The five parameters are: the data cube size, the process set size, the DMA chaining options, the adaptive routing options, and CN or CE message traffic. Investigating all possible combinations of the above simulation parameters is far beyond the scope of this work. However, the results obtained illustrate the importance a network simulator in investigating the effects of communication on performance.

Although used here to study the communication times for parallel STAP algorithms, the simulator is generic enough to be used to predict communication times for any communication pattern. The simulator is very complex and; its implementation required over 6500 lines of Java code. Future work will involve systematic approaches to parameter selection for optimizing performance.

REFERENCES

[1] Title3 – Executive Order 12931 of October 13, 1994, Section 1., paragraph (d); available Fed. Reg. Vol. 59, No. 199, Monday, October 17, 1994.

[2] K. C. Cain, J. A. Torres, and R. T. Williams, "Real-Time Space-Time Adaptive Processing Benchmark", Mitre Technical Report: MTR 96B0000021, Mitre, Center for Air Force C3 Systems, Bedford, MA, February 1997.

[3] J. L. Eaves and E. K. Reedy, *Principles of Modern Radar*, Van Nostrand Reinhold, New York, NY, 1987.

[4] T. H. Einstein, "Mercury Computer Systems' Modular Heterogeneous RACE Multicomputer," *Proceedings of the Sixth Heterogeneous Computing Workshop (HCW '97)*, sponsor: IEEE Computer Society, Geneva, Switzerland, April 1997, pp. 60-71.

[5] K. Hwang, *Advanced Computer Architecture: Parallelism, Scalability, Programmability*, McGraw-Hill, Inc, New York, NY, 1993.

[6] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994.

[7] B. C. Kuszmaul, "The Race Network Architecture," *Proceedings of the 9th International Parallel Processing Symposium (IPPS '95)*, sponsor: IEEE Computer Society Technical Committee on Parallel Processing, Santa Barbara, CA, April 1995, pp. 508-513.

[8] R. J. Mailloux, *Phased Array Antenna Handbook*, Artech House, Boston, MA, 1994.

[9] G. V. Morris, *Airborne Pulsed Doppler Radar*, Artech House, Norwood, MA, 1988.

[10] *RACEway Interlink Modules*, VITA Standards Organization (VSO), 1994.

[11] A. W. Rihaczek, *Principles of High-Resolution Radar*, McGraw Hill, Inc., New York, NY, 1969.

[12] P. K. Rowe, "COTS Radar and Sonar Systems Solutions," Multiprocessor Toolsmiths Inc., Kanata , ON Canada, 1996.

[13] M. F. Skalabrin and T. H. Einstein, "STAP Processing on a Multicomputer: Distribution of 3-D Data Sets and Processor Allocation for Optimum Interprocessor Communication," *Proceedings of the Adaptive Sensor Array Processing (ASAP) Workshop*, March 1996.

[14] M. I. Skolnik, *Introduction to Radar Systems*, McGraw Hill, New York, NY, 1962.

[15] M. I. Skolnik, *Radar Handbook*, Second Edition, McGraw Hill, New York, NY, 1990.

[16] D. Taylor and C. H. Westcott, *Principles of Radar*, Cambridge University Press, Cambridge and Bentley House, London, 1948.

[17] J. C. Toomay, *Radar Principles for the Non-Specialist*, Second Edition, Van Nostrand Reinhold, New York, NY, 1989.

[18] J. Ward, *Space-Time Adaptive Processing for Airborne Radar*, Technical Report 1015, Massachusetts Institute of Technology, Lincoln Laboratory, Lexington, MA, 1994.

[19] R. S. Pressman, *Software Engineering: A Practitioner's Approach* , Third Edition, McGraw-Hill, Inc., New York, NY, 1992.

[20] The RACE Multicomputer, Hardware Theory of Operation: Processors, I/O Interface, and the RACEway Interconnect, Volume I, ver. 1.3.

[21] G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language for Object Oriented Development," Documentation Set Version 1.1, September 1997.