Data-Intensive Systems Benchmark Suite

Analysis and Specification

Version 1.0

30 June, 1999

Contract number: MDA972-97-C-0025

submitted to:

Dr. José Mu•oz
DARPA / ITO
3701 North Fairfax Drive
Arlington, VA 22203

submitted by:

Atlantic Aerospace Electronics Corp.
470 Totten Pond Road
Waltham, MA 02451

Table of Contents

1 INTRODUCTION ____ 1

1.1 Data-Intensive Systems ____ 1

1.2 Motivation ____ 1

1.3 Goals ____ 1

1.4 Organization of this Document ____ 3

2 BACKGROUND ____ 4

2.1 Algorithm Selection ____ 4

2.2 Algorithm Analyses ____ 4

2.2.1 Method of Moments ____ 4

2.2.2 Simulated SAR Ray Tracing ____ 7

2.2.3 Image Understanding ____ 10

2.2.4 Multidimensional Fourier Transform 13

1.1.5 Data Management ____ 14

3 PROCEDURES ____ 18

3.1 Overview ____ 18

3.2 Benchmarking Procedure ____ 18

3.3 Metrics ____ 19

3.3.1 MoM Benchmark Metrics ____ 21

3.3.2 Simulated SAR Ray Tracing Benchmark Metrics ____ 21

3.3.3 Image Understanding Benchmark Metrics ____ 21

3.3.4 Multidimensional Fourier Transform Metrics ____ 21

3.3.5 Data Management Benchmark Metrics ____ 22

3.4 Measurement Procedures ____ 22

3.5 Submission of Results ____ 23

3.5.1 Required Elements ____ 23

3.5.2 How to Submit Results ____ 24

3.6 Common Data Types ____ 24

3.7 Arithmetic Precision ____ 25

4 SPECIFICATIONS ____ 27

4.1 Approach ____ 27

4.2 Benchmark Specifications ____ 27

4.2.1 Method of Moments ____ 28

4.2.1.1 Input ____ 31

4.2.1.2 Algorithmic Specification ____ 32

4.2.1.3 Output ____ 42

4.2.1.4 Acceptance Test ____ 43

4.2.1.5 Metrics ____ 44

4.2.1.6 Baseline Source Code ____ 46

4.2.1.7 Baseline Performance Figures ____ 46

4.2.1.8 Test Data Sets ____ 46

4.2.2 Simulated SAR Ray Tracing ____ 47

4.2.2.1 Input ____ 47

4.2.2.2 Algorithmic Specification ____ 58

4.2.2.3 Output ____ 67

4.2.2.4 Acceptance Test ____ 68

4.2.2.5 Metrics ____ 68

4.2.2.6 Baseline Source Code ____ 68

4.2.2.7 Baseline Performance Figures ____ 69

4.2.2.8 Test Data Sets ____ 69

4.2.2.9 References ____ 69

4.2.3 Image Understanding ____ 72

4.2.3.1 Input ____ 72

4.2.3.2 Algorithmic Specification ____ 75

4.2.3.3 Output ____ 82

4.2.3.4 Acceptance Test ____ 83

4.2.3.5 Metrics ____ 83

4.2.3.6 Baseline Source Code ____ 84

4.2.3.7 Baseline Performance Figures ____ 84

4.2.3.8 Test Data Sets ____ 84

4.2.3.9 References ____ 84

4.2.4 Multidimensional Fourier Transform 85

4.2.4.1 Input ____ 85

4.2.4.2 Algorithmic Specification ____ 86

4.2.4.3 Output ____ 88

4.2.4.4 Acceptance Test ____ 89

4.2.4.5 Metrics ____ 89

4.2.4.6 Baseline Source Code ____ 89

4.2.4.7 Baseline Performance Figures ____ 89

4.2.4.8 Test Data Sets ____ 89

4.2.4.9 References ____ 90

4.2.5 Data Management ____ 90

4.2.5.1 Input ____ 91

4.2.5.2 Algorithmic Specification ____ 95

4.2.5.3 Output ____ 100

4.2.5.4 Acceptance Test ____ 100

4.2.5.5 Metrics ____ 100

4.2.5.6 Baseline Source Code ____ 100

4.2.5.7 Baseline Performance Figures 100

4.2.5.8 Test Data Sets ____ 101

4.2.5.9 References ____ 101

5 CONTACT INFORMATION ____ 102

6 REFERENCES ____ 103

1 Introduction

As part of the DARPA Information Technology Office's Data-Intensive Systems research program, this document presents a suite of application-oriented benchmarks, and the methodology supporting it.

This section provides an introduction to the effort and the document, including an outline of the motivation for the program, the goals to be sought, and the organization of the remaining sections of the document.

1.1 Data-Intensive Systems

Many defense applications employ large data sets that are accessed non-contiguously. These applications cannot take full advantage of typical memory-access optimizations, and consequently perform at approximately two orders of magnitude below peak rates. Some data-starved applications identified by DARPA/ITO are RADAR cross-section modeling, high-definition imaging, terrain masking, relational and object-oriented databases, structural dynamics calculations, and circuit simulation.

Compounding the problem, memory access speeds have also not grown in pace with storage sizes, nor with processor speeds.

To bolster the above applications and address these problems, DARPA/ITO has launched a Data-Intensive Systems (DIS) effort, which includes two complimentary tasks: (1) incorporate logic within memory chips (processor-in-memory, or PIM), allowing manipulation of data locally in a memory subsystem; and (2) adaptive cache management, to increase cache utilization and improve data flow.

1.2 Motivation

The development of new architectures and approaches to data-intensive computing could be beneficial to many problems of interest to DARPA. Evaluation of the architectures in the context of those problems is essential in order to realize those benefits.

Equally important, the existence of simplified–but meaningful–programs derived from defense applications can provide valuable input to the development process.

Therefore, benchmarking fills a critical need in the development of Data-Intensive Systems. An appropriate benchmarking effort will accelerate insertion of DIS technology into defense systems.

1.3 Goals

The primary goal of this effort is the development of a benchmark suite that can be used to quantify the performance gains likely to be achieved for defense computer programs when implemented using approaches and architectures developed under the DIS program.

Any benchmark specification dealing with early research into new systems must remain architecture-neutral. In support of this goal, the benchmark specifications are essentially only the mathematical description of problems' solutions. Of course, over years of de-

velopment in the context of Von Neumann computer architectures, many known optimizations have been utilized, and an attempt has been made to provide or reference these, so that participants charged with implementing the benchmarks are not faced with having to independently rediscover the optimizations.

Benchmarks that focus on the measurement of relative performance frequently involve implementation only of specific, isolated functions, resulting in accurate measurement of peak performance. This level of performance is rarely realizable in general application, so benchmarks that include the processes of data movement and preparation are desirable for a more generalized measurement of real performance. Considering the variety of architectures under scrutiny in the DIS program, it would be dangerous to presume that these "overhead" functions diminish in proportional resource consumption as data sets grow larger. Therefore, avoidance of isolated tasks as benchmarks is a goal of this program; rather, performance related to the interactions between program components is intended to be included in the measurements.

[Weems], while reviewing lessons from prior benchmark efforts, points out:

> "Having a known, correct solution for a benchmark is essential, since it is difficult to compare the performance of architectures that produce different results. For example, suppose architecture A performs a task in half the time of B, but A uses integer arithmetic while B uses floating-point, and they obtain different results. Is A really twice as powerful as B?"

Therefore, a complete solution with test data sets is considered one of the essential components of the distribution of the benchmark specification.

Although there are sometimes competing ideas about how to best solve a particular problem, the goal of a benchmark is not specifically to solve a problem, but rather to test the performance of different machines doing comparable work. Since DIS architectures are likely to vary greatly, significant latitude is allowed in the implementation of a solution to benchmark problems. However, participants must remain cognizant of the fact that ultimately, the measurements taken must be meaningful in the context of defense problems, and specifically in the context of *relative* gain. So, it is not a goal of this benchmark effort to develop the best solutions for the most difficult problems; rather, it is a goal to employ pertinent solutions to problems expected to benefit from DIS research, and allow enough flexibility to maximize individual performance, yet remain consistent and comparable.

While benchmarks that are too simplistic do not offer valuable results, those that are too complex are never implemented, at least in a meaningful way. Resources are limited, so ease of implementation is a factor of consideration. It is a goal of this program to develop benchmark programs that should require relatively little source code during implementation, yet still offer meaningful results.

Often, high-performance systems are developed that remain under-utilized due to the esoteric or difficult nature of their programming. Therefore, an important goal of this effort is to evaluate the labor costs associated with use of candidate architectures. The ability to handle existing, 'legacy code' is an important consideration, as is the labor cost to exploit the powerful features of these systems.

A program will generally execute faster when its required data set is small enough to fit in main memory, as opposed to when paging or swapping is required. Likewise, when the data set is small enough to fit in cached memory, it will generally execute faster still. Balancing the competing factors of speed, size, and cost is a major engineering decision, and quantifying the effects of that decision is a goal of this effort.

Finally, in support of the primary goal of being able to quantify performance gains, it is a goal of this effort to remain open to any additional information participants wish to supply that will assist reviewers in making an accurate determination. While this document specifies minimum participation requirements, information such as results, analyses, proofs, or additional metrics is hereby solicited.

1.4 Organization of this Document

The remainder of this document is organized as follows:

1. Section 2 provides the foundation for this work, including analyses of the algorithms included in the benchmark.
2. Section 3 outlines the procedures to be followed by participants.
3. Section 4 provides the specifications for the benchmark set.
4. Sections 4 and 5 give contact information and references, for participants needing additional information.

2 Background

Given the motivation and goals outlined above, this section addresses the determination of the content of the benchmark suite. The selection of the algorithms to be included, and analyses of each–showing critical performance bottlenecks and suggesting possible gains–are provided.

2.1 Algorithm Selection

Although many classes of algorithms could benefit from systems with advanced memory or PIM elements, three classes would provide a representative scope of achievable performance improvement for problems of interest to key DARPA programs:

- *Model-Based Image Generation* – This class includes generation of synthetic signatures and scenes for targets and terrain based on complex models of objects and sophisticated camera models for various sensor types. Applications include target recognition, real-time scene simulation for visualization or training, and model-driven change detection.
- *Target Detection* – This class includes spatial- and frequency-domain target detection in scenes collected from a wide range of sensor types. Applications include automated exploitation and cueing systems.
- *Database Management*– This class includes algorithms for index maintenance, storage management, and content-based query processing. Applications include sensor data archive management and geographic information systems such as the Dynamic Database for Battlefield Situation Awareness.

From these classes, five algorithms were selected–two from Model-Based Image Generation, two from Target Detection, and one from Database Management . Analyses of these algorithms which suggest their possible performance gains are included below.

2.2 Algorithm Analyses

Each of the selected applications was analyzed, with a specific interest in the identification of computational bottlenecks in the algorithms and potential performance improvements offered by DIS architectures. Fragments of the algorithms suitable for benchmark implementation were identified, and from these the specifications found in Section 4 of this document. The remainder of this section presents individual analyses of the five selected algorithms.

2.2.1 Method of Moments

The first class of algorithms chosen for inclusion in the DIS benchmark suite are Method of Moments (MoM) algorithms, which are frequency domain techniques for computing the electromagnetic scattering from complex objects. MoM algorithms require the solution of large dense linear systems of equations. Traditionally, MoM algorithms have employed direct linear equation solvers for these systems. The high computational

complexity of the direct solver approach has limited MoM algorithms to low frequency problems. Recently, fast solvers have been introduced which have low computational complexity. The potential of these fast solvers to enable MoM algorithms to solve larger problems at higher frequencies is ultimately limited by the speed of main memory. Thus, fast MoM algorithms may benefit from the Data-Intensive Systems research effort.

In MoM algorithms the integral equation form of the Helmholtz equation is discretized by expanding the surface currents induced by the applied excitation in N basis functions. Then N test functions are used to convert the integral equation to a dense linear $N \times N$ system that takes the form $Z \bullet J = V$. Generally, N increases as the square of the frequency, and for typical problems, N is greater than 10,000. In traditional MoM algorithms, which first appeared in the late1960's, the dense linear system $Z \bullet J = V$ is solved by a direct linear equation solution algorithm, which may be composed as an in-core or out-of-core solver. On modern parallel computers, the direct solvers may be extended to work on shared or distributed-memory architectures.

The advantage of MoM algorithms is that they are exact representations of Maxwell's equations and highly accurate simulations are possible. The disadvantage of the traditional MoM algorithms is that the methods are computationally intensive, especially as the frequency goes up. The computational complexity of traditional MoM algorithms includes $O(N^2)$ integral evaluations to compute the matrix Z and $O(N^2)$ arithmetic operations to solve the system $Z \bullet J = V$ for J. The memory requirement for traditional MoM algorithms is $O(N^2)$. For these reasons, the traditional MoM algorithms are generally used only for low frequency problems. Although traditional MoM algorithms have been highly optimized on a variety of high-performance computing machines, the largest problems solved so far are for N on the order of 100,000.

Recently, new fast MoM algorithms based on fast, iterative linear equation solvers have been introduced. The iterative solvers rely on numerically stable and rapidly converging iteration procedures, such as the preconditioned GMRES method [Saad]. Fast matrix-vector multiply algorithms are used to compute products of the form $Z \bullet X$ used in the iterative procedure. The computational complexity of the fast MoM algorithms is $O(N \log N)$. The memory requirement for the fast MoM algorithms is $O(N)$. This is a remarkable reduction from the $O(N^2)$ computational complexity of the traditional MoM algorithms, and potentially, allows the solution of much larger problems at higher frequencies.

Rohklin [Rohklin-1, Rohklin-2] has introduced new fast MoM algorithms for the Helmholtz equation, which use iterative linear equation solvers and the fast multipole method (FMM) for fast matrix-vector multiplies. To compute products of the form $Z \bullet X$, the Z matrix is not formed or stored, rather the product $Z \bullet X$ is viewed as a field and approximately evaluated by the FMM. The mathematical formulation of the FMM is based on the theory multipole expansions and involves translation (change of center) of multipole expansions and spherical harmonic filtering. The computational complexity of these new methods is $O(N \log N)$ and the memory requirement is $O(N)$.

Building on the FMM approach, Dembart, Epton and Yip [Dembart-1 to 4] at Boeing have implemented a fast MoM algorithm in a production grade electromagnetics code

used by the company for radar cross-section (RCS) studies. Problems for which the number of unknowns is on the order of 10,000,000 have been solved with this code. Boeing's fast solver uses the preconditioned GMRES iterative method, which requires only the calculation of products of the form $Z \bullet X$, combined with a multilevel FMM for fast matrix-vector multiplies.

The potential of the fast MoM algorithms to solve larger problems at higher frequencies, which results from their low computational complexity, is impacted by two memory bottlenecks encountered by fast solvers: low reuse of data and non-unit stride memory access.

We introduce the issue of low reuse of data by first considering the direct solvers used in the traditional MoM algorithms. Since the computational complexity is $O(N^2)$ and the memory requirement is $O(N^2)$ for direct solvers, the ratio of computation to data access is $O(N)$. For typical problems solved by the traditional MoM algorithms, where N is greater than 10,000, data reuse is high. When data reuse is high, cache is an effective tool for enhancing processor performance. The direct solver, in-core or out-of-core, can be organized so that a block of data is placed in cache and then reused from cache. This effective use of cache makes the computer perform as if all the memory is as fast as the cache memory. Similarly, direct solvers can be optimized for shared- or distributed-memory architectures.

For the fast MoM algorithms, where the computational complexity is $O(N\log N)$ and the memory requirement is $O(N)$, the ratio of computation to data access is $O(\log N)$. Indeed, implementation of Rohklin's translation theorems shows that for the translation operations, which are key to the FMM, the ratio of memory access to computation is 3-to-1. Thus, cache cannot be used to enhance processor performance, and the speed of fast MoM algorithms is ultimately limited by the speed of main memory.

In addition to the bottleneck resulting from the low reuse of data, fast solvers based on the FMM face a second memory related bottleneck. The FMM relies on the numerical implementation of spherical harmonic filtering. The filter operates on rectangular arrays of data in three stages. The arrays are accessed first by rows, then by columns, and finally, by rows again. In the second stage, it is necessary to access memory locations that are not consecutive. Thus, the speed of the fast MoM algorithms based on the FMM is ultimately limited by the speed of accessing main memory with non-unit stride.

Fast MoM algorithms, based on efficient iterative linear equation solvers, have the potential to compute the electromagnetic scattering from complex objects at frequencies 10 to 100 times higher than possible with traditional MoM algorithms. As pointed out above, the ultimate potential of these fast MoM algorithms is limited by two memory-related bottlenecks: low reuse of data and non-unit stride. For these reasons we have chosen to use Boeing's fast solver, based the preconditioned GMRES iteration method and the FMM for fast matrix-vector multiplies, as the basis for the *Method of Moments Benchmark*. The key FMM kernels represented in the benchmark are the translation operations and spherical harmonic filtering.

2.2.2 Simulated SAR Ray Tracing

The simulation of Synthetic Aperture Radar (SAR) provides a cost-effective alternative to real data collections. In contrast to deployed sensors systems, whose operational parameters are fixed, computer simulations allow continuous variation of system and scene parameters. They have been used to simulate the performance of hypothetical sensors systems and to predict the signature of targets from a large number viewing angles as well as target signature that are inaccessible. These simulated target signatures have been used to design, test, and have been included as part of ATR systems.

Phenomenological models of targets and backgrounds and their interactions are the theoretical foundation of the computer simulations. For example, both image-domain and phase-history-domain approaches have been used to simulate synthetic aperture radar (SAR). The image domain approach uses a generalization of the physical optics approximation to compute target scattering. Such an approach is very amenable to use with a solid geometry target model sampled by ray casting. The phase history domain approach uses a variety of methods to compute target scattering: physical optics (PO), physical theory of diffraction (PTD), method of moments (MoM), and others. Hybrid implementations of these two methods have also been developed. The SAR simulation method analyzed for this benchmark is based on the image domain approach.

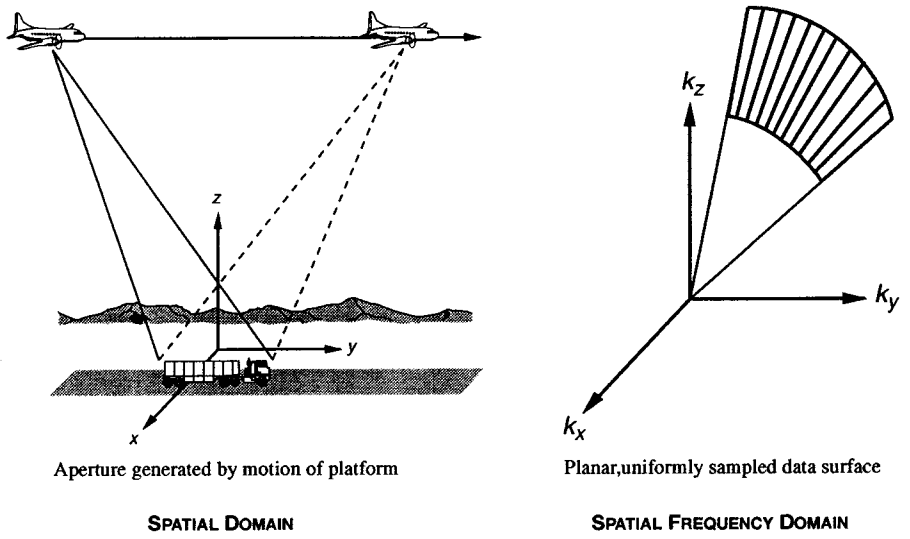


Figure 2.2.2-1. A typical geometry for airborne collection of SAR data relative to a specific ground site.

A typical SAR collection geometry is illustrated in Figure 2.2.2-1. Assume far field conditions and a narrow-band signal. Let α and β denote, respectively, the receive and transmit polarizations of the radar, $u(t)$ the transmitted waveform, and $\gamma(r')$ the so-called SAR reflectivity of the scene, The radar return signal is generally represented as

$$\nu_{\alpha\beta}(t) = K \int_S \gamma_{\alpha\beta}(r')u\left(t - 2\frac{R}{c}\right)ds'$$

where S is the illuminated portion of the scene, R = $|r - r'|$ is the distance from the radar to the point r' on S, c is the speed of light and K is a system constant. In essence, the model is based on the argument that the return from a differential surface element *ds'*, located at r', is a replica of the transmitted signal. This signal is delayed in time by the two-way propagation time from the radar to r' and back, and modified by the reflectivity of the surface element. It can be shown that such a model is consistent with physical optics, and an explicit formula for γ can be obtained.

It is customary to demodulate $\nu(t)$ by mixing with a reference signal $h(t)$, yielding

$$s(t) = h(t)\nu(t)$$

Let $\Gamma(f)$ denote the spatial Fourier transform of $\gamma(r)$:

$$\Gamma(f) = \Im\{\gamma(r)\}$$

A single range record *s(t)* can be interpreted as corresponding to the values of $\Gamma(f)$ over a radial line segment in the spatial frequency domain. The complete record of *s(t)* for a sequence of pulses transmitted and received at positions along the platform trajectory constitute a so-called phase history.

The fact that the collected data corresponds to the Fourier transform of the reflectivity density suggests that a reconstruction (image) of the reflectivity can be obtained by inverse Fourier transformation of the data. It will be at best a partial reconstruction because we have only partial data. For a linear trajectory, the phase history represents a planar surface in the spatial frequency domain over which the value of $\Gamma(f)$ has been sampled. The most that can be obtained is a two-dimensional image. Letting $A(f)$ denote a weighted, two-dimensional processing aperture over the data surface, the SAR image formation process is given by

$$g(r) = \Im^{-1}\{ A(f)\,\Gamma(f) \}$$

Additional insight is gained by noting that the image formation process is mathematically equivalent to the convolution

$$g(r) = a(r) * \gamma(r)$$

where

$$a(r) = \Im^{-1}\{ A(f) \}$$

is known as the spatial impulse response.

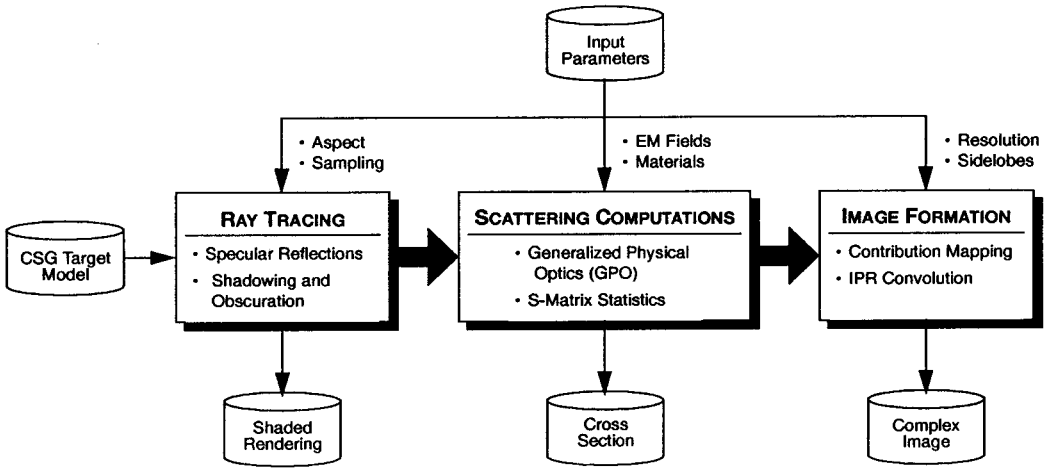


Figure 2.2.2-2. Block diagram of the generalized physical optics SAR simulation.

Simulation of the SAR system can be achieved by synthesis of the phase history followed by aperture weighting and inverse transformation, or by direct computation of the reflectivity density followed by convolution with the spatial impulse response. Both methods have been implemented in SAR simulation programs like X-PATCH and RADSIM.

In the process of analyzing this approach, the simulated SAR technique can be broken down into three steps as seen in Figure 2.2.2-2.

First, is the process of sampling a scene database made of polygons, splines, and Constructive Solid Geometry. A ray tracing system is used to accomplish this sampling, by simulating how the radar energy bounces through the scene.

Ray tracing is a process where rays are fired from a viewing window into the scene and recursively traced through their specular reflections. Each ray is defined as vector with a starting position at the sensor and a direction defined by the pixel it passes through on the viewing window, in this case our synthetic aperture. Each ray is tested against all objects in the scene to see if an intersection can be found. The process of finding the intersection involves finding the roots of a system based on the combination of the vector and object. If multiple intersections are found the closest intersection is used. Once an intersection is found, the object ID and the intersection coordinates are recorded. In addition, several other properties at the intersection point are determined. These are surface normal, curvature, surface material type, and length of the ray from the intersection point to the origin of the ray. Using the surface intersection normal and the incoming ray, a reflected ray is calculated along the perfect specular reflection direction. This ray is fired from the current intersection point and the next intersection is found. This recursive process continues until either the ray leaves the scene, or a preset number of reflections are found. The intersection results of each original ray and all of its reflections create a ray history that contains all the intersection information, normally stored in a linked list. The output of the ray-tracing section is a ray history for each pixel in the image plane.

In programs like X-PATCH, the ray-tracing portion of the process consumes 50% to 60% of the total computation time. With this being the major time component in the SAR simulation process, it is a prime candidate for parallelization. Parallel ray tracing has been investigated by several researchers and is not a simple problem. This process will be the major thrust of the benchmark effort for simulated SAR imagery.

The second step is the process of converting the ray-traced information, the ray history, into the electromagnetic (EM) response of the sampled scene data. Here each ray path is analyzed to generate a fully polarimetric EM response solution. This is a linear process and does not consume a large amount of time. This step, in the SAR simulation, would be a trivial process to parallelize because each ray history is independent of all the others. Due to the small amount of time and the simplicity of parallelization, this portion of the process is not considered as part of the benchmark.

The final step in the simulated SAR process is converting the 2-D array of EM responses into complex images. This is accomplished by mapping the 2-D array of EM responses into the slant plane. This slant plane image is then convolved with a system Impulse Response (IPR) to form a complex image that can be detected and viewed.

This final step is a unique combination of processes, from the viewpoint of parallelization, and does present the second-highest consumer of CPU time. Creating a parallel version of this section of the process will stress data-passing, as EM responses are mapped onto a rectangular grid called the *slant plane*. This output then runs through a standard convolution. Each of these steps will require different lay-outs of memory and should present some unique problems as a parallel implementation. For this reason, and because this step is a large time consumer, it is part of the simulated SAR benchmark.

2.2.3 Image Understanding

Image processing algorithms represent the third type of algorithm chosen for study. The applications of interest include target detection and classification. A sampling of these algorithms was chosen for this benchmark identifying bottlenecks that are common to image processing applications. The sampling contains algorithms that perform spatial filtering and data reduction. The algorithms selected for the benchmark are a morphological filter component, a region of interest (ROI) selection component, and a feature extraction component. These form the Image Understanding Sequence as shown in Figure 2.2.3-1. The morphological filter component provides a spatial filter to remove background clutter in the image. Next, the ROI selection component applies a threshold to determine target pixels, groups these pixels into ROIs, and selects a subset of ROIs depending on specific selection logic. Finally, the feature extraction component computes features for these selected ROIs.

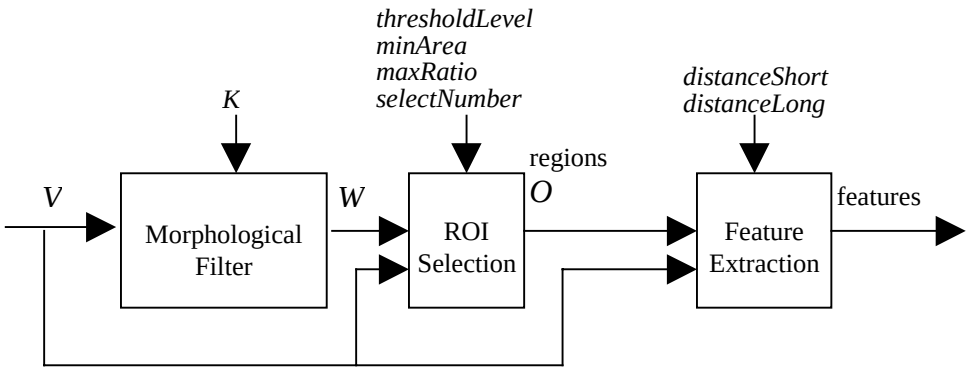


Figure 2.2.3-1: Image Understanding Sequence

Transformations that generate images from symbolic input, as well as Fourier Transforms, were excluded, since these are addressed in other portions of the Benchmark Suite.

The input required by the sequence is a set of parameters and an image, *V*. The first step in the sequence is a spatial morphological filter component generating image *W*. Then, the ROI selection component applies a simple threshold and groups connected pixels into ROIs (or targets) contained in image *W*. This component then computes initial features for each ROI in image *W*, and selects candidate ROIs, depending on the values of these features. These selected ROIs are stored in object image, *O*. The initial features for each selected ROI are stored in the list *regions.* Lastly, the feature extraction component computes additional features for the selected ROIs. The final output is a feature list, *features*, containing all the features calculated for each selected ROI. Details regarding the sequence can be found in Section 4.2.3.

Each algorithm has two associated costs: *operational* and *pixel addressing*. The operational cost is a measure of the computational burden placed upon the processors to execute the algorithm, and pixel addressing cost is a measure of the amount of memory usage or access that is required. A brief description and analysis of each component, including its bottlenecks, follows.

The morphological filter component chosen for the benchmark is a relatively straightforward procedure, designed to remove background clutter and retain objects of interest. The total cost of the morphological filter is determined by assuming the kernel is applied over the entire input image, although in practice the kernel is usually only applied over a subset of the image (the input image less a portion at the edges). The address-to-operation ratio is approximately the same for each approach. The filter utilized in this benchmark includes three distinct phases: erosion, dilation, and difference. The number of operations for the filter is

$$\text{size}(V)[4\,\text{size}(K)+1]$$

where V is the input image, K is the kernel, and size(X) is the total number of pixels in X. The operational cost consists of two multiplies, one subtraction, one minimum comparison, and one maximum comparison. The number of pixel addresses is

$$\text{size}(V)[4\,\text{size}(K)+5]$$

where the kernel and input image are accessed multiple times as the kernel is applied over the input image. The address to operation ratio is then

$$(4\,\text{size}(K)+5)/(4\,\text{size}(K)+1)$$

which is bounded in the range {1, 1.8}.

The ROI selection component of the sequence involves a threshold phase, a connected-component phase (where detected pixels are grouped into objects), an initial feature extraction phase, and a selection phase (where ROIs are selected based on the values of the initial features). The initial feature extraction phase measures five characteristics of the object region. Three of these–*centroid, area,* and *perimeter* –are descriptive of the shape and location of the ROI. The other two–*mean* and *variance* –are statistical measures of amplitude over the pixel population of the ROI. The threshold phase has an address-to-operation ratio of two. The operational and pixel addressing costs associated to the connected component phase, the initial feature extraction phase, and the selection phase vary greatly, depending on the implementation and the data involved, so no analysis of these costs is provided here.

After selecting ROIs, additional features are calculated. These give a rough measure of the texture of each ROI. As discussed in [Parker 97], a gray-level co-occurrence matrix (GLCM) contains information about the spatial relationships between pixels, by representing the joint probability that a pixel with a given value will have a neighboring pixel at a particular distance and direction with another chosen value. Since this matrix is square, with dimensions equal to the number of possible pixel values, it provides more information than can easily be analyzed. Statistical descriptors of the co-occurrence matrix have been used as a practical method for utilizing these spatial relationships. Furthermore, [Unser] designed a method of estimating these descriptors without calculating the GLCM, instead using sum and difference histograms. The descriptors chosen as features for this benchmark are *GLCM entropy* and *GLCM energy*, and are defined in terms of a sum histogram, *sumHist,* and a difference histogram, *diffHist.* These descriptors are calculated for each of two distances and four directions.

It is typical in target detection systems to calculate many features to be used in a target recognition step. The ideal is to choose the fewest and cheapest features possible that provide the best detection result. The cost for the feature extraction component is dependent upon the number of features or targets present in the input image which can range from zero to several thousand in typical applications. This makes the algorithm very difficult to execute efficiently, since many features will have a high computational cost with a small memory access cost, while a few will have a low computational cost with a high memory access cost. Thus, an *a priori*-implementation for feature extraction is generally not possible. Consequently, there is no analysis provided here of the cost involved to calculate these features.

The two main bottlenecks which occur in typical target recognition applications are the result of manipulations of large amounts of data while expending little computational effort, and of smaller amounts of data in computationally intensive functions. The intent of

this benchmark is to represent these bottlenecks within the sequence, so that attempts to remove these bottlenecks may be examined.

2.2.4 Multidimensional Fourier Transform

The Fourier Transform has wide application in a diverse set of technical fields. It is utilized in image processing and synthesis, convolution and deconvolution, and digital signal filtering, to name a few. In fact, the transform is utilized within both the Ray-Tracing and Method of Moments benchmarks described elsewhere in this document. However, special interest in the Fourier transform merits its independent inclusion in this benchmark suite. Specifically, the interest is in the nature of the memory access patterns, which are indicative of a large class of problems.

The multi-dimensional Discrete Fourier Transform (DFT) is defined as

$$F(n_1, n_2, \ldots, n_N) = \sum_{k_N=0}^{N_N} \cdots \sum_{k_1=0}^{N_1} e^{2\pi i k_N n_N / N_N} \cdots e^{2\pi i k_1 n_1 / N_1} f(k_1, k_2, \ldots, k_N) \qquad (2.2.4.1)$$

where f is an input complex multi-dimensional array of size $N = N_1 \times N_2 \times \cdots \times N_N$, and F is the output forward transform of f. The Fourier Transform is rarely implemented directly as Equation 2.2.4.1, since the process would require $O(N^2)$ operations. Instead, the transform can be accomplished in $O(N\log_2 N)$ operations, or less, using one of a series of methods generically called Fast Fourier Transforms (FFT). These FFT methods exploit one or more mathematical properties of Equation 2.2.4.1 to reduce the required number of operations.

The bottleneck associated with DFTs that is of interest here is the non-unit-stride memory access associated with the transform. Part of the subscripts of Equations 2.2.4.1 can be "pulled out" of the summations (i.e., the exponential with the subscript k_N can be pulled outside of the sum over k_{N-1} etc.), which shows that the multi-dimensional DFT can be represented by a series of one-dimensional DFTs:

$$F(n_1, n_2, \ldots, n_N) = F_N\left(F_{N-1}\left(F_{N-2}\left(\cdots F_1\left(f(k_1, k_2, \ldots, k_N)\right)\right)\right)\right) \qquad (2.2.4.2)$$

where F_k is a one-dimensional DFT over the specified index. The aspect of Equation 2.2.4.2 to note is that for whatever memory access the inner loops attempt, the outer loop will always be "opposite" or irregular, which prevents a unit-stride access. Rearrangement of the summations or manipulation of the equations can alleviate this memory access bottleneck to some extent, but some non-unit-stride access is present with most DFT implementations.

In order to simplify the implementation and specification of this benchmark, the DFT is limited to three-dimensional transforms only. The implementation of a 3-D transform is complex enough to give an indication of the performance of the architecture on higher dimensional transforms, but simple enough to be relatively easy to implement. The inclusion of one- or two-dimensional transforms would not significantly add any other performance information regarding the candidate architectures. In addition, one- and two-

dimensional input can be approximately tested by specifying the length of the remaining dimensions of the array to be one.

2.2.5 Data Management

The fifth area in which the DIS benchmark suite attempts to measure performance improvement is in data management, specifically in the area of DBMS. Applications for traditional DBMS have been dominated by archival storage and retrieval of large volumes of essentially static data. Some newer applications, such as the Dynamic Database for Battlefield Situation Awareness, demand management of complex, dynamic indices in addition to the data.

The objective of this benchmark is to measure the performance improvement of a given hardware configuration for certain elements of traditional DBMS processing. Performance improvements due to sophisticated database design or special software implementation are avoided and not intended to be part of the benchmark. This benchmark focuses on two weaknesses of conventional DBMS implementations: index algorithms and ad hoc query processing.

Large volumes of data in a DBMS are typically referenced by an index structure. The index can be used instead of brute-force searches over all the data when a query is made. The index defines one or more elements of the data entries as key values. Thus, the key values are specified in advance, and the DBMS maintains a separate index structure based on them. The index is used to accelerate query processing by minimizing the amount of data that must be accessed to satisfy the query.

Two assumptions typical of conventional algorithms are that the data will be predominately static, and that operation can be suspended for index maintenance. Neither assumption holds for the Dynamic Database or other dynamic information systems, and current applications drive standard indexing schemes into frequent wholesale index regeneration, yielding unacceptable performance.

The index structure allows efficient searches over a database when the query can use a pre-defined key value. Queries which do not use a key value are called ad hoc, non-key, or content-based queries. This query type requires a brute-force search over all database entries. Conventional applications usually process an ad hoc query in two stages: an index-based search is used for the index keys in the query formulation, if any, and brute-force search is performed on the results of the index-based search. These brute-force searches are a bottleneck in a typical DBMS. The performance impact of non-key queries can be reduced by parallel searches of the data, which may be applicable to specific hardware architectures, or by partitioning the data.

Partitioning schemes provide an additional performance boost for a general database design where the primary objective is to separate areas of the database into logical sections, each of which is then indexed by its own scheme. The partition allows more efficient searches, when the sections have been chosen well, or when an optimal scheme is known *a priori*. It also supports parallel searches across partitions.

Bottlenecks traditionally associated with DBMS primarily occur in query processing, and the majority of work done to enhance performance has been in this area. Much of this

query optimization has increased the query response speed at the expense of maintaining the index over the lifetime of the database. By definition, an index requires an increase in overhead or up-front processing in favor of quicker, cheaper searches. Typical command operations such as *insert* and *delete* have generally not been optimized. This reiterates the implied assumption of the existence of periods during operation when user interaction can be suspended to deal with index management. The cost associated with index management over the operation life of the database represents a new measure of performance for advanced data management applications, and a corresponding new bottleneck.

The indexing method chosen for use within this benchmark is an R-Tree structure. The R-Tree index allows the key to represent spatio-temporal data, which makes the R-Tree particularly applicable to geographic information; it is commonly supported by database vendors. The R-Tree structure is as close to a de-facto standard for representing such data in a database context as exists today.

The R-Tree index is a height-balanced tree containment structure, that is, nodes of the tree contain lower nodes and leaves. Thus, the tree is hierarchically organized and every level in the tree provides more detail than the previous level. The indexed data object is stored only once, but because of the containment structure, keys at all levels are allowed to overlap. This may cause multiple branches of the R-Tree to be searched for a query whose search index intersects multiple nodes.

A general measure of index maintenance cost for separate command operations is the number of node accesses required for each operation. Other measurements of cost become increasingly software-dependent, and are avoided in this analysis. A generic R-Tree implementation, which is given later in this document, has three command operations to measure: insert, delete, and query. Because the R-Tree is a height-balanced structure, the total number of paths for a full tree is given by:

$$N = \sum_{k=1}^{h} 2^{k-1} F$$

where N is the number of paths, h is the height of the tree, and F is the *fan* or *order* of the tree. Traditional performance measures have focused on the query response: for the generic R-Tree the minimum number of node accesses is h, which is expected from a height-balanced tree, and the maximum number of node accesses is N, or a complete node search over all possible paths. The maximum number is unique to the R-Tree or similar overlapping index trees and represents a significant bottleneck. The problem is exacerbated for improperly managed index structures, and can be alleviated by efficient software implementations and improved hardware architectures which allow more efficient or parallel searches.

Index management over the operation of the database represents a new type of bottleneck for advanced applications. The cost of maintaining the index can be estimated in the same manner as for query commands, by determining the number of node accesses required to complete the command in both the best and worst cases. A descriptive estimate of the average case is also given, with the caveat that the average case is highly implementation-dependent, and will vary for each system.

The insert operation has three separate phases: a search over all paths, insertion (which may cause node splitting), and index key adjustment. The best case occurs when insertion does not require node splitting and no parent keys need to be adjusted; this yields a cost of N node accesses. The worst case does require splitting along each parent, and all parent keys are adjusted; this yields a cost of $N+2h$ node accesses. An average insert would tend to require parent key adjustment and periodic node splitting. Thus, the average insert cost would tend towards the maximum cost.

The delete operation has two phases: a search for the data to be deleted, and a possible key readjustment. The best case has a cost of h node accesses, which represents no key adjustments and an immediate "one" path search for the data. The worst case has a cost of $N+h$, which represents a full search of the data and an all parent key adjustment. The average cost of a delete operation tends to the minimum case, since the operation would include key adjustment but probably not a full search.

The costs of the insert and delete operations are greater than or equal to the query operation in both the best and worst cases. Thus, index management over the operational life of the database represents a significant performance bottleneck when the data is dynamic.

This benchmark has been developed to measure the performance improvements of new hardware architectures for both index maintenance and non-key queries, which represent the two significant performance bottlenecks. One goal is to remove or "level" the algorithmic component over all of the architectures, without preventing any new or unique software implementations that would allow a significant performance improvement due to exploitation of special hardware features. This is done by defining the benchmark as the implementation of a highly simplified database with a specific index structure. The database supports only three simple aggregate data objects whose primary difference is in size. The use of different sizes of data objects is intended to prevent optimization of the implementation for an object of a specific size, and the sizes themselves were chosen to prevent similar multiples. The objects are aggregate in that they contain a set of data attributes or parts which are linked together as a list. An ad hoc query uses an attribute of the object for non-key searches. This type of search with simplified objects is relatively simple to implement, but is representative of more complicated database behavior such as object traversal. This benchmark requires the use of the R-Tree structure, but the participant is encouraged to modify or develop additional implementations tailored for new architectures.

The DIS benchmark metrics provide a measurement of the candidate architecture's ability to handle the "highest" load when the number of users is large and the system resources are taxed to their limits. The benchmark simulates this maximum resource utilization by issuing the index commands in a batch rather than a stream mode. A stream mode would more closely mimic a "real" DIS application, allowing for multiple users and possible "down" time for index maintenance. However, this benchmark is primarily interested in the extreme condition, where down-time, in which a database can perform index maintenance with no cost to the users, is assumed not to exist. The performance on successful completion of the entire data set with its multiple commands is the primary metric of this benchmark, and this must include the time required for index maintenance since this will directly affect the users under extreme conditions. Participants are allowed to introduce

artificial lags to the command input to simulate a stream mode, but the times reported for individual command completion and overall set completion must include the added lag times.

3 Procedures

This section provides information on the procedures to be used when employing these benchmarks. The primary purpose of the section is to answer the question, "How does one use this benchmark?" It begins with an overview, then describes the procedures which are common across all four benchmarks in this set. Metrics, measurement, reporting, data types, and precision are also addressed.

3.1 Overview

Procedures are a critical element of benchmarking. To be useful, benchmarks must be approached uniformly and analyzed consistently. According to [Honeywell], the following questions must be addressed by benchmarking procedures:

- How should baseline performance metrics be established, against which to compare the other results?
- How should the operations in the benchmark specification be performed?
- How can it be ensured that an implementation is solving the intended problem?
- How should measurements be made?
- How should benchmark results be reported so that anyone examining the results has sufficient information to interpret them correctly?

3.2 Benchmarking Procedure

The following procedure should be executed by any group or individual wishing to generate an implementation of this benchmark set. The sequence was published in [Honeywell], and applies to all five of the benchmarks in the set. It is presented here modified only to the degree necessary for relevance to this set.

Step	Action
1	Review the background and all procedures as specified in this document. These capture the common aspects of all benchmarks.
2	Review the benchmark specifications, as given in Section 4 of this document, noting any restrictions associated with the benchmark development activity. Understand the metrics of interest, the acceptance test, and the information that needs to be reported.
3	Develop a benchmark implementation in accordance with the benchmark specification. This step can take one of two forms: • simple compilation, with no source code modification, in the case of an 'un-optimized' code test, or • manual optimization, to the degree desired by the participant. In this case, the specific steps or operations to be performed are particular to the benchmark being implemented. A precise description of the steps

	is provided in each respective benchmark specification. Any restrictions regarding the steps are also provided. For both cases, baseline source code, written in the C programming language is provided.
4	Tabulate any information required by the Metrics portion (Section 3.3) of this document.
5	The process used to measure benchmark metrics is important in the interpretation and reproducibility of results. Section 3.3.4 provides a description of the allowable techniques for measurement. Timing must be performed as specified in this document. Benchmark runs should be repeated a sufficient number of times as to ensure reproducibility of results.
6	The use of acceptance tests is critical to determining whether a specific implementation is deemed valid. For each benchmark run, examine the results and verify that they pass the appropriate acceptance test as defined in the benchmark specifications (Section 4).
7	Reporting of results is a weakness of many existing benchmark approaches. Section 3.5 addresses the topic of results reporting in detail, providing guidelines regarding what information needs to be provided to ensure that adequate interpretation of the results is possible.

Participants are required to measure performance of all five benchmarks using 'unoptimized' code. That is, the baseline code provided for each benchmark must be compiled without any modification, and run 'as-is' to establish performance of the architecture utilizing only automatic optimizations. In addition, participants are encouraged to modify or replace this baseline source code and run the tests again, establishing performance after manual optimizations. This should be done for one of the benchmarks at the very minimum. This process may be repeated as desired, but users are reminded that each level of optimization must be documented in accordance with Section 3.5.1.

Therefore, the above procedure must be followed at least six times–once for each 'unoptimized' benchmark, and once for one 'manually optimized' benchmark. There is no limit to how many times it may be followed. However, the ability of the reviewers to effectively analyze the results must be considered.

3.3 Metrics

Of primary interest for all the benchmarks in this set is the trade-off between 'performance' and 'cost', where performance is focused mainly on maximizing throughput, and cost is focused mainly on programmer labor costs. Of course, there are many other important considerations relating to performance and cost; some important contributing factors are listed here:

Performance	Cost
Maximize throughput (primary)	Minimize programmer labor (primary)

Maximize scalability	Minimize development labor
Minimize power consumption	Maximize use of OTS parts
Maximize robustness	Minimize part count
Maximize implementation flexibility	Maximize ability to retrofit
	Minimize volume and weight

To quantify these factors, metrics are specified for each benchmark. These are only concerned with the performance aspect of the trade-off. It is expected that the cost aspect will be addressed in the *Architectural Description* and *Comments* portions of the reports. While each benchmark will require different measurements of performance, all metrics are intended to quantify throughput. How these measurements vary over the range of input data sets gives some notion of scalability for a specific configuration. How the configuration itself can be scaled is another issue to be addressed in the *Architectural Description*.

Implicitly, the implementations are expected to provide correct output to even be considered. This requirement is an element of each acceptance test, and is therefore not a metric in need of evaluation.

The energy spent by implementers laboring in the development of each benchmark implementation is of special interest. As this is ultimately difficult to measure accurately, reviewers will rely on participant's candid reporting on this subject. A frank summary of the required skills, labor expended, and problems encountered during the process would be of great benefit to those establishing the utility of a given design.

Although peripheral devices vary greatly in access speeds and communication throughput rates, it is desirable to understand the limitations on throughput induced by the architecture. Therefore, for all benchmarks, the time spent reading from input and writing to output should be included in time-for-completion metrics, but recorded separately where possible. Participants should comment on I/O features or limitations in their *Architectural Descriptions*.

Power consumption is also an important metric. Again, the early stages of development under the DIS program might make accurate quantification of power consumption impossible. However, participants are expected to include their best estimates of the power required for each benchmark in the suite. Measurement methods employed should be detailed in the report, as it is anticipated that no specific methodology may reasonably be imposed.

Although scalability with respect to problem size is largely addressed by the spectrum of input sets provided with the benchmark, scalability with respect to processor or memory configuration can only realistically be addressed by qualitative analysis. Participants should address this issue for each benchmark in their reports.

Additional metrics for each of the benchmarks are described in the following subsections. This information is also included in the benchmark specifications (Section 4). These, and the above basic metrics should be considered the minimum required for a report. It is in

the participant's interest, however, to supply a complete report, with all the details relevant to evaluators concerned with DIS applications.

3.3.1 MoM Benchmark Metrics

The metrics for the *Method of Moments Benchmark* consist of three measures: performance, scalability with respect to problem size, and scalability with respect to processors. The metrics are described in detail in Section 4.2.1.5.

The most important of the three metrics is performance, which is measured in wall-clock time. The primary measure of performance is the total wall-clock time for the calculation of the far-field by the multilevel FMM. The secondary measure of performance is the breakdown of the total time into the total time for all translation operations and the total time for all spherical harmonic filtering/synthesis calculations. The tertiary measures of performance are the breakdown of the secondary measures into the total time for each of the six steps in the multilevel FMM as specified in Section 0.

3.3.2 Simulated SAR Ray Tracing Benchmark Metrics

The primary metric for the simulated SAR ray-tracing benchmark suite will be total execution time. Secondary metrics will be scalability (how does adding more processors effect the timings and how do larger data sets effect the timings) and load distribution (how is the workload distributed among the processor). These secondary metrics are important measures for the ray tracing part of the benchmark. The major problem with parallel implementations of ray tracing is in achieving a constant work load and maintaining scalability.

3.3.3 Image Understanding Benchmark Metrics

The primary metric associated with the image understanding benchmark is total time for accurate completion of a given input data set. A series of secondary metrics for the individual times of the processing operations is also required. See section 4.2.3.5 for more detail.

3.3.4 Multidimensional Fourier Transform Metrics

There are three metrics for this benchmark. The first, and primary, is the total time required to complete the input set. This should include the time for each transform test as well as the I/O time required to load the randomly generated input, and output the result. The total time should not include the time necessary for the generation of the random data. The second metric is the time required to complete the individual transform tests. Again, this time should include any I/O time for loading of data and output of results. The third metric measures the "mflops" [Johnson] of the individual transform tests. The "mflops" for a given transform is defined to be

$$"mflops" = \frac{5(X \times Y \times Z)\log_2(X \times Y \times Z)}{(\text{time for one DFT in } \mu\text{s})}$$

where X, Y, and Z are the lengths of the first, second, and third dimensions, respectively. The rational behind using this metric is to provide a reasonable comparison between different architectures, implementations, and transform sizes. Note that "mflops" are not equivalent to MFLOPS (millions of floating-point operations per second), but are instead an estimate of that value that assumes a common baseline number of operations for any implementation as

$$5(X \times Y \times Z)\log_2(X \times Y \times Z) + \vartheta(N)$$

which is the radix-2 Cooley-Tukey FFT[Cooley]. This third metric is common in the FFT literature and for more discussion of the reasoning behind the metric, the reader is referred to [Johnson].

3.3.5 Data Management Benchmark Metrics

The primary metric associated with the Data Management benchmark is total time for accurate completion of a given input data set. A series of secondary metrics are the individual times of the command operations: *Insert*, *Delete*, and *Query*. Best, worst, average, and standard deviation times should be reported for all operations for each data set.

The time for a non-response command operation to complete is defined as the time difference between the time immediately before the command is placed in the database input queue and the time immediately before the next command is placed in the same input queue. This time difference is essentially the rate at which each line of the input data set is read and executed. This definition is applied to the *Insert* and *Delete* command operations. The time for a *Query* command operation to complete is defined as the time difference between the time immediately before the command is placed in the input queue to the time immediately after the response is placed in the output queue.

3.4 Measurement Procedures

When measuring performance of a benchmark implementation, the following considerations must be made:

- Actual platform measurements are preferred over simulated results. It is understood that early iterations through the benchmarking process will necessarily be based on simulation, but these must give way to measurements of actual systems for reliable determinations to be achieved.
- If simulations are used, a description of the model and tools used, and the bases for the timing values, should be provided.
- All data sets should be used. They have been provided in a range of sizes, so as to test fixed-system scaling effects resulting from limited-resource optimizations. Should particular data sets be unusable for some reason (e.g., the dataset requires more memory than that which is available), the reason should be reported.
- There may be no recompilation or manipulation of the software or hardware between runs producing final measurements. Recall that quantifying the effects of system design decisions is one of the goals of this effort. Therefore, the environment must be consistent throughout the tests to ensure validity of measurements relative to one another.

- Tests should be repeated enough times to ensure reproducibility.
- As the DIS effort is primarily concerned with memory issues, measurement of time to perform I/O operations shall ideally be factored out. However, because the relative need for–and speed of–I/O is determined by the architecture, these times should be measured and included in the report. If possible, the time for these operations should be noted, so they can be excluded when appropriate.

3.5 Submission of Results

3.5.1 Required Elements

Participants are expected to supply the following items as a result of their tests:

Item	Description
Architecture description	A detailed description of the hardware and software environments utilized during testing should be supplied. The description should be sufficient that strengths and weaknesses of the architecture pertinent to the benchmarks can be understood. Known performance measures such as bisection bandwidth and feature size should be included. Limits of the architecture (e.g., maximum of 32 processors, or maximum clock rate of 100Mhz) should be identified, and if predicted performance is to be considered, it must be justified in the *Comments* section of the report. As it is unwise to compare raw timings, even for similar architectures, without considering the differences in technology between the systems, this description is critical to the process, and should be organized, detailed, and complete.
Source code	If modifications are made to the baseline source code in support of optimized performance, the revised source code used during testing should be supplied, along with corresponding documentation of the changes, and detailed documentation of the code compilation, assembly, and execution.
Implementation documentation	A detailed record of the implementation, including rationale and approach to optimizations, is expected. This is particularly important when deviations from the baseline code are employed, or when problems in implementation are encountered. An accurate account of the labor required to implement each benchmark is required.
Output data	Output data sets should be made available. Any deviations from the output data specification should be explained.
Measurements	Performance figures for each applicable benchmark should be supplied, along with a description of how they were obtained. Any missing measurements should be explained. Metrics in addition to those required by this specification are encouraged, but they must be accompanied by documentation of how they were gathered, and how they are pertinent to the analysis. See Section 3.3 for more information.
Comments	Participants are encouraged to include any other information pertinent to the benchmarking process, including explanations of special circum-

	stances, or recommendations for improving the benchmark. To be considered, theoretical performance of an unbuilt architecture should be given and justified. Particular attention should be given to the scalability of the architecture with respect to each of the benchmarks in the suite. Results from implementations of other benchmarks are welcomed, also, though these should be sufficiently delineated so as not to obscure the data directly relevant to this benchmark.

3.5.2 How to Submit Results

[To be supplied, July 1999]

3.6 Common Data Types

The reference to several common data types used throughout this document and the accompanying specifications are described in this section. The descriptions given here apply to all data type references unless specifically noted.

Type	Description
byte	A *byte* consists of eight contiguously stored bits, with bit 0 being the least significant bit (LSB) and bit 7 being the most significant bit (MSB). A *signed* variant uses bit 7 as a sign bit.
char	A *char* represents a 7-bit ASCII character with a decimal range of 0 to 127 as defined by the ANSI specification, stored in a byte (with bit 7 always set to zero). The term *whitespace* refers collectively to the characters of space (value 32), horizontal (value 9) and vertical (value 11) tabs, line-feed (value 10), and form-feed (value 12).
short integer	A short integer is a whole number stored contiguously as one 16-bit word (in 2 bytes). Bits 0:15 contain the integer, with bit 0 being the LSB and bit 15 being the MSB. Bit 16 is the sign bit.
integer	An *integer* is a whole number stored contiguously as one 32-bit word (in 4 bytes). Bits 0:30 contain the integer, with bit 0 being the LSB and bit 30 being the MSB. Bit 31 is the sign bit.
float	A *float* is a single-precision 4-byte real number stored contiguously as one 32-bit word in *excess 127* notation, and has a one-bit sign, an 8-bit biased exponent, and a 23-bit fraction. Bits 0:22 contain the 23-bit fraction with bit 0 being the LSB of the fraction and bit 22 being the MSB; bits 23:30 contain the 8-bit biased exponent with bit 23 being the LSB of the biased exponent and bit 30 being the MSB; bit 31 is the sign bit. The value of a float is given by:

<table>
<tr><td></td><td>
<table>
<tr><th>Value</th><th>Bit Pattern</th></tr>
<tr><td>$(-1)^{sign} \times 2^{exponent\ -127} \times 1.fraction$</td><td>$0 < exponent < 255$</td></tr>
<tr><td>$(-1)^{sign} \times 2^{-126} \times 0.fraction$</td><td>$exponent = 0; fraction \neq 0$</td></tr>
<tr><td>$(-1)^{sign} \times 0.0$ (signed zero)</td><td>$exponent = 0; fraction = 0$</td></tr>
<tr><td>+INF (positive infinity)</td><td>$sign = 0; exponent = 255; fraction = 0$</td></tr>
<tr><td>-INF (negative infinity)</td><td>$sign = 1; exponent = 255; fraction = 0$</td></tr>
<tr><td>NaN (Not - a - Number)</td><td>$exponent = 255; fraction \neq 0$</td></tr>
</table>
This description conforms to the IEEE 754 Floating-Point Arithmetic standard and the reader is referred to it for further information including minimum and maximum values.
</td></tr>
<tr><td>double</td><td>
A double is a double-precision 8-byte real number stored contiguously as two successively addressed 32-bit words in excess 1023 notation, and has a one-bit sign, an 11-bit biased exponent, and a 52-bit fraction. Bits 0:51 contain the 52-bit fraction with bit 0 being the LSB of the fraction and bit 51 being the MSB; bits 52:62 contain the 11-bit biased exponent with bit 52 being the LSB of the biased exponent and bit 62 being the MSB; and the highest-order bit (63) contains the sign.

The value of the double is given by:
<table>
<tr><th>Value</th><th>Bit Pattern</th></tr>
<tr><td>$(-1)^{sign} \times 2^{exponent\ -1023} \times 1.fraction$</td><td>$0 < exponent < 2047$</td></tr>
<tr><td>$(-1)^{sign} \times 2^{-1022} \times 0.fraction$</td><td>$exponent = 0; fraction \neq 0$</td></tr>
<tr><td>$(-1)^{sign} \times 0.0$ (signed zero)</td><td>$exponent = 0; fraction = 0$</td></tr>
<tr><td>+INF (positive infinity)</td><td>$sign = 0; exponent = 2047; fraction = 0$</td></tr>
<tr><td>-INF (negative infinity)</td><td>$sign = 1; exponent = 2047; fraction = 0$</td></tr>
<tr><td>NaN (Not - a - Number)</td><td>$exponent = 2047; fraction \neq 0$</td></tr>
</table>
This description conforms to the IEEE 754 Floating-Point Arithmetic standard and the reader is referred to it for further information including minimum and maximum values.
</td></tr>
</table>

3.7 Arithmetic Precision

The mathematics in the benchmark algorithms requires the manipulation of values which cannot be stored in finite-precision memory representations. Since the successful completion of a benchmark is determined by a comparison between the output of the benchmark implementation and results provided by the baseline solution, the question of precision and accuracy must be addressed.

The data types used for the input to the benchmarks conform to the IEEE 754 specification, which also specifies the manipulation of these data types to ensure the mathematically expected results and expected properties for finite arithmetic. The output provided with the benchmarks conform to the IEEE 754 standard, as does the baseline implemen-

tation. Benchmark participants are not required to implement this specification, but the output of their implementations must have the same level of precision, and perform to at least the same level of accuracy for numeric calculations. This requirement allows the baseline results to be accurately compared over the various architectures and implementations.

4 Specifications

This section provides the specifications of each of the five benchmarks, preceded by an explanation of the approach and intent relative to the specifications. The specifications utilize a common outline, and are intended to contain the information needed by implementers.

4.1 Approach

Each specification provided here is intended to be separable from the remainder of the document; it contains all the information needed by a developer charged with building an implementation of a benchmark, with the exceptions of the Common Data Types specification from Section 3.6 and the Arithmetic Precision specification from Section 3.7. Complete specifications of input, algorithm, output, acceptance tests, and required metrics are included.

In several cases, common algorithms used in the solution of the selected problems are optimized for use with traditional systems. For example, certain steps in the Method of Moments algorithm are only present to take advantage of unit-stride memory accesses. While these steps are not strictly part of the solution algorithm, it would be onerous to require participants to independently rediscover them. In some cases, it would require implementers to become experts in the application field. So, the algorithmic descriptions are intended to cover the mathematics of the solutions only. Known optimizations are additionally provided for informational purposes, but implementation of these is not required.

Pseudo-code provided in the algorithmic specifications is intended to provide guidance and clarification of algorithms **only**; it is not intended to represent optimal–*or efficient*–implementations of problem solutions. Similarly, pseudo-code is not intended to represent ‘known optimizations’ as described above, except when specifically identified as such.

4.2 Benchmark Specifications

Detailed descriptions of the benchmark algorithms are included in this section. Any suggestion of the specific implementation of an algorithm is not intentional; all descriptions implying a specific implementation should be viewed as examples only.

4.2.1 Method of Moments

The first class of algorithms chosen for inclusion in the DIS benchmark suite are Method of Moments (MoM) algorithms, which are frequency-domain techniques for computing the electromagnetic scattering from complex objects. MoM algorithms require the solution of large dense linear systems of equations. Traditionally, MoM algorithms have employed direct linear equation solvers for these systems. The high computational complexity of the direct solver approach has limited MoM algorithms to low-frequency problems. Recently, fast solvers have been introduced which have low computational complexity. The potential of these fast solvers to enable MoM algorithms to solve larger problems at higher frequencies is ultimately limited by the speed of main memory. Thus, fast MoM algorithms may benefit from the Data-Intensive Systems research effort.

The scattering of a plane wave of a specified frequency, ω, from an object is given by Maxwell's equations. The Electric Field Integral Equation (EFIE) and the Magnetic Field Integral Equation (MFIE) formulations, which describe the surface current densities induced by an incoming plane wave of frequency ω, are given by

$$\hat{n} \times E(\vec{r}) = \frac{1}{i\omega\varepsilon} \hat{n} \times \int_{\vec{r}' \varepsilon S} \left(-\omega^2 \mu \varepsilon J(\vec{r}') \Psi + (\nabla' \cdot J(\vec{r}')) \nabla' \Psi \right) d\vec{r}' \tag{4.2.1.1}$$

$$2\hat{n} \times H(\vec{r}) = J(\vec{r}) - 2\hat{n} \times \int_{\vec{r}' \varepsilon S} (J(\vec{r}') \times \nabla' \Psi) d\vec{r}' \tag{4.2.1.2}$$

where

$$\Psi(\vec{r}) = \frac{e^{ikR}}{4\pi R} \tag{4.2.1.3}$$

is the Green's function for the incoming plane wave.

The Method of Moments (MoM) approach to solving the EFIE or the MFIE is to discretize the equation by expanding $J(\vec{r})$ in terms of a set of basis functions $B_n(\vec{r})$ as follows

$$J(\vec{r}) = \sum_{n=1}^{N} j_n B_n(\vec{r}) \tag{4.2.1.4}$$

where $J = \{j_n\}$ the vector of expansion coefficients. When this expansion is substituted into the integral equation (4.2.1-1) and the result multiplied by a basis function and integrated over the scattering surface, the problem reduces to solving a linear system of equations

$$Z \cdot J = V \tag{4.2.1.5}$$

for the vector of expansion coefficients $J = \{j_n\}$, where the entries in the matrix $Z = [Z_{mn}]$ and the vector $V = \{v_m\}$ are complicated double integrals over the scattering surface and must be calculated numerically. For example, the entries in $Z = [Z_{mn}]$ are given by

$$Z_{mn} = \int_{\vec{r} \in S} B_m(\vec{r}) \cdot \left\{ \frac{1}{i\omega\varepsilon} \hat{n} \times \int_{\vec{r}' \in S} \left(-\omega^2 \mu\varepsilon B_n(\vec{r}')\Psi + (\nabla' \cdot B_n(\vec{r}'))\nabla'\Psi \right) d\vec{r}' \right\} d\vec{r} \qquad (4.2.1.6)$$

Generally, *N* increases as the square of the frequency, and for typical problems, *N* is greater than 10,000. In traditional MoM algorithms, which first appeared in the late 1960's, the dense linear system $Z \bullet J = V$ is solved by a direct linear equation solution algorithm, which may be composed as an in-core or out-of-core solver. On modern parallel computers, the direct solvers may be extended to work on shared or distributed memory computer architectures.

The advantage of MoM algorithms is that they are exact representations of Maxwell's equations and highly accurate simulations are possible. The disadvantage of the traditional MoM algorithms is that the methods are computationally intensive, especially as the frequency goes up. The computational complexity of traditional MoM algorithms includes $O(N^2)$ integral evaluations to compute the matrix *Z* and $O(N^3)$ arithmetic operations to solve the system $Z \bullet J = V$ for *J*. The memory requirement for traditional MoM algorithms is $O(N^2)$. For these reasons, the traditional MoM algorithms are generally used only for low frequency problems. Although traditional MoM algorithms have been highly optimized on a variety of high-performance computing machines, the largest problems solved so far are for *N* on the order of 100,000.

Recently, new fast MoM algorithms based on fast, iterative linear equation solvers have been introduced. The iterative solvers rely on numerically stable and rapidly converging iteration procedures, such as the preconditioned GMRES method [Saad]. Fast matrix-vector multiply algorithms are used to compute products of the form used in the iterative procedure. The computational complexity of the fast MoM algorithms is $O(N\log N)$. The memory requirement for the fast MoM algorithms is $O(N)$. This is a remarkable reduction from the $O(N^3)$ computational complexity of the traditional MoM algorithms, and potentially allows the solution of much larger problems at higher frequencies.

Rohklin [Rohklin-1, Rohklin-2] has introduced new fast MoM algorithms for the Helmholtz equation, which use iterative linear equation solvers and the fast multipole method (FMM) for fast matrix-vector multiplies. To compute products of the form $Z \bullet X$, the *Z* matrix is not formed or stored, rather the product $Z \bullet X$ is viewed as a field and approximately evaluated by the FMM. The mathematical formulation of the FMM is based on the theory of multipole expansions, and involves translation (change of center) of multipole expansions and spherical harmonic filtering. The computational complexity of these new methods is $O(N\log N)$ and the memory requirement is $O(N)$.

Building on the FMM approach, Dembart, Epton and Yip [Dembart-1 to 4] at Boeing have implemented a fast MoM algorithm in a production grade electromagnetics code used by the company for radar cross-section (RCS) studies. Problems for which the number of unknowns is on the order of 10,000,000 have been solved with this code. Boeing's fast solver uses the preconditioned GMRES iterative method, which requires only the calculation of products of the form $Z \bullet X$, combined with a multilevel FMM for fast matrix-vector multiplies. The solver may be summarized as follows:

1. The preconditioned GMRES iterative solution method is used to solve the system $Z \bullet J = V$. This method does not require computation and storage of the matrix Z, but rather requires the capability to compute, for a given vector X, the product $Z \bullet X$.
2. To compute the product $Z \bullet X$, the Z matrix is first decomposed into two pieces which represent contributions from "close together" and "well separated" basis functions, respectively

$$Z = Z_{near} + Z_{far} \tag{4.2.1.7}$$

3. The matrix Z_{near} is sparse, and it is computed once and for all, directly from the integral representation for its entries, in $O(N)$ integral evaluations. The product $Z_{near} \bullet X$ is computed directly for each X in $O(N)$ arithmetic operations.
4. The matrix Z_{far} is dense, but it is never computed at all—rather the product $Z_{far} \bullet X$ is computed for each X by a multilevel FMM in $O(N\log N)$ arithmetic operations.

Boeing's multilevel FMM method is formulated by enclosing the scatterer in a cube and then successively refining the cube into subcubes until the dimensions of the finest cubes are on the order of several wavelengths. At each level in the cube hierarchy two multipole expansions are computed: the outer expansion (field outside the cube due to sources inside the cube) and the inner expansion (field inside the cube due to sources outside the cube). The outer and inner expansions are efficiently represented by far-field signature functions. The key computations are translating multipole expansions (change of center) and harmonic analysis/synthesis of signature functions. The multilevel FMM calculation begins by computing the outer expansion at the finest level from X. Next, the outer-to-outer translation operation is applied to traverse up the cube hierarchy computing the outer expansions at all levels. Next, the outer-to-inner and inner-to-inner translations are used to traverse down the cube hierarchy computing the inner expansion at all levels. Finally, the matrix-vector product $Z_{far} \bullet X$ is then computed from the inner expansion at the finest level.

The $O(N\log N)$ computational complexity of the FMM results from the computational strategy of applying the outer-to-inner translation at each of the cube hierarchy as follows. At the coarsest level in the cube hierarchy, the outer-to-inner translation is applied to all pairs of cubes that are non-adjacent. For all finer levels in the cube hierarchy, the outer-to-inner translation is applied only to pairs of cubes that are non-adjacent at the given level, and whose parents are adjacent at the next higher level.

Fast MoM algorithms, such as Boeing's described above, have the potential to compute the electromagnetic scattering from complex objects at frequencies 10 to 100 times higher than possible with traditional MoM algorithms. The ultimate potential of these fast MoM algorithms is limited by two memory-related bottlenecks: low reuse of data and non-unit stride. For these reasons, we have chosen to base the *Method of Moments Benchmark* on Boeing's fast solver. The benchmark computes the far-field component of the wave field generated by a collection of radiating scalar sources. The evaluation of the far-field corresponds to the evaluation of the matrix-vector product $Z_{far} \bullet X$ described above. The computational method implemented in the benchmark is a scalar multilevel FMM similar to the multilevel FMM used in Boeing's fast solver. The key FMM kernels represented

in the benchmark are the translation operations and spherical harmonic filtering. Detailed specifications of the *Method of Moments Benchmark* are given in the following sections.

4.2.1.1 Input

An input set for the *Method of Moments Benchmark*, which contains everything required to run the benchmark, consists of three binary input files: source strengths, cubes, and translation operators. The contents of the files are described in the sections below.

4.2.1.1.1 Source Strength File

The source points and their corresponding source strengths are specified in the *Source Strength* input file. The record types in the file are specified in the following table. The first record in the file is a record of type 1 containing a single integer number defining the number of source points. This is followed by a record of type 2 for each source point containing four floating point numbers (double) defining the three spatial coordinates of the source point and the strength of the source at the source point.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	double float Q

4.2.1.1.2 Cube File

The cube hierarchy on which the multilevel FMM operates is specified in the *Cube* input file. The record types in the file are specified in the following table. The first record in the file is a record of type 1 containing a single integer number defining the number of levels in the cube hierarchy. For each level in the cube hierarchy, there is a set of records defining the cubes in the level. The first record for a level is a record of type 1 containing a single integer number defining the number of cubes for the level. This is followed by a set of records for each cube. The first record for a cube is a record of type 2 containing three floating point numbers (double) defining the three spatial coordinates of the cube's center and an integer number defining the number of cubes in the level adjacent to the cube. This is followed by a record of type 1 for each adjacent cube containing a single integer number defining the number of the adjacent cube.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	integer M

4.2.1.1.3 Translation Operators

The tables defining the translation operators are specified in the *Translation Operator* file. The record types in the file are specified in the following table. The file contains four tables: outer-to-outer operator, inner-to-inner operator, outer-to-inner at top level, and outer-to-inner at levels below top level. The four tables all have the same structure: a list of complex numbers followed by a list of integer numbers.

The first record in the table is of type 1, containing a single integer number defining the number of complex numbers in the table. This is followed by a record of type 2, containing a single complex number (double), for each entry in the list. The next record in the table is a record of type 1, containing a single integer number defining the number of integer numbers in the table. This is followed by a record of type 1, containing a single integer, for each entry in the list.

Record Type	Contents
1	integer N
2	double complex X

4.2.1.2 Algorithmic Specification

The *Method of Moments Benchmark* computes the far-field component of the wave field generated by a collection of radiating scalar sources. The computational method implemented in the benchmark is a scalar multilevel FMM similar to the multilevel FMM used by Boeing in its fast MOM solver discussed above. The mathematical formulation of the scalar multilevel FMM relies on the theory of scalar multipole expansions and the theory of harmonic analysis/synthesis of signature functions. We present the specifications of the benchmark in the following order.

1. Field Generated by a Distribution of Scalar Sources
2. Multilevel Fast Multipole Method
3. Translation Operations
4. Spherical Harmonic Synthesis/Analysis

4.2.1.2.1 Field Generated by a Distribution of Scalar Sources

For the *Method of Moments Benchmark*, we define a scalar wave field as a field $\phi(\vec{x})$ which satisfies the scalar Helmholtz equation

$$\left(\nabla^2 + k^2\right)\phi = 0 \qquad (4.2.1.8)$$

where k is the wave number. The benchmark computes the field generated by a collection of sources radiating from source points $\{\vec{x}_i\}_{i=1}^{N}$ with amplitudes $\{q_i\}_{i=1}^{N}$ at a collection of field points $\{y_i\}_{j=1}^{M}$. The value of the field at the field points is given by

$$\phi(\vec{y}_j) = \sum_{i=1}^{N} q_\alpha h_0(k|\vec{y}_j - \vec{x}_i|) \qquad (4.2.1.9)$$

where h_o is the spherical Bessel function of the first kind (we assume a time variation of $e^{-i\omega t}$).

Figure 4.2.1-1. Coarse Grid and Scatterer

For convenience in the *Method of Moments Benchmark*, we take the field points to be identical to the source points. Under this assumption, the computational complexity of computing the field values at the field points by approximately evaluating the Bessel functions and doing the sum is $O(N^2)$. This is similar to the computational complexity of the matrix-vector multiply step in an iterative MoM solver for N unknowns.

4.2.1.2.2 Multilevel Fast Multipole Method

In this section we formulate the multilevel fast multipole method and show that the computational complexity of the method is $O(N\log N)$.

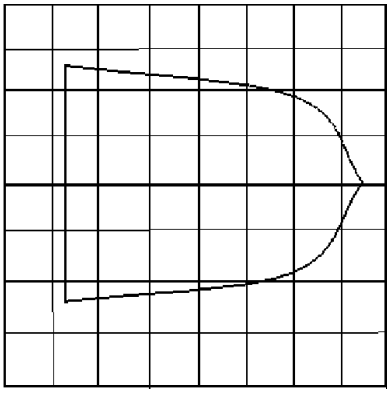


Figure 4.2.1-2. Fine Grid and Scatterer

We begin by introducing the multilevel cubical grid system utilized in the multilevel FMM. The source and field points are enclosed in a cube. This cube is then subdivided into eight equal subcubes, and each of those is similarly sub-divided, until a sufficiently fine grid is achieved. The dimensions of the cubes at the finest level are on the order of several wavelengths and the number of levels is $O(N\log N)$. At each level of this grid system there is a set of "active" cubes, that is, cubes which contain the source or field points. Only the active cubes are used in the multilevel FMM. One level of such a grid is depicted in Figure 4.2.1-1. A second, finer grid is shown in Figure 4.2.1-2. To simplify the figures the illustrations are two-dimensional. Cubes are represented as squares and the

scatterer as a one-dimensional curve. The source and field points are distributed along the scatterer and are not explicitly shown in the figures. To simplify the description of the multilevel FMM, we will describe the method for the two level (fine and coarse) cubical grid system, shown in Figure 4.2.1-1 and Figure 4.2.1-2. As appropriate, we will indicate which steps must be repeated for cube hierarchies with more than two levels.

First, we introduce the outer-to-inner translation. Referring to the fine grid shown in Figure 4.2.1-2, we consider two distinct cubes Cd and Ce with centers $\vec{d}$ and $\vec{e}$, respectively. The finite degree outer expansion centered at $\vec{d}$ for the wave field $\phi_d^{(d)}(\vec{x})$ outside Cd due to the sources inside Cd has the form

$$\phi_d^{(d)}(\vec{x}) = \sum_{n=0}^{N_d} \sum_{m=-n}^{n} D_n^m h_n\left(k\left|\vec{x}-\vec{d}\right|\right) Z_n^m\left(\left(\vec{x}-\vec{d}\right)\Big/\left|\vec{x}-\vec{d}\right|\right) \tag{4.2.1.10}$$

where the coefficients D_n^m are given by

$$D_n^m = \sum_{i=1}^{N} q_i 4\pi j_n\left(k\left|\vec{x}_i-\vec{c}\right|\right) Z_n^{-m}\left(\left(\vec{x}_i-\vec{c}\right)\Big/\left|\vec{x}_i-\vec{c}\right|\right) \tag{4.2.1.11}$$

The outer-to-inner translation from $\vec{d}$ to $\vec{e}$ is the construction of an inner expansion for $\phi_d^{(d)}(\vec{x})$ centered at $\vec{e}$ having the form

$$\phi_e^{(e)}(\vec{x}) = \sum_{n=0}^{N_e} \sum_{m=-n}^{n} E_n^m j_n\left(k\left|\vec{x}-\vec{e}\right|\right) Z_n^m\left(\left(\vec{x}-\vec{e}\right)\Big/\left|\vec{x}-\vec{e}\right|\right) \tag{4.2.1.12}$$

that is valid for all $\vec{x}$ inside Ce. The calculation of the coefficients E_n^m from the coefficients D_n^m is described in the discussion of the translation operations below.

For a given decomposition into cubes and specified values for Nd and Ne, the accuracy of the outer-to-inner translation from $\vec{d}$ to $\vec{e}$ depends only on the distance $\vec{d}-\vec{e}$ between the cubes. For a specified accuracy, we say two cubes satisfy the far-field condition if they are sufficiently separated so that the accuracy of the outer-to-inner translation for the pair satisfies the specified accuracy.

The outer-to-inner translation provides a (1-level) computational tool to compute the field at the field points. The calculation proceeds in several steps

- For all cubes Ce we apply, for all cubes Cd that satisfy the far-field condition, the outer-to-inner translation to translate Cd's outer expansion to an inner expansion at Ce's center and sum the translated inner expansions. The result is the inner expansion for all cubes Ce due to all the sources in cubes that satisfy the far-field condition with respect to Ce.
- For all cubes Ce, we compute the field at field points in Ce as the sum of the far-field, which is given by the inner expansion computed in step1, and the near-field due to all sources inside cubes (including Ce) that don't satisfy the far-field condition with respect to Ce.

If we decompose the domain into very fine cubes, the computational complexity of the second step is reduced to only $O(1)$, but the computational complexity of the first step remains $O(N_2)$. Similarly, if we decompose the domain into course cubes, the computational complexity of the first step is reduced to only $O(1)$, but the computational complexity of the second step remains $O(N_2)$. This problem is resolved by the multilevel FMM. To specify the multilevel FMM we introduce the outer-to-outer and inner-to-inner translations.

Referring to Figure 4.2.1-1 and Figure 4.2.1-2, we consider a cube Cd at the coarse level with center $\vec{d}$ and a cube Cc at the fine level with center $\vec{c}$. The finite degree outer expansion centered at $\vec{c}$ for the wave field $\phi_c^{(c)}(\vec{x})$ outside Cc due to the sources inside Cc has the form

$$\phi_{\vec{c}}^{(c)}(\vec{x}) = \sum_{n=0}^{N_c} \sum_{m=-n}^{n} C_n^m h_n\left(k\left|\vec{x}-\vec{c}\right|\right) Z_n^m\left(\left(\vec{x}-\vec{c}\right)/\left|\vec{x}-\vec{c}\right|\right) \tag{4.2.1.13}$$

where the coefficients C_n^m are given by

$$C_n^m = \sum_{i=1}^{N} q_i 4\pi j_n\left(k\left|\vec{x}_i-\vec{c}\right|\right) Z_n^{-m}\left(\left(\vec{x}_i-\vec{c}\right)/\left|\vec{x}_i-\vec{c}\right|\right) \tag{4.2.1.14}$$

The outer-to-outer translation from $\vec{c}$ to $\vec{d}$ is the construction of an outer expansion for $\phi_c^{(c)}(\vec{x})$ centered at $\vec{d}$ having the form

$$\phi_{\vec{d}}^{(d)}(\vec{x}) = \sum_{n=0}^{N_d} \sum_{m=-n}^{n} D_n^m h_n\left(k\left|\vec{x}-\vec{d}\right|\right) Z_n^m\left(\left(\vec{x}-\vec{d}\right)/\left|\vec{x}-\vec{d}\right|\right) \tag{4.2.1.15}$$

that is valid for all $\vec{x}$ outside Cd. The calculation of the coefficients D_n^m from the coefficients C_n^m is described in the discussion of the translation operations below.

Referring again to Figure 4.2.1-1 and Figure 4.2.1-2, we consider a cube Ce at the coarse level with center $\vec{e}$ and a cube Cf at the fine level with center $\vec{f}$. Suppose the we have constructed a finite degree inner expansion centered at $\vec{e}$ for the wave field $\phi_{\vec{e}}^{(e)}(\vec{x})$ inside Ce due to the sources outside Ce of the form

$$\phi_{\vec{e}}^{(e)}(\vec{x}) = \sum_{n=0}^{N_e} \sum_{m=-n}^{n} E_n^m j_n\left(k\left|\vec{x}-\vec{e}\right|\right) Z_n^m\left(\left(\vec{x}-\vec{e}\right)/\left|\vec{x}-\vec{e}\right|\right) \tag{4.2.1.16}$$

The inner-to-inner translation from $\vec{e}$ to $\vec{f}$ is the construction of an inner expansion for $\phi_{\vec{e}}^{(e)}(\vec{x})$ centered at $\vec{f}$ having the form

$$\phi_{\vec{f}}^{(f)}(\vec{x}) = \sum_{n=0}^{N_f} \sum_{m=-n}^{n} F_n^m j_n\left(k\left|\vec{x}-\vec{f}\right|\right) Z_n^m\left(\left(\vec{x}-\vec{f}\right)/\left|\vec{x}-\vec{f}\right|\right) \tag{4.2.1.17}$$

that is valid for all $\vec{x}$ in Cf. The calculation of the coefficients F_n^m from the coefficients E_n^m is described in the discussion of the translation operations below.

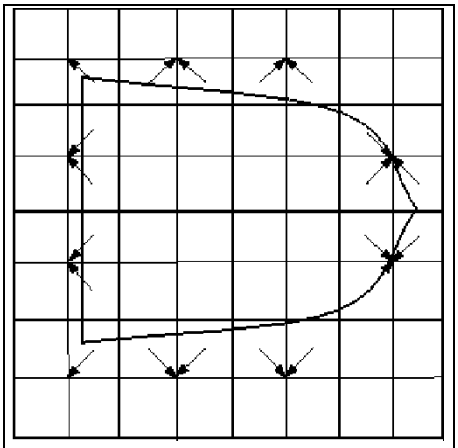


Figure 4.2.1-3. Outer-to-Outer Translation from Fine to Coarse Level

The outer-to-outer and inner-to-inner translations provide the computational tools we need to use the cube hierarchy to efficiently compute the field at the field points. The calculation proceeds in several steps, traversing up and down the cube hierarchy.

1. For all cubes Cc at the fine level we compute the finite degree outer expansion for Cc from the sources at the source points.
2. For all cubes Cd at the coarse level, we apply the outer-to-outer translation to translate the outer expansion for each of Cd's children to Cd's center and sum the translated outer expansions. The result is that we have constructed the outer expansion for all cubes Cd at the coarse level. The outer-to-outer translation from the coarse level to the fine level is shown in Figure 4.2.1-3.
3. For all cubes Ce at the coarse level we apply, for all cubes Cd at the coarse level that satisfy the far field condition, the outer-to-inner translation to translate Cd's outer expansion to an inner expansion at Ce's center and sum the translated inner expansions. The result is the inner expansion for all cubes Ce at the coarse level due to all the sources in cubes at the coarse level that satisfy the far-field condition with respect to Cf. The outer-to-inner translation at the coarse level is shown in Figure 4.2.1-4.

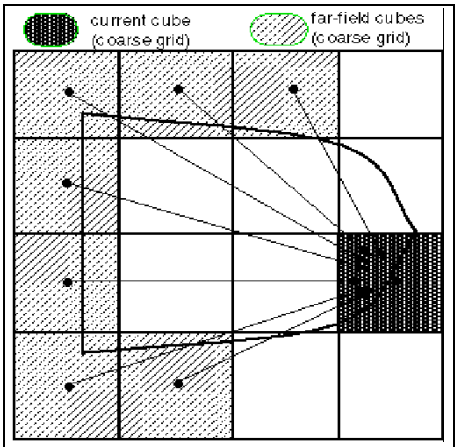


Figure 4.2.1-4. Outer-to-Inner Translation at the Coarse Level

4. For all cubes Ce at the coarse level, we apply, for each child Cf of Ce the inner-to-inner translation to translate the inner expansion for Ce to Cf's center. The result is the inner expansion for all cubes Cf at the fine level due to sources in cubes Cc at the fine level such that the parents of Cf and Cc satisfy the far-field condition. The inner-to-inner translation from the fine level to the coarse level is shown in Figure 4.2.1-5.
5. For all cubes Cf at the fine level we apply, for all cubes Cc at the fine level that satisfy the far field condition *and for which the parents of Cf and Cc don't satisfy the far-field condition*, the outer-to-inner translation to translate Cc's outer expansion to an inner expansion at Cf's center and sum the translated inner expansions. The result is the inner expansion for all cubes Cf at the fine level due to all the sources in cubes at the fine level that satisfy

the far-field condition with respect to Cf. The outer-to-inner translation at the fine level is shown in Figure 4.2.1-7.

6. For all cubes Cf at the fine level, we compute the field at field points in Cf as the sum of the far-field, which is given by the inner expansion computed in step5, and the near-field due to all sources inside cubes (including Cf) that don't satisfy the far-field condition with respect to Cf. The evaluation of the near-field is shown in Figure 4.2.1-6.

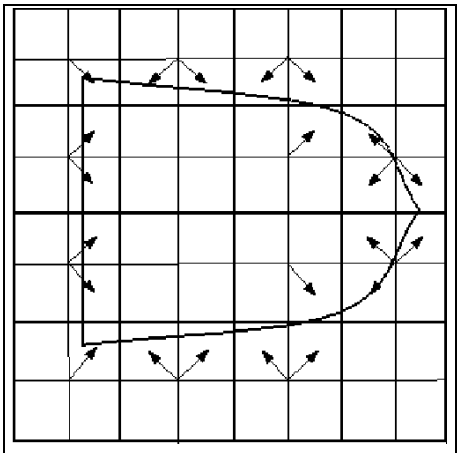


Figure 4.2.1-5. Inner-to-Inner Translation from Fine Level to Coarse Level

The two-level computation described above is easily extended to a multilevel cube hierarchy. The efficiency of the multilevel calculation derives from step 5, where the outer-to-inner translation only need to be applied at a level to cube pairs at the level for which the outer-to-inner translation was not already accounted for at the higher levels. As a result the computational cost of the translation operations is roughly the same at all levels of the cube hierarchy. Also, by using approximately log *N* levels the computational cost of the step 6 is roughly the same for all field points. Thus, the computational complexity of the multilevel FMM is $O(N\log N)$.

4.2.1.2.3 Translation Operations

In this section we use Rohklin's translation theorems to specify the translation operations: outer-to-inner, outer-to-outer and inner-to-inner. The theorems are expressed in terms of far field signature functions on the unit sphere. For a scalar wave field as a field $\phi(\vec{x})$, we define the associated far field signature function $\hat{\phi}(\vec{s})$ as follows

$$\hat{\phi}_{\vec{c}}(\vec{s}) = \lim_{\tau\to\infty}\left[ik\tau e^{-ik\tau}\phi(\vec{c}+\tau\vec{s})\right] \qquad (4.2.1.18)$$

for points $\vec{s}$ on the unit sphere. When $\phi(\vec{x})$ is defined by a multipole expansion, we may

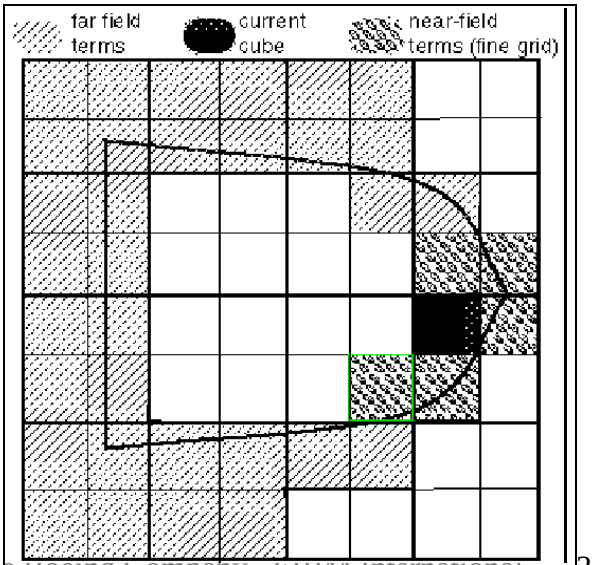


Figure 4.2.1-6. Near-field Contributions at Fine Level

Figure 4.2.1-7. Outer-to-Inner Translation at Fine Level

use the spherical harmonic transform to evaluate the associated far field signature function. Accordingly, for the finite degree outer expansions $\phi_c^{(c)}(\vec{x})$ and $\phi_d^{(d)}(\vec{x})$ and the finite degree inner expansions $\phi_e^{(e)}(\vec{x})$ and $\phi_f^{(f)}(\vec{x})$ discussed above, the associated far field signature functions are given by

$$\hat{\phi}_{\hat{c}}^{(c)}(\vec{s}) = \sum_{n=0}^{N_c} \sum_{m=-n}^{n} C_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.19}$$

$$\hat{\phi}_{\hat{d}}^{(d)}(\vec{s}) = \sum_{n=0}^{Nd} \sum_{m=-n}^{n} D_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.20}$$

$$\hat{\phi}_{\hat{e}}^{(e)}(\vec{s}) = \sum_{n=0}^{Ne} \sum_{m=-n}^{n} E_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.21}$$

$$\hat{\phi}_{\hat{f}}^{(f)}(\vec{s}) = \sum_{n=0}^{N_f} \sum_{m=-n}^{n} F_n^m Z_n^n(\vec{s}) / i^{n+1} \tag{4.2.1.22}$$

Rohklin's translation theorems also make use of three operators defined on the unit sphere: the harmonic projection operator Π_N, the outer-to-outer/inner-to-inner translation operator $G(\vec{s};\vec{r},N)$ and the outer-to-inner translation operator $M(\vec{s};\vec{r},N)$.

The spherical harmonic projection operator Π_N is defined by

$$\Pi_N[f(\vec{s})] = \frac{1}{4\pi} \iint_{S_1} \delta_N(\vec{s}\cdot\vec{t}) f(\vec{t}) d\omega_t \tag{4.2.1.23}$$

where

$$\delta_N(\vec{s}\cdot\vec{t}) = \sum_{0}^{N} (2n+1) P_n(\vec{s}\cdot\vec{t}) = \sum_{n=0}^{N} \sum_{m=-n}^{n} 4\pi Z_n^m(\vec{s}) \left[Z_n^m(\vec{t}) \right]^* \tag{4.2.1.24}$$

The outer-to-outer/inner-to-inner translation operator $G(\vec{s};\vec{r},N)$ is defined by

$$G(\vec{s};\vec{r},N) = \sum_{n=0}^{N} (2n+1) i^n j_n(k|\vec{r}|) P_n(\vec{s}\cdot\vec{t}) \tag{4.2.1.25}$$

$$= \sum_{n=0}^{N} \sum_{m=-n}^{n} 4\pi i^n j_n(k|\vec{r}|) Z_n^m(\vec{s}) (-1)^m Z_n^{-m}(\vec{r}) \tag{4.2.1.26}$$

The outer-to-inner translation operator $M(\vec{s};\vec{r},N)$ is defined by

$$M(\vec{s};\vec{r},N) = \sum_{n=0}^{N} (2n+1) i^n h_n(k|\vec{r}|) P_n(\vec{s}\cdot\vec{t}) \tag{4.2.1.27}$$

$$= \sum_{n=0}^{N} \sum_{m=-n}^{n} 4\pi i^n h_n(k|\vec{r}|) Z_n^m(\vec{s}) (-1)^m Z_n^{-m}(\vec{r}) \tag{4.2.1.28}$$

With these definitions in place, Rohklin's translation theorems may be states as follows:

- Outer-to-Outer Translation Theorem. The finite degree outer signature functions $\hat{\phi}_{\vec{c}}^{(c)}(\vec{s})$ and $\hat{\phi}_{\vec{d}}^{(d)}(\vec{s})$ are related to one another by the identity

$$\hat{\phi}_{\vec{d}}^{(d)}(\vec{s}) = \Pi_{N_d}\left[G(\vec{s};\vec{d}-\vec{c},N_c+N_d)\hat{\phi}_{\vec{c}}^{(c)}(\vec{s})\right] \tag{4.2.1.29}$$

- Outer-to-Inner Translation Theorem. The finite degree outer signature function $\hat{\phi}_{\vec{d}}^{(d)}(\vec{s})$ and the finite degree inner signature function $\hat{\phi}_{\vec{e}}^{(e)}(\vec{s})$ are related to one another by the identity

$$\hat{\phi}_{\vec{e}}^{(e)}(\vec{s}) = \Pi_{N_e}\left[G(\vec{s};\vec{e}-\vec{d},N_d+N_e)\hat{\phi}_{\vec{d}}^{(d)}(\vec{s})\right] \tag{4.2.1.30}$$

- Inner-to-Inner Translation Theorem. The finite degree inner signature functions $\hat{\phi}_{\vec{e}}^{(e)}(\vec{s})$ and $\hat{\phi}_{\vec{f}}^{(f)}(\vec{s})$ are related to one another by the identity

$$\hat{\phi}_{\vec{f}}^{(f)}(\vec{s}) = \Pi_{N_f}\left[G(\vec{s};\vec{f}-\vec{e},N_e+N_f)\hat{\phi}_{\vec{e}}^{(e)}(\vec{s})\right] \tag{4.2.1.31}$$

The importance of Rohklin's translation theorems is that they reduce translation of multipole expansions to pointwise multiplication on the unit sphere of a signature function by a translation operator, followed by filtering.

4.2.1.2.4 Spherical Harmonic Synthesis/Analysis

In this section we specify Epton and Dembart's method for filtering and interpolating signature functions. Filtering of signature functions is needed for implementation of spherical harmonic projection operator Π_N used in Rohklin's translation theorems stated above. Interpolation and filtering of signature functions is needed to move discrete tabulations of the functions on the unit sphere between levels without losing accuracy. First, we define spherical harmonic analysis and synthesis for finite degree signature functions tabulated on a grid on the unit sphere. Then we specify Epton and Dembart's algorithm for performing spherical harmonic analysis and synthesis. Finally, we describe Epton and Dembart's method for filtering and interpolating signature functions.

For a finite degree signature function $f(\vec{s})$ of the form

$$f(\vec{s}) = \sum_{n=0}^{N}\sum_{m=-n}^{n} f_n^m Z_n^m(\vec{s}) \tag{4.2.1.32}$$

a tabulation $\{f(\theta_k,\phi_l): 0 ? k \le N_\theta - 1, 0 \le l \le N_\phi - 1\}$ of $f(\vec{s})$ on a grid of $\vec{s}$ values on the unit sphere defined as follows (N_θ and N_ϕ will be specified in terms of N.)

$$\Theta = \{\theta_k : \theta_k = (k+1/2)\pi/N_\theta, k = 0; N_\theta - 1\} \tag{4.2.1.33}$$

$$\Phi = \{\phi_l : \phi_l = 2\pi l/N_\phi, l = 0; N_l - 1\} \tag{4.2.1.34}$$

The process of obtaining the coefficients $\{f_n^n : 0 \le n \le N, |m| \le n\}$ from the tabulation $\{f(\theta_k,\phi_l): 0 ? k \le N_\theta - 1, 0 \le l \le N_\phi - 1\}$ is called spherical harmonic analysis. The in-

verse process of obtaining the tabulation $\{f(\theta_k,\phi_l): 0 ? k \le N_\theta - 1, 0 \le l \le N_\phi - 1\}$ from the coefficients $\{f_n^n : 0 \le n \le N, |m| \le n\}$ is called spherical harmonic synthesis.

Epton and Dembart's algorithm for performing spherical harmonic analysis and synthesis is based on the observation that the spherical harmonics $Z_n^m(\vec{s})$ can be viewed as trigonometric polynomials in θ and ϕ as follows

$$f(\vec{s}) = \sum_{n=0}^{N}\sum_{m=-n}^{n} f_n^m Z_n^m(\vec{s}) = \sum_{n=0}^{N}\sum_{m=-n} f_n^m z_n^m(\theta) e^{im\phi} = f(\theta,\phi) \tag{4.2.1.35}$$

where $\vec{s} \ldots (\theta,\phi)$ and $Z_n^m(\vec{s}) = z_n^m(\theta)e^{im\phi}$. Changing the order of summation gives

$$f(\theta,\phi) = \sum_{m=-N}^{N}\sum_{n=|m|}^{N} f_n^m z_n^m(\theta) e^{im\phi} = \sum_{m=-N}^{N} f^{(m)}(\theta) e^{im\phi} \tag{4.2.1.36}$$

where the functions $\{f^{(m)}(\theta): -N \le m \le N\}$ are given by

$$f^{(m)}(\theta) = \sum_{n=|m|}^{N} f_n^m z_n^m(\theta) \tag{4.2.1.37}$$

A linear system of equations relating the f_n^m and the $f(\theta_k,\phi_l)$ is formulated by comparing discrete and analytic expansions of $f^{(m)}(\theta)$ as follows.

1. Application of the discrete Fourier inversion theorem to $f(\theta,\phi)$ gives $f^{(m)}(\theta)$ as follows

$$f^{(m)}(\theta) = \frac{1}{N_\phi}\sum_{l=0}^{N_\phi-1} e^{im\phi_l} f(\theta,\phi_l) \tag{4.2.1.38}$$

2. Application of the shifted cosine transform to $f^{(m)}(\theta)$ for m even gives

$$f^{(m)}(\theta) = \sum_{p=0}^{N_\theta-1} \cos[p\theta] F_p^{(m)} \cdots m = even \tag{4.2.1.39}$$

where the numbers $\{F_p^{(m)} : m = even, -N \le m \le N, 0 \le p \le N_\theta - 1\}$ are given by

$$F_p^{(m)} = \sum_{k=0}^{N_\theta-1} \frac{1}{\cos[p\theta]} \left\{ \frac{1}{N_\phi}\sum_{l=0}^{N_\phi-1} e^{im\phi_l} f(\theta_k,\phi_l) \right\} \cdots m = even \tag{4.2.1.40}$$

3. Similarly, application of the shifted sine transform to $f^{(m)}(\theta)$ for m odd gives

$$f^{(m)}(\theta) = \sum_{p=0}^{N_\theta-1} \sin[(p+1)\theta] F_p^{(m)} \cdots m = odd \tag{4.2.1.41}$$

where the numbers $\{F_p^{(m)} : m = odd, -N \le m \le N, 0 \le p \le N_\theta - 1\}$ are given by

$$F_p^{(m)} = \sum_{k=0}^{N_\theta - 1} \frac{1}{\sin[(p+1)\theta]} \left\{ \frac{1}{N_\phi} \sum_{l=0}^{N_\phi - 1} e^{im\phi_l} f(\theta_k, \phi_l) \right\} \cdots m = odd \tag{4.2.1.42}$$

4. Expanding $z_n^m(\theta)$ in a cosine series for m even gives

$$f^{(m)}(\theta) = \sum_{n=|m|}^{N} f_n^m z_n^m(\theta) = \sum_{n=|m|}^{N} f_n^m \sum_{p=0}^{n} A_{n,p}^m \cos[p\theta] \cdots m = even \tag{4.2.1.43}$$

$$= \sum_{p=0}^{N} \left\{ \sum_{\substack{n=|m| \\ n \geq p}}^{N} A_{n,p}^m f_n^m \right\} \cos[p\theta] \cdots m = even \tag{4.2.1.44}$$

5. Similarly, expanding $z_n^m(\theta)$ in a sine series for m odd, gives

$$f^{(m)}(\theta) = \sum_{n=|m|}^{N} f_n^m z_n^m(\theta) = \sum_{n=|m|}^{N} f_n^m \sum_{p=0}^{n-1} A_{n,p+1}^m \sin[(p+1)\theta] \cdots m = odd \tag{4.2.1.45}$$

$$= \sum_{p=0}^{N-1} \left\{ \sum_{\substack{n=|m| \\ n \geq p+1}}^{N} A_{n,p+1}^m f_n^m \right\} \sin[(p+1)\theta] \cdots m = odd \tag{4.2.1.46}$$

6. By comparing the discrete and analytic cosine expansions of $f^{(m)}(\theta)$ for m even, we obtain the following system of linear equations

$$\sum_{\substack{n=|m| \\ n \geq p}}^{N} A_{n,p}^m f_n^m = F_p^{(m)} \cdots m = even \tag{4.2.1.47}$$

7. Similarly, by comparing the discrete and analytic sine expansions of $f^{(m)}(\theta)$ for m odd, we obtain the following system of linear equations

$$\sum_{\substack{n=|m| \\ n \geq p+1}}^{N} A_{n,p+1}^m f_n^m = F_p^{(m)} \cdots m = odd \tag{4.2.1.48}$$

Epton and Dembart's method for filtering and interpolating signature functions consists of the following computational steps to transform an input tabulation $f(\theta_k, \phi_l)$ of a finite degree signature function $f(\vec{s})$ on the unit sphere to an output tabulation $g(\theta_k, \phi_l)$ of the filtered/interpolated finite degree signature function $g(\vec{s})$ on the unit sphere.

Step	Action
1	Starting with a tabulation $f(\theta_k, \phi_l)$ of a finite degree signature function $f(\vec{s})$ on the unit sphere, perform the discrete Fourier transform in the ϕ direction using an FFT.
2	Split the transformed data into even and odd frequency data arrays.

3	Transpose the even and odd frequency data arrays.
4	Apply the discrete shifted cosine transform to the even frequency data array in the θ direction using a FFT. Similarly, apply the discrete shifted sine transform to the odd frequency data array in the θ direction using a FFT.
5	Solve the linear system of equations defined by equations 4.2.1-46 and 4.2.1-47 to compute f_n^m.
6	Compute g_n^m from f_n^m. For filtering, g_n^m is obtained from f_n^m by dropping terms. For interpolation, g_n^m is obtained from f_n^m by adding zero terms.
7	Apply the inverse discrete shifted cosine transform to the even frequency data array in the θ direction using a FFT. Similarly, apply the inverse discrete shifted sine transform to the odd frequency data array in the θ direction using a FFT.
8	Transpose the even and odd frequency data arrays.
9	Combine the transformed even and odd frequency data into a single data array.
10	Perform the inverse discrete Fourier transform in the ϕ direction using an FFT to compute the tabulation $g(\theta_k, \phi_l)$ of the finite degree signature function $g(\vec{s})$ on the unit sphere.

4.2.1.3 Output

An output set for the *Method of Moments Benchmark* consists of two binary files: computed far-field and metrics report. The contents of the files are described in the sections below.

4.2.1.3.1 Far-Field

The far-field computed at the field points, which are identical with the source points specified in the *Source Strength* file, are output to the *Far-Field* file. The record types in the file are specified in the table below. The first record in the file is a record of type 1 containing a single integer number defining the number of field points. This is followed by a record of type 2 for each field point containing three floating point numbers (double) and a single complex number (double) defining the three spatial coordinates of the field point and strength of the computed far-field at the field point.

Record Type	Contents			
1	integer N			
2	double float	double float	double float	double complex

	X	Y	Z	E

4.2.1.3.2 Metrics Report

The *Method of Moments Benchmark* collects data for the evaluation of the metrics specified in Section 4.2.1.5. The metric data is output in the *Metrics Report* output file. The record types in the file are specified in the table below. The first record of the file is a record of type 1 containing three floating-point numbers (float) defining the first and secondary performance metrics. The second record of the file is a record of type2 containing six floating-point numbers (float) defining the tertiary performance measures for the translation operations. The third record of the file is a record of type2 containing six floating-point numbers (float) defining the tertiary performance measures for the spherical harmonic filtering/synthesis calculations.

Record Type	Contents					
1	float T1	float T2	float T3			
2	float T1	float T2	float T3	float T4	float T5	float T6

Implementers may add additional records to the *Metric Report* file to record data for evaluating their implementation relative to the metrics for scalability with respect to problem size and scalability with respect to processors, if these metrics apply to their implementation.

4.2.1.4 Acceptance Test

The *Acceptance Test* for the *Method of Moments Benchmark* consists of a comparison between the reference far-field and the computed far-field. The reference far-field is specified in the *Far-Field Reference* file. The number of digits of accuracy specified for the acceptance test is also defined in the *Far-Field Reference* file. The contents of the *Far-Field Reference* file are described below.

The computed far-field at a field point passes the acceptance test if the computed far-field and the reference far-field agree to the specified number of digits. To avoid possible self-checking errors, implementers should perform the acceptance test on a computer separate from the PIM computer that is being benchmarked. Implementers should report the results of the acceptance test in the *Acceptance Test Report* file. The contents of the *Acceptance Test Report* file are described below.

4.2.1.4.1 Far-Field Reference

The field points and their corresponding far-field strengths used in the acceptance test are specified in the *Far-Field Reference* input file. The record types in the file are specified in the table below. The first record in the file is a record of type 1 containing a single integer number defining the number of digits of accuracy specified for the acceptance test.

The second record in the file is of type 2 containing a single integer number defining the number of field points. This is followed by a record of type 3 for each field point containing three floating point numbers (double) and a single complex number (double) defining the three spatial coordinates of the field point and strength of the far-field at the field point.

Record Type	Contents			
1	integer N			
2	double float X	double float Y	double float Z	double complex E

4.2.1.4.2 Acceptance Test Report

The results of the acceptance test are output in the *Acceptance Test Report* file. The record types in the file are specified in the table below. The first record of the file is a record of type 1 containing a single integer number defining how many field points failed the acceptance test. If any of the field points failed the acceptance test, the first record is followed by a record for each field point that failed the acceptance test. The records are of type 2 containing three floating point numbers (double) and two complex numbers (double) defining the three spatial coordinates of the field point and the strengths of the reference field and computed far-field at the field point.

Record Type	Contents				
1	integer N				
2	double float X	double float Y	double float Z	dble complex E1	dble complex E2

4.2.1.5 Metrics

The *Metrics* for the *Method of Moments Benchmark* consist of three measures: performance, scalability with respect to problem size and scalability with respect to processors. The metrics are described the sections below. The performance metric is a quantitative metric measured directly by the *Method of Moments Benchmark* code and reported in the *Metric Report* file. The scalability metrics are subjective measures and should be evaluated by the implementers with respect to their PIM architecture.

4.2.1.5.1 Performance

The performance of an implementation of the *Method of Moments Benchmark* is measured in wall-clock time. The performance measures are summarized in the table below. The primary measure of performance is the total wall-clock time for the calculation of the

far-field by the multilevel FMM. The secondary measure of performance is the breakdown of the total time into the total time for all translation operations and the total time for all spherical harmonic filtering/synthesis calculations. The tertiary measures of performance are the breakdown of the secondary measures into the total time for each of the six steps in the multilevel FMM as specified in Section 0.

Metric	Wall-Clock Time
primary	Total Time for Multilevel FMM
secondary	Total Time for Translation Operations
tertiary	Total Time for Translation Operations During Initialize Sig. Functions at Finest Level
tertiary	Total Time for Translation Operations During Outer-to-Outer below Coarsest Level
tertiary	Total Time for Translation Operations During Outer-to-Inner at Coarsest Level
tertiary	Total Time for Translation Operations During Inner-to-Inner above Finest Level
tertiary	Total Time for Translation Operations During Outer-to-Inner below Coarsest Level
tertiary	Total Time for Translation Operations During Evaluation of Far-Field at Finest Level
secondary	Total Time for Filtering/Synthesis
tertiary	Total Time for Filtering/Synthesis During Initialize Sig. Functions at Finest Level
tertiary	Total Time for Filtering/Synthesis During Outer-to-Outer below Coarsest Level
tertiary	Total Time for Filtering/Synthesis During Outer-to-Inner at Coarsest Level
tertiary	Total Time for Filtering/Synthesis During Inner-to-Inner above Finest Level
tertiary	Total Time for Filtering/Synthesis During Outer-to-Inner below Coarsest Level
tertiary	Total Time for Filtering/Synthesis During Evaluation of Far-Field at Finest Level

The demonstration benchmark code includes timing subroutines. Calls to the timing routines are embedded in the benchmark code to measure the specified performance metrics. The benchmark code also includes a subroutine to output the performance metrics to the *Metrics Report* file.

4.2.1.5.2 Scalability With Respect to Problem Size

The *Method of Moments Benchmark* includes a "flat plate" test series that sets the field points equal to the source points and geometrically scales the number of points. Over this set of test cases, a log/log plot of the total wall-clock time for the calculation of the far-field by the multilevel FMM (the primary performance measure) versus number of points should give approximately a straight line with slope +1. Deviations from this expected behavior give an indication of the scalability with respect to problem size of an implementation of the *Method of Moments Benchmark*. The flat plate series is described in more detain in Section 4.2.1.8.

4.2.1.5.3 Scalability With respect to Processors

The flat plate test series can also be used to study the scability of an implementation of the *Method of Moments Benchmark* with respect to processors. For a given size problem, a log/log plot of the total wall-clock time for the calculation of the far-field by the multilevel FMM (the primary performance measure) versus number of processors should give approximately a straight line with slope –1, if the implementation scales linearly with the number of processors. Deviation from the straight line indicates the extent of linear performance. Plotting the total time versus number of processors over the set of test cases in the flat plate test series gives an indication of how the scalability with respect to processor varies over problem size.

4.2.1.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.1.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.1.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.2 Simulated SAR Ray Tracing

The Simulated SAR image process consists of three major steps:

- Geometry Sampling
- Electromagnetic Scattering Prediction
- Image Formation.

Of these, the Geometry Sampling and the Image Formation steps take the majority of the CPU time, and present the most interesting sub-applications to build benchmarks around.

The Geometry Sampling is accomplished by using ray tracing to simulate the physical optics part of the electromagnetic scattering problem. Here, rays are sent out from an idealized synthetic aperture, and intersections between objects and these rays are found. At each intersection, the information about the object intersected is determined and recorded. A specular reflection ray is generated at the intersection point, and it is then fired into the object database. This creates a recursive process that allows the radar energy to be followed as it bounces through the target database. The rays are followed until a user-defined number of reflections have occurred, or until the ray leaves the database area. This process results in a linked list of intersection information that is called a *ray history*. The ray history is the output of the Geometry Sampling process.

For this benchmark, the Geometry Sampling section is further broken down into two sub-parts that, when put together, form the complete Geometry Sampling benchmark.

The first sub-part is a ray server. This consists of the intersection-finding part of the ray-tracing problem. It tests a bounding box structure, described in detail later, to find a list of objects that possibly intersect the ray under test. It then applies the intersection code for each type of object in this list. Once all intersections are found, it returns the intersection that is closest to the starting point of the ray under test.

The second sub-part is the Ray-Tracing Controller. This generates the grid of rays that simulate the synthetic aperture and generates the specular reflection rays based on the intersection information. It calls the ray server and passes a new ray from either the synthetic aperture grid or a reflection ray. It also tests the ray before it is sent, to see if the maximum number of reflections have been processed. This controller then takes the intersection information returned from the ray server and creates the ray history by creating the linked list for each ray fired from the synthetic aperture grid.

The Image Formation part of the simulated SAR benchmark suite takes an array of electromagnetic responses and maps them into a rectangular grid of the slant plane. This remapped EM array is then passed through a convolution that adds the effects of the system IPR. The final step is to convert the complex image into something that can be displayed. To achieve this, magnitude detection is performed on the complex image.

4.2.2.1 Input

The input for the Simulated SAR benchmark is divided into two parts: the inputs for the Ray-Tracing portion of the benchmark, and a separate set of inputs for the Image Formation portion. Each of these inputs specifications contain subsections for *Input Variables*

and *Input Data*. The *Input Variables* contain the information that would be contained in a command line argument. The *Input Data* contains the databases used by each of the benchmarks.

4.2.2.1.1 Recursive Ray Tracer Input

4.2.2.1.1.1 Input Variables

Aperture specification	The aperture specification gives the location of the radar in global coordinates, a look direction vector, and a field of view (FOV) that simulates the synthetic aperture. If the FOV is given as ZERO, then a Parallel Projection is used instead of a Perspective Projection. In the Parallel case the target is centered in the Aperture and the look direction vector will give the azimuth and elevation angles need to place the aperture in the correct position.
Number of sample points	The number of sample points tells the ray-tracer the sample resolution in both range (Vertical) and cross range (Horizontal). This is directly related to the radar resolution. Three sampling resolutions will be used in this benchmark. • 512 x 512 for a small, low resolution, sample. • 2048 x 2048 for a medium sample. • 4096 x 4096 for a large sample.
Maximum number of reflections	This will specify the maximum number of reflections that are to be traced. This will be set to 3 for all benchmark runs.
Database Specifications	This will include the database name that will be used to select between the target databases specified in the following section. This specification will also contain a target rotation (yaw, pitch, and roll) and translation that positions the target in global coordinates.

4.2.2.1.1.2 Input Data

The target databases will consist of models built from both polygons and solid geometry using Constructive Solid Geometry. Each target database will also contain a hierarchical bounding-box structure. For the Polygon target model, we will assume triangular facets or three-sided polygons. The file format of each model type is described below.

4.2.2.1.1.2.1 Polygon target model file format

The overall file format is as follows.

File header
Part1
Sub-part 1
Sub-part 2
.
.

Sub-part n
Part2
Sub-part 1
Sub-part 2
.
.
Sub-part n
.
.
Part-n
Sub-part 1
Sub-part 2
.
.
Sub-part n

File header		
Target model name	char[256]	ASCII test description
Model bounding box	float [3][2]	Minimum X target value Maximum X target value Minimum Y target value Maximum Y target value Minimum Z target value Maximum Z target value
Number of parts	int	Total number of modeled parts
Part format		
Part name	char[256]	ASCII part description
Part bounding box	float[3][2]	Minimum X part value Maximum X part value Minimum Y part value Maximum Y part value Minimum Z part value Maximum Z part value
Number of sub-parts	int	Total number of sub-parts
Sub-part format		
Sub-Part name	char[256]	ASCII subpart description
Sub-Part bounding box	float[3][2]	Minimum X part value Maximum X part value Minimum Y part value Maximum Y part value Minimum Z part value Maximum Z part value
Number of nodes	int	Number of facet vertices
Vertex list	float[N][3]	Number of nodes by x,y,z Coordinates

Number of Facets	int	Number of 3-sided facets Built from the above nodes
Facet list	int [M][5]	Number of facets by vertex 1 index into vertex list, vertex 2 index into vertex list, vertex 3 index into vertex list, Material index, and Surface index.

Here is an example file for a simple unit box with the lower left corner at (0,0,0). The material type is 1 and the surface type is 3 for all facets.

```
Box
0.0 1.0 0.0 1.0
6

Front Face
0.0 1.0 0.0 0.0 0.0 1.0
1
Front Face
0.0 1.0 0.0 0.0 0.0 1.0
4
0.0 0.0 0.0
0.0 0.0 1.0
1.0 0.0 0.0
1.0 0.0 1.0
2
1 2 4       1     3
1 3 4       1     3

Back Face
0.0 1.0 1.0 1.0 0.0 1.0
1
Back Face
0.0 1.0 1.0 1.0 0.0 1.0
4
0.0 1.0 0.0
0.0 1.0 1.0
1.0 1.0 0.0
1.0 1.0 1.0
2
1 2 4       1     3
1 3 4       1     3

Left Face
0.0 0.0 0.0 1.0 0.0 1.0
1
Left Face
0.0 0.0 0.0 1.0 0.0 1.0
4
0.0 0.0 0.0
0.0 1.0 0.0
0.0 0.0 1.0
0.0 1.0 1.0
2
1 2 4       1     3
1 3 4       1     3
```

```
Right Face
1.0 1.0 0.0 1.0 0.0 1.0
1
Right Face
1.0 1.0 0.0 1.0 0.0 1.0
4
1.0 0.0 0.0
1.0 1.0 0.0
1.0 0.0 1.0
1.0 1.0 1.0
2
1 2 4       1     3
1 3 4       1     3

Top Face
1.0 0.0 0.0 1.0 1.0 1.0
1
Top Face
0.0 0.0 0.0 1.0 1.0 1.0
4
0.0 0.0 1.0
0.0 1.0 1.0
1.0 0.0 1.0
1.0 1.0 1.0
2
1 2 4       1     3
1 3 4       1     3

Bottom Face
1.0 1.0 0.0 1.0 0.0 0.0
1
Bottom Face
1.0 1.0 0.0 1.0 0.0 0.0
4
0.0 0.0 0.0
1.0 0.0 0.0
0.0 1.0 0.0
1.0 1.0 0.0
2
1 2 4       1     3
1 3 4       1     3
```

4.2.2.1.1.2.2 CSG solid target model file format

For the CSG solid target models, we will use the BRL CAD .cg geometry description file format. This is an older format, but it is easier to work with. The geometry description file contains all the information required to define the physical components of a geometry model and the operations that are required to construct it. The format of each line of data is a detailed record format. The entire file is divided into four basic parts: title control, primitive definitions, region definitions, and region identifications.

The title control section simply provides a name for the model and the units of measure (millimeters, centimeters, meters, inches, or feet abbreviated as mm, cm, m, in, and ft).

The primitive definition section uniquely describes each of the primitive solids to be used in the model construction (See Figure 4.2.2-1).

The region definition section identifies each constructed region and the Boolean operation that is to be used to create it. Boolean operations will be performed in the order in which they appear in the region definition section. As seen in the example of Figure 4.2.2-3, a space or blank character may be used as an optional operator. If the operator is blank and the following solid/region number is positive then the operator is assumed to be a "union". Otherwise, if the solid/region number is negative then the operator is assumed the "difference"

The region identification section simply contains a list of all region IDs and their associated attributes. Figure 4.2.2-3 contains a sample of a complete geometry description file.

There is one special line of data that is required in the description file that is not part of the geometry description or construction. Between the region definition section and region identification section, the ray tracer expects to find the value of -1 in columns 1-5; this is used as a delimiter to mark the end of the region definition section.

TITLE record	
Columns	*Contents*
1-5	Model units (in, ft, mm, cm, m)
6-65	Name for targets

CONTROL record	
Columns	*Contents*
1-5	Number of primitives
6-10	Number of regions

PRIMITIVE DEFINITION records
See Table A

REGION DEFINITION records	
Columns	*Contents*
1-5	Region number
7-8	Boolean operator
9-13	Primitive number
14-15	Boolean operator
16-20	Primitive number
21-22	Boolean operator
23-27	Primitive number

28-29	Boolean operator
30-34	Primitive number
35-36	Boolean operator
37-41	Primitive number
42-43	Boolean operator
44-48	Primitive number
49-50	Boolean operator
51-55	Primitive number
56-57	Boolean operator
58-62	Primitive number
63-64	Boolean operator
65-69	Primitive number
71-80	Comments

Notes on Boolean operations:
"DIFFERENCE" - A negative primitive number
"INTERSECTION" - A positive primitive number
"UNION" - 'or' in the Boolean operator columns
The union operation is performed between the two sets of primitives that are listed before and after an *'or'* operator.

eg... 2 2 -4or 5 6 or 7 -1
(region) (solids...)

The operations for this example will be performed in the following order:
1) DIFFERENCE between primitives 2 and 4
2) INTERSECTION between primitives 5 and 6
3) UNION the results of steps 1 and 2
4) DIFFERENCE between primitives 7 and 1
5) UNION the results of steps 3 and 4

In this example, implied parentheses exist around (2 -4), (5 6) and (7 -1).
Other examples of operations include:
2 2 -4 6 or 7 -1

In this example, the DIFFERENCE is taken between primitives 2 and 4, and then the INTERSECTION is taken between that result and primitive 6. A DIFFERENCE is taken between primitives 7 and 1. The UNION of these results is then taken.

REGION IDENTIFICATION record	
Columns	*Contents*
1-3	Region number
4-10	Component code number (1-9999)

11-15	Space code number (1-99)
16-20	Material code
21-30	(unused)
31-80	Region description

Geometry Description Record Formats	
Half Space	HAF
Arbitrary Tetrahedron	ARB4
Polyhedron - 5 Vertices	ARB5
Polyhedron - 6 Vertices	ARB6
Arbitrary Wedge	ARW
Right Angle Wedge	RAW
Polyhedron - 7 Vertices	ARB7
Polyhedron - 8 Vertices	ARB8
Box	BOX
Rectangular Parallelepiped RPP	
Triangular - faceted Polyhedron	ARS
Truncated General Cone	TGC
Truncated Elliptical Cone	TEC
Truncated Right Cone	TRC
Right Elliptical Cylinder	REC
Right Circular Cylinder	RCC
Right Parabolic Cylinder	RPC
Right Hyperbolic Cylinder RHC	
Elliptical Paraboloid	EPA
Elliptical Hyperboloid	EHY
General Ellipsoid	ELLG
Ellipsoid of Revolution	ELL
Sphere	SPH
Elliptical Torus	ETO
Circular Torus	TOR

Cols 1-5	6-8	9-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
No.	RPP		xmin	xmax	ymin	ymax	zmin	zmax	comments
No.	BOX		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			W_x	W_y	W_z	D_x	D_y	D_z	comments
No.	RAW		V_x	V_y	V_z	D_x	D_y	D_z	comments
No.			H_x	H_y	H_z	W_x	W_y	W_z	comments
No.	SPH		V_x	V_y	V_z	R			comments
No.	ELL		V_x	V_y	V_z	A_x	A_y	A_z	comments
No.			R						comments
No.	TOR		V_x	V_y	V_z	N_x	N_y	N_z	comments
No.			R_1	R_2					comments
No.	RCC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			R						comments
No.	REC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	B_x	B_y	B_z	comments
No.	TRC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			R_1	R_2					comments
No.	EHY		V_x	V_y	V_z	H_x	H_y	B_z	comments
No.			A_x	A_y	A_z	R_1	R_2	C	comments
No.	EPA		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	R_1	R_2		comments
No.	RHC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			B_x	B_y	B_z	R	C		comments
No.	ARW		V_x	V_y	V_z	$H1_x$	$H1_y$	$H1_z$	comments
No.			$H2_x$	$H2_y$	$H2_z$	B_x	B_y	B_z	comments
No.	RPC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			B_x	B_y	B_z	R			comments
No.	ARB4		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.	ARB5		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5				comments

Figure 4.2.2-1. Solid Primitive Definitions

Cols 1-5	6-8	9-10	11-20	21-30	31-40	41-50	51-60	61-70	71-80
No.	ARB6		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5	X_6	Y_6	Z_6	comments
No.	ARB7		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5	X_6	Y_6	Z_6	comments
No.			X_7	Y_7	Z_7				comments
No.	ARB8		X_1	Y_1	Z_1	X_2	Y_2	Z_2	comments
No.			X_3	Y_3	Z_3	X_4	Y_4	Z_4	comments
No.			X_5	Y_5	Z_5	X_6	Y_6	Z_6	comments
No.			X_7	Y_7	Z_7	X_8	Y_8	Z_8	comments
No.	TEC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	B_x	B_y	B_z	comments
No.			ratio						comments
No.	TGC		V_x	V_y	V_z	H_x	H_y	H_z	comments
No.			A_x	A_y	A_z	B_x	B_y	B_z	comments
No.			T(A)	T(B)					comments
No.	HAF		N_x	N_y	N_z	D_h			comments
No.	ETO		V_x	V_y	V_z	N_x	N_y	N_z	comments
No.			C_x	C_y	C_z	R	R_d		comments
No.	ELLG		V_x	V_y	V_z	X_x	X_y	X_z	comments
No.			Y_x	Y_y	Y_z	Z_x	Z_y	Z_z	comments
No.	ARS		M	N					comments
No.			$X_{1,1}$	$Y_{1,1}$	$Z_{1,1}$	$X_{1,2}$	$Y_{1,2}$	$Z_{1,2}$	comments
.			.						.
.			.						.
.			.						.
No.			$X_{1,N}$	$Y_{1,N}$	$Z_{1,N}$				
No.			$X_{2,1}$	...					
.			.						
.			.						
No.			$X_{M,1}$	...					
.			.						
.			.						
No.			$X_{M,N}$	$Y_{M,N}$	$Z_{M,N}$				

Figure 4.2.2-2. Solid Primitive Definitions (continued)

For ARS the symbol 'M' represents the number of curves and the symbol 'N' represents the number of points required to define the curve with the most number of points. (eg. If M = 2 where curve 1 requires 3 points to define it and curve 2 requires 4 points then the value of N must be 4 .)

The Equation for a half plane is: NxX + NyY + NzZ = Dh

Where V is the vector, D is the depth vector, H is the height vector, W is the width vector, N is the normal vector, R is the radius, A is the semi-major axis of ellipse, and B is the semi-minor axis of ellipse.

An example file follows.

```
m   Simple test object in meter units
  2  2
  1rpp    -1.5000   1.5000  -1.5000  1.5000  -1.5000   1.5000
  2rpp    -0.5000   0.5000  -0.5000  0.5000  -0.5000   0.5000
  1     1    -2
  2     2
 -1
  1  100   0   0   0   main box
  2  100   2   0   0   inside box
```

Figure 4.2.2-3. Example .cg file of a hollow box.

4.2.2.1.2 Image Formation Inputs

4.2.2.1.2.1 Input Variables

The input variables for the image formation benchmark will be stored as ASCII text. Each item will be on a separate line, with the variable number first followed by a text comment identifying that line of the input variable file. The system IPR will be written out in the text file in row column order. There will be a separate Input Variable file for each input data set.

Input data file name	This will be the name of the Input data file that will be used with this Input Variable file.
Aperture specification	The aperture specification here will include the number of samples in range and cross range and image resolution in meters.
System IPR	This will be an array of floats representing the convolution kernel. The kernel can be as small as 3 x 3, where only the main lobe information will appear in the final image, or as large as 35 x 35, where numerous side lobes appear in the final image.

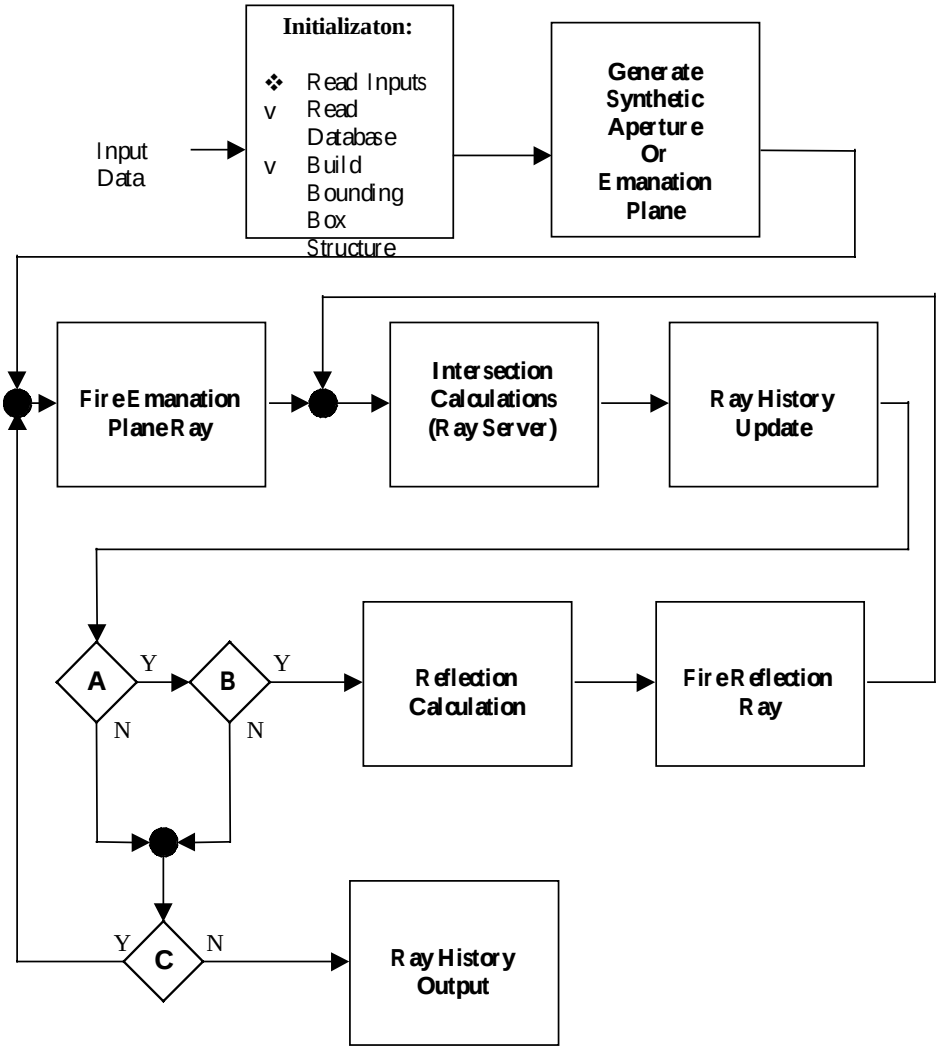
4.2.2.1.2.2 Input Data

The input data arrays are EM responses from the ray history arrays. The EM responses are stored in complex format. The ray history and EM responses will come in three sizes and in both float and double precision formats. These are ray histories with EM responses added right after the subpart name.

- A small array at 512 X 512 sample points,
- A medium array at 2048 X 2048 sample points, and
- A large array at 4096 X 4096 sample points.

4.2.2.2 Algorithmic Specification

4.2.2.2.1 Recursive Ray Tracing Benchmark Algorithm Specifications



4.2.2.2.1.1 Initialization

The initialization process reads in the input variables and input database. It then builds the bounding-box structure and object structure from the input database. The bounding-

box structure contains the bounding hierarchy and each bounding surface contains tags to the object structure. In the initialization the Emanation Ray History pointer array is created and initialized.

Both the bounding-box structure and the object structure should be available to all routines.

Because the BRL-CAD .cg format does not support bounding boxes, a separate file will be provided that contains the bounding information. The format of this bounding-box file is the same as that of the polygon models, except it will contain no polygon data. The object and sub-object name will match the names found in the .cg file thus providing the linkage between the objects and their bounding boxes.

4.2.2.2.1.2 Generation of the Emanation Plane

Using the aperture specification and the number of sample points, from the input data, an Emanation plane is generated.

If the FOV is ZERO, and it always will be for this benchmark, this means that a parallel projection is to be used. With this projection, all the rays are fired in the look direction. The only thing that needs to be determined is the position and scale of the sample points. For this benchmark, we will assume that for the parallel projection case the target will be centered in the sample space and that the top-level bounding box just fits into the sample window. This means that the target translation is ignored and only the rotation is used. Each point in the top level bounding box must be put through the target rotation matrix and then through the viewing matrix. The viewing matrix is built from the rotations needed to rotate the target into the viewing azimuth and elevation. This rotation can be derived using the following pseudo code.

```
Assume that z is up and create a vector up = [0,0,1]
Find the vector that is the cross product of the lookat vector and the
up vector.
Make sure that this new vector is a unit vector and call it alpha
Now find the cross product of the lookat vector and the alpha vector.
Again, make sure it is a unit vector and call it Beta.
The 3x3 Euler rotation matrix is then:
      Alpha_i,    Alpha_j,    Alpha_k
      Beta_i,     Beta_j,     Beta_k
      Lookat_i,   Lookat_j,   Lookat_k
```

Keep this viewing rotation matrix around, as it will be used in the generation of the emanation rays in the next step. It is possible at this point to combine the target rotation matrix and the viewing matrix, so only one matrix multiplication has to be done.

When these rotations are applied to the top-level bounding-box points, a new bounding box is derived that shows the target extent in the viewing frame, if the maximum distance in the horizontal and vertical directions are taken. The Delta size of the viewing window or Emanation plane is known in target units. To get the needed size of an individual pixel, take the larger of the two and divide by the number of sample points in that direction. This benchmark will assume square pixels; thus, a pixel will have the same dimension in both the vertical and horizontal. This same bounding information can be used to center the target in the sample space. The result of this will be a (x, y) coordinate in viewing space based on the row and column in the sample space.

The distance from the radar to the center of the target is found by subtracting the target translation vector from the radar location vector. The magnitude of this difference vector is the desire distance number.

4.2.2.2.1.3 Emanation Plane Ray Firing

This routine is the beginning of the outside loop the Ray tracing system. It takes the information derived in generating the Emanation plane and produces global starting coordinates and direction vectors for the out going rays. Every time it is called a new starting ray position and direction vector are calculated for the next sample point in the Emanation plane.

It generates the starting point in global coordinates by taking the row and column numbers and using the scaling information from the generation routine to derive an x, y, z point in viewing space. The distance from the center of the target to the radar is added to the z value to arrive at the viewing coordinate. This is then converted into a global coordinate by passing this vector through the inverse of the viewing and target rotation matrices. This global coordinate is then the starting position for the current ray.

The direction vector for this benchmark, because we are only using parallel projections, is just the look direction vector from the input variables, aperture specification.

A flag is set that lets the ray history update routine know that a new emanation plane rays has been fired. The starting position and direction are then passed to the Ray Server.

This routine is called on the first ray, when the previous ray leaves the target database, or when the maximum number of reflections, of the previous ray, has been reached.

4.2.2.2.1.4 Intersection Calculations (Ray Server)

The ray server receives a ray, starting position and direction vector, and determines what, if any, objects in the target database this ray intersects. Once the intersections are found, they are processed and the nearest surface intersection is returned. The Ray Server is divided into two major parts.

4.2.2.2.1.5 Bounding-Box Intersections

The first section is the bounding box intersection process. Here the bounding-box hierarchy is tested against the ray. The top level bounding box is tested first. If it is intersected then each object bounding box is tested. If the current ray does not intersect the top-level box, the ray server returns a flag that no intersections were found. This process continues down the bounding box hierarchy until all possible intersected objects have been found.

4.2.2.2.1.6 Object Intersection

The second section finds the actual points of intersection, if they exist, for each object identified in the above process. Once the intersection points have been identified they are sorted by distance from the ray starting point and the nearest intersection is returned. The ray server also returns the surface normal information at the intersection point.

Polygon Models	With Polygon models, this is a straightforward process of testing

	each polygon, in each sub-object, against the ray under test. If an intersection is found it is recorded in the intersection list array. When all the intersections are found, the intersection nearest the starting point of the ray under test is return along with the surface normal at that polygon. In this benchmark, all the polygons are three sided and thus the surface normal is easily calculated using the polygon vertices. It should be noted that the normal is the same for any point on that polygon.
Solid Models with CSG	With solid models and CSG operators, things are a little more difficult. Finding the intersections with the solid models requires more effort, do to their more complex shapes. The intersections come in pairs. One intersection enters the object and the other leaves the object. The real work comes in using the intersection pairs and the CSG operators to find the true intersection point for any object. The supported CSG operators can be seen in Figure 4.2.2-4. If a ray were to pass through center top of the cube in Figure 4.2.2-4, the intersection pairs would be operated on as shown in Figure 4.2.2-5. The CSG operations are executed as written in the .cg file. It is noted here, that with solids, the bounding boxes for objects surround a complete CSG object. The sub-object bounding boxes surround the basic solid primitives that are combined, using the CSG operators, to form the complete CSG object. In the example in Figure 4.2.2-4, the object bounding box would surround the cube and cylinder. The sub-object bounding boxes would be around the cube as sub-object 1 and cylinder as sub-object 2.

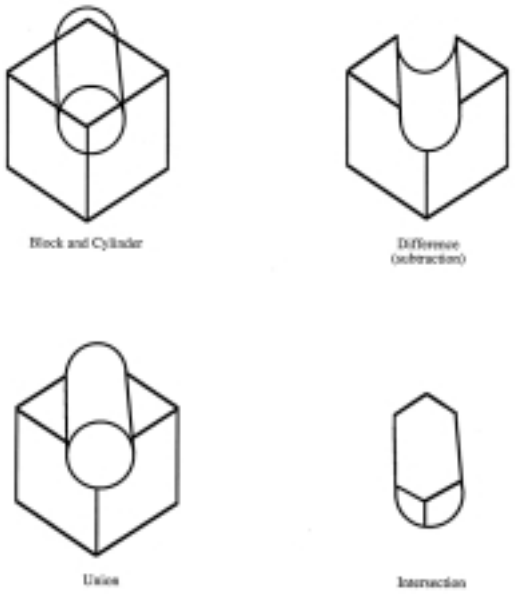


Figure 4.2.2-4. Supported CSG Operators

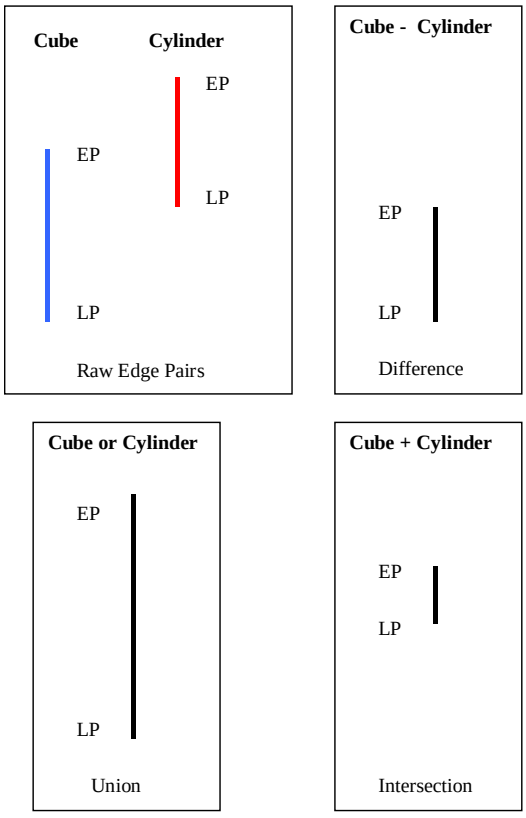


Figure 4.2.2-5. CSG Operators on Intersection Pairs

4.2.2.2.1.7 Surface Intersection Algorithms

All the surface intersection algorithms needed for this benchmark can be found in " An Introduction to Ray Tracing", Edited by Andrew S. Glassner, Academic Press, ISBN 0-12-286160-4. This is referenced in the Bibliography section, but it is given special note here. This is a single resource that has much, if not all, the needed information about Ray Tracing algorithms. It is felt that the coverage of intersection algorithms on pages 33 through 119 of that text is much better and less confusing than what this author could put forth.

4.2.2.2.1.8 Ray History Update

This routine handles the ray history linked list. There are three possible functions that may run.

If the ray that is being processed is a new ray from the emanation plane, the ray history update routine starts a new ray history link and generates a new node structure, initializes the node, records the information returned by the ray server, and adds one to the total intersection count.

If the ray that is being processed is a reflected ray, the ray history update generates a new node structure, initializes it, and places the appropriate links in the new node, the previous node, and records the information returned by the Ray Server. It also updates the previous node with the reflected ray information that is now available. It then adds one to the total intersection count.

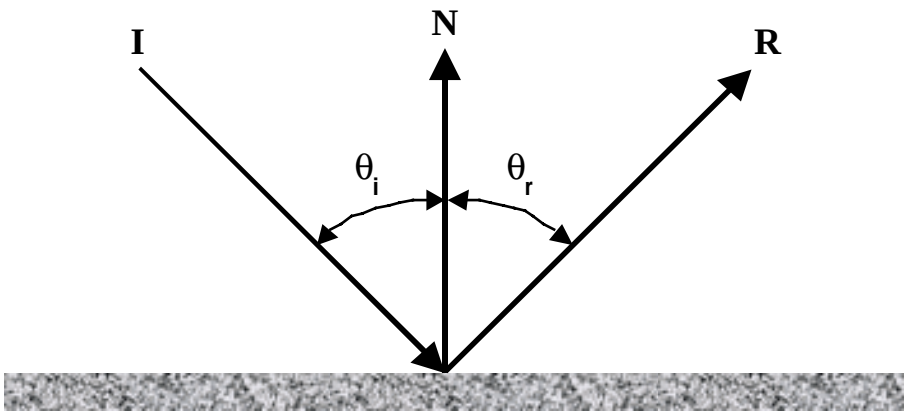
If the ray that is being processed returns from the Ray Server without finding an intersection, the ray history server updates the current node with the reflected ray information and then does nothing else.

The node generation consists of allocating a new section of memory for the new node structure and then initializes it by entering a NULL into the previous_node and next_node pointer variables.

The process of starting a new ray history link involves placing a pointer to the newly generated node structure into the emanation ray history pointer array. This array has the same dimensions as the emanation plane and array keeps a pointer to each ray history node associated with a new ray being fired from the emanation plane.

4.2.2.2.1.9 Reflection Calculations

A reflection ray is calculated for every intersection point as long as the maximum number of intersections has not been exceeded. The process of finding a reflection ray is base on Snells' law for a perfect specular reflection. This law states that an incoming ray of will reflect at an equal and opposite angle, relative to the normal at the reflection point:



$$\theta_i = \theta_r$$

The formula for the direction of a specularly reflected ray is

$$\mathbf{R} = \mathbf{I} - 2(\mathbf{N} \bullet \mathbf{I})\mathbf{N}$$

where **I** is the incident ray, **N** is the normal at the reflection point, and **R** is the reflected ray.

4.2.2.2.1.10 Reflection Ray Firing

The reflection ray firing is similar to the Emanation ray firing. In this case, the reflected ray calculated above becomes the look direction vector and the intersection point becomes the starting position of the ray. These are passed to the Ray Server.

4.2.2.2.1.11 Ray History Output

After all the emanation rays have been fired and their reflection paths followed, the final step in the geometry sampling process is to output the ray history. This is done by using the emanation ray history pointer array to follow each ray history. As each node is traversed the data in the node structure is written out to the ray history file in the form specified in section 4.2.2.3.1. As this data is written out, the pointers for previous_node and next_node get replace with index numbers as if each node was an entry in a large 1-D array of nodes. Those nodes that have NULL pointers are replaced by -1.

4.2.2.2.2 Image Formation Benchmark Algorithm Specifications

The Image Formation process consists of three steps that convert the EM Contributions into a slant plane SAR Image.

4.2.2.2.2.1 Mapping EM contributions

The first step in the image formation process is to map the EM contributions of each ray history into the slant plane. The slant plane image is a grid of cells measured in range and cross range (see Figure 4.2.2-6).

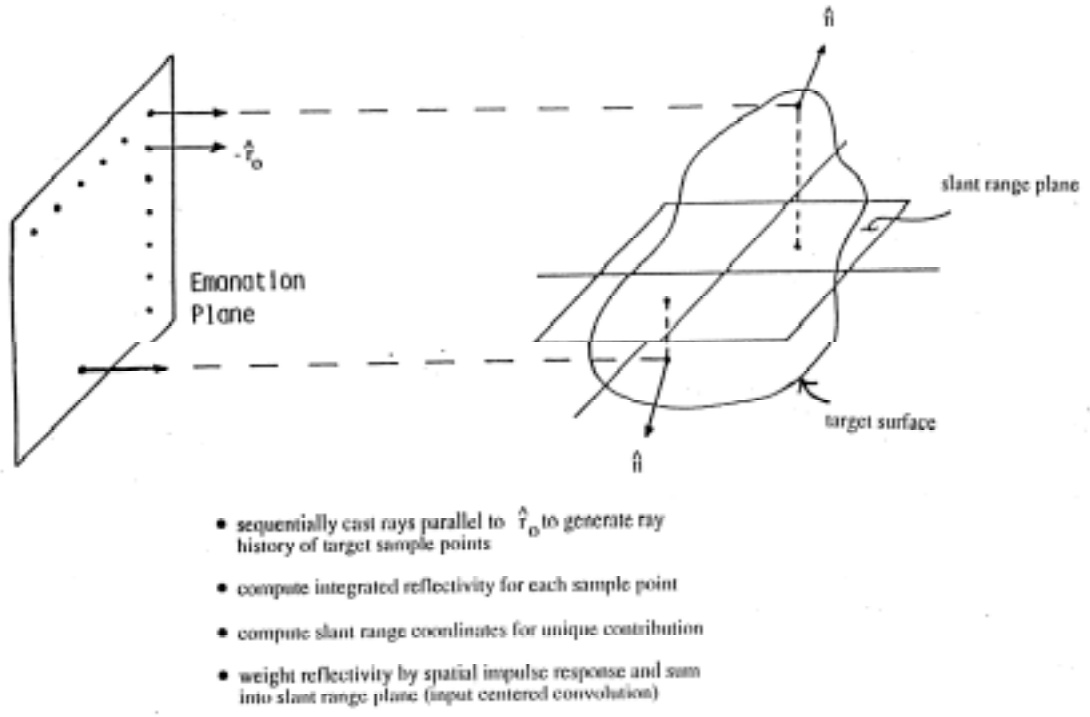
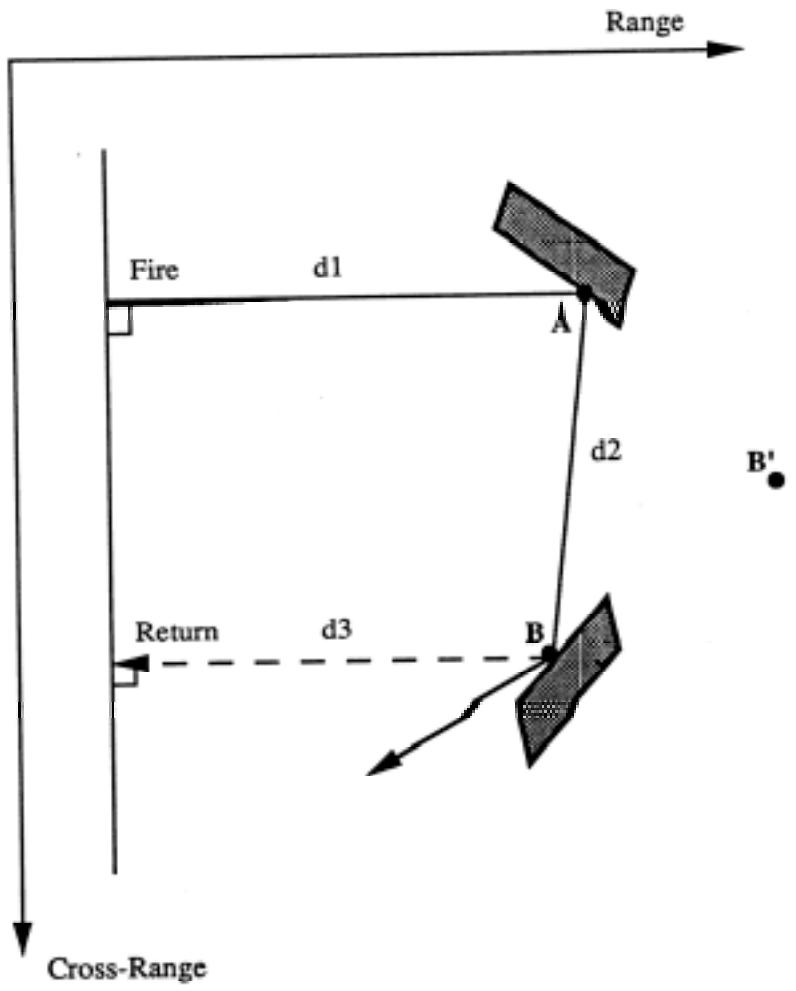


Figure 4.2.2-6. Formation of the SAR Image

The first intersection EM contribution is mapped into this plane based on the position in the emanation plane, for the cross range position, and the distance from the firing point to the intersection point, for the range position. The multiple reflections are mapped to a mean aspect and more distant range consistent with their appearance in the SAR imagery (See Figure 4.2.2-7).



For Reflection A:

Contribution is mapped directly at A in SAR image

For Reflection B:

Contribution is mapped at B' in SAR image

Range equals total distance (d1+d2+d3) divided by 2
Cross-range position is average of cross-range positions of A and B.

Figure 4.2.2-7. SAR Mapping of Returns

This process is followed for each ray history and each EM contribution for each reflection resulting in a complex data array.

4.2.2.2.2.2 IPR Convolution

Once the EM scattering is mapped to the slant plane, it must be convolved with the system IPR. This is an input centered convolution. The convolution edge effects are accounted for by mirroring the edge pixels of the slant plane. Pixels are copied from the left and right edges, range, and then copied from the top and bottom, cross-range. This insures the corners of the mirrored images are filled correctly. The number of pixels copied is equal to half of the IPR convolution width. This is a standard 2-D convolution process.

4.2.2.2.2.3 Detection

The final process in the Image Formation sequence is the Detection process. This is nothing more than finding the Magnitude image of the complex image plane. The output from this process forms a viewable, or real-valued, image and is the desired output for this portion of the benchmark.

4.2.2.3 Output

4.2.2.3.1 Recursive Ray Tracing Benchmark Output

The output for the Recursive Ray Tracing Benchmark is a link list of the intersection information for each sample in the aperture. This ray history file will be a binary file based on an array of the following structure with the first number in the file being the number of array entries

```
int  number_of_entries
struct  Ray_History {
      char  object_name[256];
      char  part_name[256];
      char  subpart_name[256];
      float       intersection_x;
      float       intersection_y;
      float       intersection_z;
      float       normal_vec_i;
      float       normal_vec_j;
      float       normal_vec_k;
      float ray_length;
      float ray_start_i;
      float       ray_start_j;
      float       ray_start_k;
      float       ray_vector_i;
      float       ray_vector_j;
      float       ray_vector_k;
      float       reflection_vector_i;
      float       reflection _vector_j;
      float       reflection _vector_k;
      int   previous_node;
      int   next_node;
}
```

If the previous_node value is -1 this is the beginning of a ray history. If the next_node has a value of -1 this is the last intersection point for this ray history. Using the first intersection point and the normal of each ray history one can create a shaded image of the target under test. This can be a good tool for evaluating the ray tracer.

4.2.2.3.2 Image Formation Benchmark Output

The outputs for the Image Formation Benchmark will be, float or double, binary files output at each stage of the process. The final binary file, after detection, is the only file that is not complex. When displaying these images one should us a dB scaling do to the nature of the detected image. Using -40dB down from the peak value will provide a good-looking image. Other conversions that don't account for the wide bandwidth in the final image may hide defects. When doing a timing run, only the final detected or real valued image should be output. The in-between images are used only to validate each step in the process.

4.2.2.4 Acceptance Test

A given data set will be considered successfully executed when the processing sequence results match with the corresponding output provided with the benchmark. Precision is discussed in Section 3.7.

4.2.2.4.1 Acceptance Test for Recursive Ray Tracer Benchmark

It must be realized that small differences in intersection location and in the calculation of the normal can result in large errors after several reflections. Acceptance of this section of the benchmark should look at each level in the ray history to see if it meets the tolerances discussed in Section 3.7.

4.2.2.4.2 Acceptance Test for the Image Formation Benchmark

An output data set for each step in the process is provided and a match should be achieved for each step in the process.

4.2.2.5 Metrics

4.2.2.5.1 Metrics for Recursive Ray Tracer Benchmark

The primary metric for the Recursive Ray Tracer Benchmark is the total time to complete the evaluation of all rays in the given aperture. Secondary metrics are based on processor loading as a function of time. Also were possible a metric should be attempted that gives the scalability ratios for input database size, aperture resolution, and number of processors. These secondary metrics will provide useful information on known problems with parallel ray-tracing applications.

4.2.2.5.2 Metrics for the Image Formation Benchmark

The primary metric for the Image Formation benchmark is total time to complete all the steps with the given input data set. A secondary metrics consists of the individual times for each step in the Image Formation process.

4.2.2.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.2.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.2.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.2.9 References

Ray Tracing References

[1] K. Bouatouch and T. Priol. Parallel space tracing: An experience on an iPSC hypercube. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International '88)*, pages 170–187, New York, 1988. Springer-Verlag.

[2] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, pages 3–12, 1986.

[3] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, and K. Sims. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[4] M. A. Z. Dipp´ e and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *ACM Computer Graphics*, 18(3):149–158, jul 1984.

[5] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, pages 12–27, nov 1989.

[6] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4(4):197–209, 1988.

[7] A. J. F. Kok. *Ray Tracing and Radiosity Methods for Photorealistic Image Synthesis*. PhD thesis, Delft University of Technology, jan 1994.

[8] T. T. Y. Lin and M. Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7:187–199, 1991.

[9] D. J. Plunkett and M. J. Bailey. The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, aug 1985.

[10] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing on a MIMD hypercube. *The Visual Computer*, 5:109–119, 1989.

[11] E. Reinhard. Hybrid scheduling for parallel ray tracing. TWAIO final report, Delft University of Technology, jan 1996.

[12] I. D. Scherson and C. Caspary. A self-balanced parallel ray-tracing algorithm. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, volume 4, pages 188–196, Wokingham, 1988. Addison-Wesley Publishing Company.

[13] L. S. Shen, E. Deprettere, and P. Dewilde. A new space partition technique to support a highly pipelined parallel architecture for the radiosity method. In *Advances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware*. Springer-Verlag, 1990.

[14] E. R. Frederik, W. Jansen . Rendering Large Scenes Using Parallel Ray Tracing. *Parallel Computing*, pages 873-885, 1997

[15] T. Wilson, N. Doe. Acceleration Schemes for Ray Tracing. Report Number: CS-TR-92-22, Department of Computer Science, University of Central Florida, September 1992.

[16] R.L. Cook, T. Porter, L. Carpenter. Distributed Ray Tracing. *Computer Graphics* (Proceedings of SIGGRAPH 1984), 18(3), 137-145, July 1984.

[17] R.L. Cook. Stochastic sampling in computer graphics, ACM Transaction in Graphics 5(1), 51-72, January 1986.

[18] A. S. Glassner (Editor), *An Introduction to Ray Tracing*, Academic Press 1989.

[19] Ray Tracing Bibliography, http://www.cm.cf.ac.uk/Ray.Tracing/RT.Bibliography.html

Simulated SAR References

[1] D.J. Andersh, M. Hazlett, S.W. Lee, D.D. Reeves, D.P. Sullivan and Y. Chu, "Xpatch: A high fre-quency electromagnetic-scattering prediction code and environment for complex three-dimensional objects," *IEEE Antennas & Propagation. Magazine*, vol. 36, pp.65-69, 1994.

[2] J. Baldauf, S.W. Lee, L. Lin, S.K. Jeng, S.M. Scarborough, and C.L. Yu, "High frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiment," *IEEE Transacrions on Antennas and Propagation*, vol. 39, pp. 1345-1351, 1991.

[3] R. Bhalla and H. Ling, *Image-domain ray tube integration formula for the shooting and bouncing ray technique*, University of Texas Report, NASA Grant NCC 3-273, July 1993.

[4] R. Bhalla and H. Ling, "A fast algorithm for signature prediction and image formation using the shooting and bouncing ray technique," to appear in *IEEE Transactions on Antennas and Propagation*, 1995.

[5] G. Franceschetti, M. Migliaccio, D. Riccio, and G. Schirinzi, "SARAS: A Synthetic Aperture Radar (SAR) Raw Signal Simulator," *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 30, No. 1, January 1992.

[6] G. Franceschetti, M. Migliaccio, and D. Riccio, "SAR Raw Signal Simulation of Actual Ground Sites in Terms of Sparse Input Data," *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 32, No. 6, November 1994.

[7] D.E Herrick and I.J. LaHaie, *SRIM Polarimetric Signature Modeling*, ERIM IR&D Final Report 675805-1-F, December 1988.

[8] D.E Herrick and B.J. Thelen, "Computer Simulation of Clutter in SAR Imagery," *Proceedings of the Progress in Electromagnetics Research Symposium*, Cambridge, MA, July 1991

[9] D.E Herrick, "Computer Simulation of Polarimetric Radar and Laser Imagery," in *Direct and Inverse Methods in Radar Polarimetry,* W.-M. Boerner *et al.* (eds), Klumer Academic Publishers, The Netherlands 1992.

[10] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Modeling of Foliage Effects in UHF SAR", *Proceedings qfthe Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[11] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Synthesizing SAR Signatures of Ground Vehicles with Complex Scattering Mechanisms", *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[12] E.R. Keydel, D.E *Henick,* and W.D. Williams, "Interactive Countermeasures Design and Analysis Tool," *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[13] S.W. Lee and D.J. Andersh, *On Nussbaum Method for Exponential Series,* Electromagnetic Laboratory Technical Report ARTI-92-11, University of Illinois, Urbana, November, 1992.

[14] H. Ling, R.C. Chou, and S.W. Lee, "Shooting and Bouncing Rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Transactions on Antennas and Propagation,* vol. 37, pp. 194-05, 1989.

[15] J.M. Nasr and D. Vidal-Madjar, "Image Simulation of Geometric Targets for Spaceborne Synthetic Aperture Radar", *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 6, November 1991.

[16] N.D. Taket, S.M. Howarth, and R.E. Burge, "A Model For the Imaging of Urban Areas by Synthetis Aperture Radar," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 3, May 1991.

[17] M.R. Wohlers, S.Hsiao, J. Mendelsohn, and G. Gerdner, "Computer Simulation of Synthetic Aperture Radar Images of Three-Dimensional Objects," *IEEE Transactions on Aerospace and Electronic Systems,* Vol. AES-16, No. 3, May 1980.

4.2.3 Image Understanding

Algorithms were selected for this benchmark that perform spatial filtering to determine regions of interest (ROIs) and operate on a set of ROIs. The *Image Understanding* benchmark consists of a sequence of components depicted in Figure 4.2.3-1. Also included in the figure are names for input parameters, images, and intermediate output at different parts of the sequence, which are referred to later in this document. The morphological filter component provides a spatial filter to remove background clutter in the image. Next, the ROI selection component applies a threshold to determine target pixels, groups these pixels into ROIs, and selects a subset of ROIs depending on specific selection logic. Finally, the feature extraction component operates over and computes features for the selected ROIs.

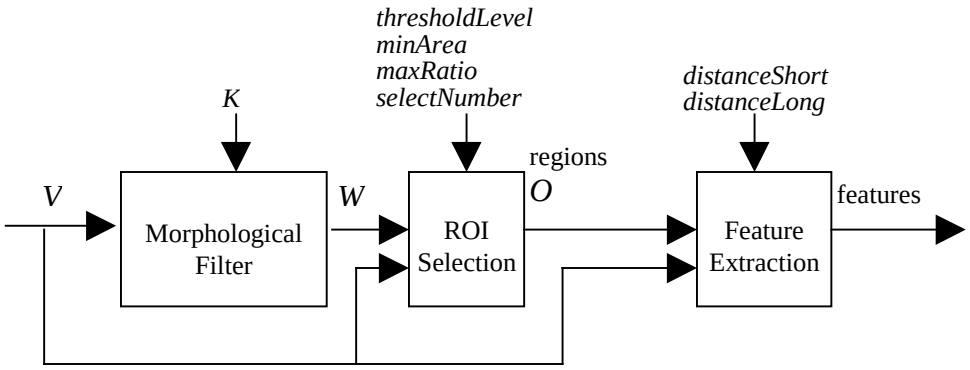


Figure 4.2.3-1: Image Understanding Sequence

The input required by the sequence is a set of parameters and an image, *V*. The first step in the sequence is a spatial morphological filter component generating image *W*. Then, the ROI selection component performs a thresholding and groups connected pixels into ROIs (or targets) contained in image *W*. This component then computes initial features for each ROI in image *W,* and selects a list of ROIs depending on the values of these features. These selected ROIs are stored in object image, *O*. The initial features for each selected ROI are stored in list, *regions.* Lastly, the feature extraction component computes additional features for the selected ROIs. The output at the end of the sequence is a feature list, *features,* with both sets of features computed for each selected ROI.

4.2.3.1 Input

An input data set for the Image Understanding benchmark, which contains all input required for a single run of the benchmark, is provided in one binary input file, in the following order:

1. input image *V* (stored as an array of short integers)
2. morphological kernel *K* (stored as an array of unsigned bytes)
3. *thresholdLevel* (short integer)

4. minimum acceptable area value, *minArea* (integer)
5. maximum acceptable perimeter-to-area ratio, *maxRatio* (float)
6. number of ROIs to select, *selectNumber* (integer), and
7. *distanceShort* and *distanceLong* (integers).

The above values are provided in binary representation for an 8-bit unsigned byte, 2-byte short integer, 4-byte integer, or 4-byte float as described in Section 3.6. References to these inputs are found in Figure 4.2.3-1 and in the sections describing each component of the sequence below. Descriptions or formats for input image V and kernel K are provided in the following subsections.

4.2.3.1.1 Image V

Images provided as input for the benchmark are rectangular, with square pixels, and stored in row-dominant order. Let image V have X columns and Y rows, and v(x,y) be any pixel in V where $0 \le x < X$ and $0 \le y < Y$, as shown here:

v(0,0)	v(1,0)	v(2,0)	• • •	v(X-1,0)
v(0,1)	v(1,1)	v(2,1)	• • •	v(X-1,1)
v(0,2)	v(1,2)	v(2,2)	• • •	v(X-1,2)
•	•	•		•
•	•	•		•
•	•	•		•
v(0,Y-1)	v(1,Y-1)	v(2,Y-1)	• • •	v(X-1,Y-1)

Figure 4.2.3-2: Sample Image V with X columns and Y rows

Preceding the image data are two *integers*, representing the number of columns and the number of rows, respectively. Next, pixel values are provided, as *short integers*, in row-dominant order, as shown in the following table. Therefore, an image with three rows and five columns requires thirty-eight bytes, where the first eight bytes contain two 4-byte *integer* values indicating the number of columns and rows, followed by fifteen 2-byte *short integer* pixel values.

Table 4.2.3-1. File containing byte image V.

byte offset ↓	contents	byte offset ↓	contents	byte offset ↓	contents
0		**6+2*X***	*v(X-1,0)*	•	•
2	*X*	**8+2*X***	*v(0,1)*	•	•
4		•	•	**6+2*XY*-2*X***	*v(X-1,Y-2)*
6	*Y*	•	•	**8+2*XY*-2*X***	*v(0,Y-1)*
8	*v(0,0)*	**6+4*X***	*v(X-1,1)*	•	•
10	*v(1,0)*	**8+4*X***	*v(0,2)*	•	•
•	•	•	•	**4+2*XY***	*v(X-2,Y-1)*
•	•	•	•	**6+2*XY***	*v(X-1,Y-1)*

4.2.3.1.2 Kernels

Kernels are small images that define a neighborhood or window to be used in the processing of a larger image. Kernels are provided in the same format as images and an example of the file containing a kernel is contained in Table 4.2.3-2 below. Note that a kernel with three rows and five columns requires twenty-three bytes, where the first eight bytes contain two 4-byte *integer* values indicating the number of columns and rows, followed by 15 *unsigned-byte* pixel values.

Table 4.2.3-2: File Containing Unsigned Byte Kernel *K*

byte offset ↓	contents	byte offset ↓	contents	byte offset ↓	contents
0		**8**	*k(0,0)*	•	•
1	*X*	**9**	*k(1,0)*	•	•
2		•	•	**7+*XY*-*X***	*k(X-1,Y-2)*
3		•	•	**8+*XY*-*X***	*k(0,Y-1)*
4		**7+*X***	*k(X-1,0)*	•	•
5	*Y*	**8+*X***	*k(0,1)*	•	•
6		•	•	**6+*XY***	*k(X-2,Y-1)*
7		•	•	**7+*XY***	*k(X-1,Y-1)*

A kernel-oriented procedure uses the location of the kernel's center pixel as the location in the output image for the output value. To facilitate this, kernel rows and columns always contain an odd number of pixels.

4.2.3.2 Algorithmic Specification

The *Image Understanding* benchmark consists of a series of operations to be performed, in sequence, using given image and operating parameters as initial input. Output from the benchmark consists of a table of feature values for each ROI selected. All of the operations utilize the result from the prior step as input. In addition, the ROI selection and feature extraction components utilize both the prior result and the input image, *V*, as depicted in Figure 4.2.3-1. Each component in the sequence is described individually below.

4.2.3.2.1 Morphological Filter

The morphological filter component chosen for the benchmark uses a structuring element, or kernel, *K*, that is a two-dimensional image with dimensions that are odd. Let *V* represent the input image and define the morphological operations, *erosion* ($\ominus$) and *dilation* ($\oplus$) as follows:

$$[V \ominus K] = MIN[v(x+m, y+n)] \quad m,n \in Ros(K), k(m,n) \neq 0 \qquad (4.2.3.1a)$$

$$[V \oplus K] = MAX[v(x+m, y+n)] \quad m,n \in Ros(K), k(m,n) \neq 0 \qquad (4.2.3.1b)$$

where each output pixel is computed at location (*x*, *y*) for a morphological kernel, *K*, which has a local region of support (Ros) that defines its geometric filtering properties with *M* columns and *N* rows. For these primitive morphological operations, MAX and MIN are computed locally for every pixel. Only nearby pixels are required to compute output pixels, specifically for the pixels in *K* that are non-zero.

For this benchmark, the morphological filter is defined as follows. As shown in Figure 4.2.3-1, *V* is the input image and *W* is the output, where

$$W = V - [(V \ominus K) \oplus K] \qquad (4.2.3.2)$$

A detailed discussion on morphology can be found in [Maragos] and [Parker 97]. The pixel values for input kernel *K* will be provided with *unsigned byte* precision and the pixels in the input image *V* will be provided with *short integer* precision as discussed in Section 4.2.3.1. The pixels in the output image *W* are required to have a minimum of *short integer* precision. Pseudo-code for the morphological filter component follows.

```
/* morphological filter component */
Get image V, kernel K
Clear images W, O1 and O2 (set to 0)
/* First calculate erosion O1 = V ⊖ K */
Loop for each pixel V(x,y) where (M-1)/2<=x<X-(M-1)/2 and
      (N-1)/2<=y<Y-(N-1)/2
      initialize minval
      Loop for each non-zero pixel k(m,n)
            minval = MIN[ minval, V(x+m,y+n)]
      End loop
      o1(x,y) = minval
End loop
/* Next calculate dilation O2 = O1 ⊕K */
Loop for each pixel o1(x,y) where (M-1)/2<=x<X-(M-1)/2 and
      (N-1)/2<=y<Y-(N-1)/2
      initialize maxval
```

```
            Loop for each non-zero pixel k(m,n)
                  maxval = MAX[ maxval, o1(x+m,y+n)]
            End loop
            o2(x,y) = maxval
      End loop
      /* Last output W = V -  O2 */
      Loop for each pixel in w(x,y) where M-1<=x<X-M+1 and N-1<=y<Y-N+1
      W(x,y) = V(x,y) - o2(x,y)
      End loop
```

Note that, after processing, the valid output region will be smaller than the valid input region, since there is not enough valid input data at the outer edges to calculate valid output. In particular, for a given kernel with *M* columns and *N* rows, the outer frame of invalid data will be *a* rows at the top and bottom and *b* columns at the left and right of the image, where *a* and *b* are defined by

$$a = \frac{N-1}{2}, \qquad b = \frac{M-1}{2} \qquad (4.2.3.3)$$

This effect accumulates during a process involving sequential steps, so that the final output will have an outer frame of undefined data equal in size to the sum of all the edge effects from each step within the process. This undefined outer frame should be set to the value 0 (zero).

For example, if a simple 3x3 kernel is used for the morphological filter (performing an erosion followed by a dilation), both *M* and *N* equal three. In that case, *a* and *b* are $(N-1)/2 = (M-1)/2 = 1$ pixel for each erosion or dilation. Therefore, the frame of undefined data in the output at the end of the filter is the sum $1+1=2$ pixels in width.

4.2.3.2.2 ROI Selection

The ROI selection component uses the input image *V*, and input parameters *thresholdLevel*, *minArea*, *maxRatio*, and *selectNumber*, provided in the input file as discussed in Section 4.2.3.1. The image *W* from the morphological filter component is the final input required. This component applies a threshold to image *W*, using the input *thresholdLevel*, to differentiate target pixels from background pixels. Then, target pixels are grouped together to determine how many isolated ROIs have been found. This is achieved by traversing *W* and assigning each detected target pixel to an ROI. Areas that are connected to each other are considered part of the same ROI. Two areas are declared connected if any target pixel from one area is 8-adjacent with any target pixel in the other (i.e., if they are horizontal, vertical, or diagonal neighbors). Next, an initial feature extraction is performed on these ROIs. Finally, a subset of these ROIs is selected, based on the values of the initial features. This subset of selected ROIs is used to generate the output of this component: an image *O* of detected objects or ROIs, and a list, *regions*, which includes initial features for each selected ROI.

The output *O* does not need to be an image, but does need to contain enough information so that each selected ROI is differentiated from other ROIs, and so each pixel within an ROI can be referenced. For the sake of simplicity and readability, the baseline imple-

mentation of the benchmark generates *O* in the form of an image. The depth of the values in *O* is driven by the maximum number of possible ROIs ($(2^{16}-2)$ as specified below).

Other implementations of the benchmark need not constrain *O* to be an image, as long as the utility of *O* remains. For example, instead of a complete image containing all the ROIs, each labelled with a distinct index, a subimage or *chip* could be extracted for each ROI where the chip boundaries could be defined by the smallest rectangular region containing the ROI. Then a method of obtaining the location of the ROI relative to the filtered image *W* must also be retained (i.e., an offset to place the chip over the proper location in *W*). In this manner, there would be *selectNumber* chips and offsets to specify the selected ROIs. As another example, an ROI could be specified as a list of pixel locations.

The feature extraction process is split up into two stages: 1) an initial set of features is calculated during the ROI selection component, and 2) an additional set of features is computed during the feature extraction component. This split is frequently done in deployed systems in order to minimize computation. Features in the first stage are typically not as computationally expensive as those in later stages. The features from the first stage are often used, as in this benchmark, to cull the ROI list before continuing, so that the second stage features are not computed except where necessary.

Implementations of this component are required to handle at most $(2^{16}-2)$ number of ROIs (before going through the selection logic). Using 8-adjacent connectivity, the theoretical maximum number of distinct ROIs is approximately one quarter the number of pixels in the entire original image. Should there exist more than $(2^{16}-2)$ ROIs in *W*, the ROIs beyond this number may be ignored.

Initial features for this component are extracted for each ROI that is found within *W*. The features are *centroid, area, perimeter, mean,* and *variance*. See [Parker 94] or [Castleman] for a detailed description of these features. The first three–*centroid, area,* and *perimeter*–measure properties of the ROI defined after the threshold is applied to *W*. Let *T* be an image, the same size as *W*, defined herein to be the value 1 over the target or ROI in question, and 0 elsewhere.

The *centroid* is the location that is central to the ROI, represented as *T*, and is computed using the following equations:

$$centroidCol = \frac{\sum t(x,y) * x}{area(T)}$$

$$x, y \in Ros(T) \qquad (4.2.3.4)$$

$$centroidRow = \frac{\sum t(x,y) * y}{area(T)}$$

where the *x* and *y* locations are relative to the pixel coordinates defined for the registered image *W* with column and row ranges $0 \le x < X$ and $0 \le y < Y$.

The *area* is a count of the number of target pixels contained in the ROI.

The *perimeter* is a count of the number of pixels on the target that are 8-adjacent to a background pixel. To find this value, let two pixels be defined as 8-adjacent if they are horizontal, vertical, or diagonal neighbors.

Both the thresholded version of *W* and the input image *V* are used to calculate the *mean* and *variance* features. These features are statistical measures computed for the pixel values in each ROI using the equations:

$$mean = \frac{\sum v(x,y)t(x,y)}{pixelCount} \qquad x,y \in Ros(T)$$

$$variance = \frac{\sum [v(x,y)t(x,y)]^2}{pixelCount} - mean^2 \qquad x,y \in Ros(T) \qquad (4.2.3.5)$$

where the summation for the *mean* and *variance* is calculated over a single ROI, and *pixelCount* is the number of pixels in that ROI (represented above as *T*).

Once the initial features have been calculated, selection logic using these features creates a subset of selected ROIs that are retained. The three input parameters–*minArea, maxRatio,* and *selectNumber*–specify the selection criteria. The features calculated in the ROI selection component–*area, mean,* and *perimeter*–are used with these input parameters to determine whether an ROI will be selected. Any ROI with a feature that does not pass the selection criteria is removed from the ROI list. Two of the selection tests are defined below.

$$minArea \le area$$
$$\frac{perimeter}{area} \le maxRatio \qquad (4.2.3.6)$$

The final selection test requires ranking the list of ROIs by the value of the product (*mean* * *area*) from largest to smallest value and selecting the top *selectNumber* of ROIs that also satisfy the selection tests in Equation 4.2.3.6 above. Thus, the *selectNumber* ROIs with the largest product *mean*area,* that also satisfy the selection tests in Equation *4.2.3.6,* compose the set of selected ROIs.

Then, for the labelled ROIs, an output image, *O,* and a list, *regions,* must be constructed. Image *O* defines the shapes and locations of the selected ROIs, and *regions* contains a list of the selected ROIs including the initial features.

The output image, *O,* is required to have a minimum of *short integer* precision, to handle labels for at most $(2^{16} - 2)$ ROIs. The list shown as *regions* in Figure 4.2.3-1 must include the six features described in this section for each ROI identified in image *O.* The floating-point features–*centroidCol, centroidRow, mean,* and *variance*–are required to have a minimum of *float* precision. The other features–*area* and *perimeter*–are required to have a minimum of *integer* precision. Pseudo-code for a correct–though inefficient–implementation of the ROI selection component follows.

```
/* ROI selection component */
```

```
/* V is input image */
/* W is morphological filtered image */
/* thresholdLevel is level to use to determine target pixels */
/* minArea, maxRatio, and selectNumber are parameters
      to be used in ROI selection logic */
Get images V and W
Get parameters thresholdLevel, minArea, maxRatio, selectNumber
Clear image G (set to 0)
nt = 0
/* first threshold image W to determine target pixels
   and group pixels belonging to the same ROI, by
   marking each ROI with a unique id  */
Loop for each (x,y) in W      /* scan filtered image */
      If w(x,y) > thresholdLevel then     /* if target pixel */
            Loop for (u,v) in 8 neighbors of (x,y)
                  If g(u,v) > 0 then   /* if already tagged */
                        If g(x,y) > 0 and g(x,y) ≠ g(u,v) then
                              /* we are connecting two ROIs */
                              gt = g(u,v)
                              Loop for each (i,j) in G up to and
                                    including (x,y)
                                    If g(i,j) = gt then
                                          g(i,j) = g(x,y)
                                    Endif
                              End loop
                        Else        /* first tagged neighbor */
                              g(x,y) = g(u,v)
                        Endif
                  Endif
            End loop
            If g(x,y) = 0     /* ROI is isolated */
                  If (nt > (2^16-2)) then
                        g(x,y) = 0 /* ignore >(2^16-2) ROIs */
                  Else
                        nt = nt + 1 /* increment ROI count */
                        g(x,y) = nt /* label ROI with new id */
                  Endif
            Endif
      Endif
End loop

/* Compute initial features for each ROI */
/* F is list of initial features */
Initialize F
Loop for each ROI in G
      clear centroidCol, centroidRow, area, perimeter,
            mean, and variance
      Loop for each pixel g(x,y) in ROI
            centroidCol = centroidCol + x
            centroidRow = centroidRow + y
            area  = area  + 1
            If pixel is 8-adjacent to background pixel
                  then perimeter = perimeter + 1
            Endif
            mean = mean + v(x,y)
            variance = variance + v(x,y)*v(x,y)
      End loop
      centroidCol = centroidCol / area
```

```
            centroidRow = centroidRow / area
            mean = mean / area
            variance = (variance/area) -mean^2
            add features to list F
      End loop

      /* use selection logic to select subset of ROIs */
      order list F ranking mean*area value from largest to smallest
      numROIs = 0
      initialize list regions
      clear image O
      Loop for each ROI on list F
            If (minArea <= area) AND
               (perimeter/area <= maxRatio) AND
               (numROIs < selectNumber) Then
                  numROIs = numROIs + 1
                  add ROI to object image O
                  add initial features to list regions
            Endif
      End loop for each ROI on list F
```

4.2.3.2.3 Feature Extraction

In the final component of the sequence, additional features are calculated for the ROIs selected from the previous component. As shown in Figure 4.2.3-1, two input parameters are provided in the input file for this component. The parameters–*distanceShort* and *distanceLong*–are provided in *integer* precision as discussed in Section 4.2.3.1. The input image, *V,* the object image, *O,* and the list, *regions,* complete the inputs required for this component. The input *O* does not need to be an image, but does need to contain enough information so that each selected ROI is differentiated from each other and so each pixel within an ROI can be referenced. In this implementation, this is achieved by having *O* be an image. The depth of the values in *O* is driven by the maximum number of ROIs possible $(2^{16} - 2)$.

The additional features calculated in this component give a measure of the texture of each ROI. As discussed in [Parker 97], a grey level co-occurence matrix (GLCM) contains information about the spatial relationships between pixels within an image. Statistical descriptors of the co-occurence matrix have been used as a practical method for utilizing these spatial relationships. Furthermore, [Unser] designed a method of estimating these descriptors without calculating the GLCM, instead using sum and difference histograms. The features to be calculated here are *GLCM entropy* and *GLCM energy*, and are defined in terms of a sum histogram, *sumHist,* and a difference histogram, *diffHist.* These histograms are dependent on a specific distance and direction just as the GLCM. The sum histogram, *sumHist*, is a normalized histogram of the sums of all pixels at a given distance and direction. Likewise, the difference histogram, *diffHist,* is a normalized histogram of the differences of all the pixels at a given distance and direction. The GLCM descriptors are defined as:

$$GLCM\ entropy\ =\ -\sum_i sumHist(i) * log[sumHist(i)] - \sum_j diffHist(j) * log[diffHist(j)] \qquad (4.2.3.8)$$

$$GLCM\ energy\ =\ \sum_i sumHist(i)^2 * \sum_j diffHist(j)^2$$

where *sumHist(i)* is the normalized sum histogram and *diffHist(j)* is the normalized difference histogram for the particular distance and direction of interest.

For this benchmark, rather than calculate these measures for all possible distances and directions, two distances are given in the input file, and four directions of interest must be used. These directions are defined as: 0°, 45°, 90°, and 135°. Therefore, the feature extraction component will compute, for each selected ROI, a total of sixteen features (two descriptors at each of two distances and four directions). Both of the descriptors – *GLCM entropy* and *GLCM energy* – are required to have a minimum of *float* precision.

The final output of the benchmark is a feature list, *features*, containing all twenty-two features (both initial and additional features), for each selected ROI. Pseudo-code for a correct–though inefficient–implementation of the feature extraction component follows.

```
/* feature extraction component */

/* V is the input image */
/* O is object image */
/* regions is list which includes initial features */
/* features is list of all features
                  (initial and additional features) */
Get images V and O
Get list regions
Initialize features
numROI = 0
Loop for each ROI in O  /* scan object image */
      numROI = numROI + 1 /* keep track of number of ROIs */
      Get initial features for ROI from list regions
      Loop for distance = distanceShort and distanceLong
            /* for 0 degree direction (horizontal) */
            dx = distance
            dy = 0
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,0°) to features
            /* for 45 degree direction (right diagonal) */
            dx = distance
            dy = distance
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,45°) to features
            /* for 90 degree direction (vertical) */
            dx = 0
            dy = distance
            call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
            add all features(distance,90°) to features
            /* for 135 degree direction (left diagonal) */
            dx = - distance
            dy = distance
```

```
                call calcDescriptors(V,O,numROI,dx,dy,energy,entropy)
                add all features(distance,135°) to features
          End loop for distances
     End ROI loop
     End feature extraction

     routine calcDescriptors( V, O, numROI, dx, dy, energy, entropy)
     numlevels     = (number grey-levels in image)
     numhistlevels = 2*numlevels
     get array sumHist size numhistlevels, initialize to 0
     get array diffHist size numhistlevels, initialize to 0

     totalnumpixels = 0
     Loop for each pixel in ROI numROI
          /* calculate sum and difference histograms */
          If (x,y) AND (x+dx,y+dy) legal pixel addresses in ROI Then
                Increment sumHist( [v(x,y) + v(x+dx,y+dy)] )
                Increment diffHist( numlevels+[v(x,y)-v(x+dx,y+dy)])
                Increment totalnumpixels
          Endif legal pixel address
     End loop for numROI
     /* normalize sumHist and diffHist */
     Loop for i = 0, numhistlevels
          sumHist(i) = sumHist(i) / totalnumpixels
          diffHist(i) = diffHist(i) / totalnumpixels
     End loop for i
     energyS = 0
     energyD = 0
     entropy = 0
     /* calculate descriptors from sumHist and diffHist */
     Loop for i = 0, numhistlevels
          entropy = entropy - sumHist(i)*log(sumHist(i))
                            - diffHist(i)*log(diffHist(i))
          energyS = energyS + sumHist(i) * sumHist(i)
          energyD = energyD + diffHist(i) * diffHist(i)
     End loop for i
     energy = energyS * energyD
     End routine calcDescriptors
```

4.2.3.3 Output

The output for the *Image Understanding* benchmark should be provided as a char (7-bit ASCII stored in 8-bit bytes) file where there is one line of ASCII text for each selected ROI. This entry should contain all the features calculated for that ROI: six initial features from the ROI selection component and sixteen additional features from the feature extraction component. The format for each entry is described in the following table.

Field	Description	Type	Format
1	centroidCol	*float*	*m.dddd* E±*xx*
2	centroidRow	*float*	*m.dddd* E±*xx*
3	area	*integer*	*dddddd*

4	perimeter			*inte-ger*	*dddddd*
5	mean			*float*	*m.dddd* E±*xx*
6	variance			*float*	*m.dddd* E±*xx*
7	DistanceShort	0°	GLCM entropy	*float*	*m.dddd* E±*xx*
8			GLCM energy	*float*	*m.dddd* E±*xx*
9		45°	GLCM entropy	*float*	*m.dddd* E±*xx*
10			GLCM energy	*float*	*m.dddd* E±*xx*
11		90°	GLCM entropy	*float*	*m.dddd* E±*xx*
12			GLCM energy	*float*	*m.dddd* E±*xx*
13		135°	GLCM entropy	*float*	*m.dddd* E±*xx*
14			GLCM energy	*float*	*m.dddd* E±*xx*
15	DistanceLong	0°	GLCM entropy	*float*	*m.dddd* E±*xx*
16			GLCM energy	*float*	*m.dddd* E±*xx*
17		45°	GLCM entropy	*float*	*m.dddd* E±*xx*
18			GLCM energy	*float*	*m.dddd* E±*xx*
19		90°	GLCM entropy	*float*	*m.dddd* E±*xx*
20			GLCM energy	*float*	*m.dddd* E±*xx*
21		135°	GLCM entropy	*float*	*m.dddd* E±*xx*
22			GLCM energy	*float*	*m.dddd* E±*xx*

Table 4.2.3-3: Output Record Specification for Each ROI

One space (*char* value 32) should be used to delimit each field, and a carriage return/line feed should follow the last field for each ROI.

4.2.3.4 Acceptance Test

The software implementation will be considered successful for a given input data set when the implementation is executed for the given input data set and produces results that match with the corresponding output provided with the benchmark. Precision is discussed in Section 3.7.

4.2.3.5 Metrics

The primary metric associated with the *Image Understanding* benchmark is the total time required to run a given input data set through the Image Understanding sequence gener-

ating accurate results. A series of secondary metrics for the individual times of the processing components is also required. The time associated with a processing component is defined as the time at the beginning of one part in the flow to the beginning of the next part in the flow. For example, the time to perform the Morphological Filter component is the time it takes to apply the morphological filter to the input image *V* and obtain the output image *W*, not including time to read input image, *V*, and kernel, *K*.

4.2.3.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.3.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.3.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.3.9 References

[Castleman] Castleman, K., *Digital Image Processing*, Prentice-Hall, 1979.

[Maragos] Maragos, P., "Tutorial on advances in morphological image processing and analysis," *Optical Engineering*, vol. 26, no. 7, pp. 623-632, July 1987.

[Parker 94] Parker, J., *Practical Computer Vision Using C*, Wiley, 1994.

[Parker 97] Parker, J., *Algorithms For Image Processing And Computer Vision*, Wiley Computer Publishing, 1997.

[Unser] Unser, M., "Sum and Difference Histograms for Texture Classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, 1:118-125, 1986.

[Weeks] Weeks, A., *Fundamentals of Electronic Image Processing*, SPIE/IEEE series on imaging science & engineering, 1996.

4.2.4 Multidimensional Fourier Transform

The Fourier Transform has wide application in a diverse set of technical fields. It is utilized in image processing, convolution and deconvolution, and digital signal filtering to name a few. This benchmark attempts to measure the performance of a typical range of transforms using the candidate hardware configurations.

The DIS *Fourier Transform Benchmark* consists of applying a three-dimensional Discrete Fourier Transform (DFT) to a series of transform tests, all of which have the same number of dimensions, but may have different sizes and are repeated a specified number of times. The three-dimensional DFT is defined by

$$F(x,y,z) = \sum_{k_3=0}^{Z} \sum_{k_2=0}^{Y} \sum_{k_1=0}^{X} e^{2\pi i k_3 z/Z} e^{2\pi i k_2 y/Y} e^{2\pi i k_1 x/X} f(k_1, k_2, k_3) \qquad 4.2.4.1$$

where f is the input complex three-dimensional array of size $X \times Y \times Z$, and F is the output forward transform of f of the same size. An individual transform test consists of a specification of the size of the input array and the number of times the input array is transformed into the output array. Note that this benchmark does not specify any recursive application of the DFT (for example, applying the transform, calculating the inverse, and applying the transform to the result). Thus, the input array, f, should not be overwritten during the calculation of the transform output, F.

The size of the three-dimensional data array to be transformed is specified by an input file (i.e., the values of X, Y, and Z), but the initial values for the array data, f, are not provided. The DFT computes the transform of a three-dimensional array of complex floats, but the computational speed or efficiency of the DFT is not data dependent. Because of this separation between data and performance, and to reduce the size of the input set, only the lengths of the three dimensions are specified. The values for the array should be randomly initialized once for each transform test. The characteristics of the random input initialization is left to the individual implementers with the conditions that the random number generator have a period larger than 2^{32} and the values generated lie within the bounds detailed for *floats* in Section 3.6 (Common Data Types). Note that almost all implementations of the standard UNIX *rand()* function satisfy these conditions.

4.2.4.1 Input

An input set for the *Fourier Transform Benchmark* is provided in a single ASCII text file as a list of transform tests. All values within the input file are integers and are white-space-delimited (here "white space" indicates carriage returns, line feeds, or spaces). The first value in the input file is an integer, which specifies the number of transform tests detailed in the file. The rest of the file consists of a series of four integers where each set of four specifies a transform test. The first three integers of a set are the lengths of the first, second, and third dimensions of the transform, respectively. The fourth integer of a set is the number of times to repeat this particular test. Table 4.2.4-1 shows a schematic of an input file where M is the number of transform tests detailed in the file and X, Y, and Z are the lengths of the first, second, and third dimensions, respectively.

Table 4.2.4-1: Fourier Transform Input Schematic

Value	Description	
integer	number of transform tests, M	
integer	length of 1st dimension, X	transform test 1
integer	length of 2nd dimension, Y	
integer	length of 3rd dimension, Z	
integer	number of iterations	
integer	length of 1st dimension, X	transform test 2
integer	length of 2nd dimension, Y	
integer	length of 3rd dimension, Z	
integer	number of iterations	
:	:	:
integer	length of 1st dimension, X	transform test M
integer	length of 2nd dimension, Y	
integer	length of 3rd dimension, Z	
integer	number of iterations	

An example of an input file is given in the table below. This example file specifies five separate transform tests: a 200x100x1 transform repeated 10,000 times, a 5000x100x1 transform repeated 5000 times, a 20,000x800x1 transform repeated 100 times, a 5000x200x2 transform repeated 250 times, and a 4000x1000x1 transform repeated 500 times.

Table 4.2.4-2: Fourier Transform Input Example

```
5
200     100     1     10000
5000    100     1     5000
20000   800     1     100
5000    200     2     250
4000    1000    1     500
```

number of transform tests, *M*
parameters for transform test 1
parameters for transform test 2
parameters for transform test 3
parameters for transform test 4
parameters for transform test 5

4.2.4.2 Algorithmic Specification

The discussion of the algorithmic specification for the DFT in this benchmark will be limited, as the amount of material freely available to the implementers is extremely large. Rather, a brief description of several FFT algorithms, with appropriate references, will be

given. The focus of the discussion is on the FFT methods since almost all transform implementations are FFT rather than a direct implementation of the DFT Equation 4.2.4.1. However, any implementation which yields valid results is acceptable. The algorithm descriptions provided here are not meant as a complete listing of all available or allowable methods; implementers are encouraged to use any methods that will demonstrate the advantages of their hardware configurations. Also, a brief discussion of implementing algorithms which require dimensions of a power of two using "zero padding" is given.

The majority of FFT methods transform the original DFT into a series of subproblems which achieves a lower computational complexity[Duhamel90]. The most common subproblem decomposition is to assume that the dimensions of the input array are powers of two. This allows the summations present in Equation 4.2.4.1 to be split into two subproblems [Cooley]. The even- and odd-numbered frequencies are separated and the problem is recursively split by two until the original transform of length $N = X \times Y \times Z$ is reduced to transforms of length one, which is simply the identity operation that copies its input number to its output slot. The total process is on the order of $N \log_2 N$ and requires a bit reversal either on the input or the output depending upon the specific algorithm. Methods that do the bit-reversal then build up the transform are generally called *decimation-in-time* (DIT) or Cooley-Tukey FFT methods. Methods that manipulate the input data and then do bit-reversal on the output values are generally called *decimation-in-frequency* (DIF) or Sande-Tukey FFT methods.

The same type of reasoning can be applied, but the recursive subdivision stopped, at higher powers of two (typically four and eight with this type of algorithm called radix-4 or radix-8 methods[Ganapa]). These small transforms are done using highly optimized code, which provides a modest but appreciable performance improvement. A combination of subproblems of lengths two, four, or eight are also possible, and are generally called split-radix methods[Duhamel84].

The division of the DFT into subproblems is not limited to powers of two, but can be applied using prime numbers[Rader] and combinations of powers of two and primes with relatively sophisticated decision trees to determine the "optimal" subproblem divisions for a given problem [Frigo], [Frigo99].

The subproblem division of Equation 4.2.4.1 into powers of two requires that most of the computations, especially complex multiplications, be done in the initial stages of the algorithms for the Sande-Tukey FFT methods. However, the Cooley-Tukey methods place most of the complex computation at the final stages of the algorithms. A combination of the DIT and DIF methods with a transition stage between the domains would then lead to computation savings which is the idea behind Decimation-In-Time-Frequency methods [Saidi].

Several FFT algorithms require that the input array have dimensions that are a power of two. One method for using these algorithms when the array dimensions are not powers of two is to use a technique called "zero padding". This technique simply increases the memory size of the original array to the next power of two and initializes the extra space to zero. The numerical accuracy of the DFT algorithm is essentially unaffected by these "extra" zeros, and the result should be identical to other DFT methods. The primary

trade-off is in terms of excess storage required for the technique that can become critical for large input arrays.

4.2.4.3 Output

The output of this benchmark consists of an ASCII text file indicating the "mean fractional error" of the individual transform tests. All values placed in the output file are white-space-delimited (see Section 3.6 for the definition of "white space"). The first value in the file is an integer that specifies the number of transform tests performed and should match the corresponding value from the input file. The float values for the "mean fractional error", ε_{mfe}, for each test are then listed in the order they were performed. Table 4.2.4-3 shows a schematic of an output file.

Table 4.2.4-3: Fourier Transform Output Schematic

Value	Description
integer	number of transform tests, M
float	mean fractional error of test 1
float	mean fractional error of test 2
:	:
float	mean fractional error of test M

The "mean fractional error", ε_{mfe}, is defined to be

$$\varepsilon_{mfe} = \frac{1}{XYZ}\sum_{z}\sum_{y}\sum_{x}\frac{2\left|f_{xyz} - \hat{f}_{xyz}\right|}{\left|f_{xyz}\right| + \left|\hat{f}_{xyz}\right| + \varepsilon} \tag{4.2.4.2}$$

where f is the original input to the transform, $\hat{f}$ is the inverse of the transform of f, i.e.,

$$\hat{f} = F^{-1}(f) = \frac{1}{XYZ}\sum_{k_3=0}^{Z}\sum_{k_2=0}^{Y}\sum_{k_1=0}^{X} e^{-2\pi i k_3 z/Z} e^{-2\pi i k_2 y/Y} e^{-2\pi i k_1 x/X} F(k_1, k_2, k_3) \tag{4.2.4.3}$$

which differs from Equation 4.2.4.1 by simply changing the sign within the exponents and dividing the result with the total size of the transform. The variable ε within Equation 4.2.4.2 is a small value used to prevent division by zero. The value ε_{mfe} is then a measure of the perturbation of the transformed data from the original. Note that it is not necessary to implement an inverse DFT to calculate $\hat{f}$. The array $\hat{f}$ can be calculated from the identity,

$$\hat{f} = F^{-1}(f) = F^{*}(f^{*}) \tag{4.2.4.4}$$

where * indicates complex conjugation.

4.2.4.4 Acceptance Test

A given input set will be considered successfully executed when each transform test successfully passes the output provided with the benchmark. An individual transform test is considered successfully executed when the value for ε_{mfe} is less than or equal to the value [To be supplied June 1999]. An input set is considered successfully executed when all of the individual transform tests pass this same level of required accuracy.

4.2.4.5 Metrics

There are three metrics for this benchmark. The first, and primary, is the total time required to complete the input set. This should include the time for each transform test as well as the I/O time required to load the randomly generated input and output the result. The total time should not include the time necessary for the generation of the random data. The second metric is the time required to complete the individual transform tests. Again, this time should include any I/O time for loading of data and output of results. The third metric measures the "mflops" [Johnson] of the individual transform tests. The "mflops" for a given transform is defined to be

$$"mflops" = \frac{5(X \times Y \times Z)\log_2(X \times Y \times Z)}{(\text{time for one DFT in } \mu\text{s})} \qquad (4.2.4.5)$$

where X, Y, and Z are the lengths of the first, second, and third dimensions, respectively. The rational behind using this metric is to provide a reasonable comparison between different architectures, implementations, and transform sizes. Note that the "mflops" is not the MFLOPS (millions of floating-point operations per second), but an estimate of that value which assumes a common baseline number of operations for any implementation as

$$5(X \times Y \times Z)\log_2(X \times Y \times Z) + \vartheta(N) \qquad (4.2.4.6)$$

which is the radix-2 Cooley-Tukey FFT[Cooley]. This third metric is common in the FFT literature and for more discussion of the reasoning behind the metric, the reader is referred to [Johnson].

4.2.4.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.4.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.4.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.4.9 References

[Duhamel90] Duhamel and Vetterli, "Fast Fourier transforms: a Tutorial Review and State of the Art," *Signal Processing*, vol. 19, pp.259-299, April 1990.

[Cooley] Cooley and Tukey, "An Algorithm for Machine Computation of Complex Fourier Series,," *Math. Comp.*, vol. 19, pp.297-301, April 1965.

[Ganapa] Ganapathiraju, Hamaker, Picone and Skjellum, "Analysis and Characterization of Fast Fourier Transform Algorithms," MS State High Performance Computing Laboratory, Oct. 1997.

[Duhamel84] Duhamel and Hoolomann, "Split Radix FFT Algorithm," *Electronic Letters*, vol. 20, pp.14-16, Jan 1984.

[Rader] Rader, "Discrete Fourier Transforms when the Number of Data Samples is Prime," *Proc. of the IEEE*, vol. 56, pp.1107-1108, June 1968.

[Frigo] Frigo and Johnson, The FFTW web page, http://theory.lcs.mit.edu/~fftw

[Frigo99] Frigo, "A Fast Fourier Transform Compiler," MIT Laboratory for Computer Science, Feb. 16, 1999.

[Saidi] Saidi, "Decimation-In-Time-Frequency FFT Algorithm," *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, vol. III, pp.453-456, Adelaide, Australia, April 1994.

[Johnson] Frigo and Johnson, The BenchFFT web page, http://theory.lcs.mit.edu/~benchfft

4.2.5 Data Management

The Data Management Benchmark measures application-level timing performance of typical DBMS. This benchmark focuses on index management and ad hoc or content-based queries since these two areas are the primary weaknesses of traditional DBMS.

The benchmark is implemented as a simplified object-oriented database with an R-Tree indexing scheme. The R-Tree index is a height-balanced containment structure that uses multidimensional hyper-cubes as keys. The intermediate nodes are built up by grouping all of the hyper-cubes at the lower level. The grouping hyper-cube of the intermediate node completely encloses all of the lower hyper-cubes, which may be points. The system must respond to a set of command operations: *Insert, Delete*, and *Query*, queries being either key-based or content-based. The commands are to be issued to the system in a batch form as a data set.

The *Insert* command operation places a new data object into the database with the specified attribute values. Each *Insert* command contains all the information contained by the data object, including the hyper-cube key and list of non-key attribute values.

The *Query* command operation searches the database and returns all data objects that are consistent with a list of input data attribute values. The input attribute values can specify

attribute values which are *key*, *non-key*, or both. A data object is consistent with the *Query* when the input values intersect the stored values of the data object.

The *Delete* command operation removes all objects from the database that are consistent with a list of input data attribute values. The types and conditions of the input attribute list, as well as the description for consistency, are the same as for the *Query* operation.

4.2.5.1 Input

The input for each test of this benchmark consists of one data set. All of the data sets share a common format. Each set is a 8-bit ASCII character file and consists of a series of sequentially issued commands delimited by a carriage return, i.e., each line of the file represents a separate command. Table 4.2.5-1 gives the command operations, the character code used to designate the command, the data placed after the command code on the rest of the line, the return expected from the application, and a brief description of the operation.

Table 4.2.5-1: Command Operations

Command	Code	Line Elements	Return	Description
Initialization	0	Fan Size	NULL	Initializes the index by specifying the fan of the tree.
Insert	1	Object Type Key Attribute Key Attribute : Key Attribute Non-Key Attribute Non-Key Attribute : Non-Key Attribute	NULL	Insert new entry into database. See below for discussion of the Object Type and key and non-key attributes.
Query	2	Attribute Code Attribute Value Attribute Code Attribute Value : Attribute Code Attribute Value	Data Object List	Return all data objects that are consistent with the input attributes specified. Note that attribute codes and values always appear as pairs.
Delete	3	Attribute Code Attribute Value Attribute Code Attribute Value : Attribute Code Attribute Value	NULL	Delete all data objects that are consistent with the input attributes specified. Note that attribute codes and values always appear as pairs.

Each data object has a set of attributes, where the first eight attributes are used by the R-Tree index as the key and represent two points that specify a hyper-cube. Each point consists of four 32-bit IEEE-formatted floating-point numbers denoting a four-dimensional point in Euclidean space as the T-position, X-position, Y-position, and Z-position. Thus, the index key, which consists of a "lower" and "upper" point, is eight 32-bit floating-point numbers. Note that a point in hyper-space can be thought of as time (T) and a three-dimensional point (X,Y,Z), but this is immaterial to this benchmark.

The total number of attributes assigned to a data object is the sum of the key and non-key attributes. The number of non-key attributes for a given data object is determined by the Object Type, and is given in the table below. The Object Type used by the *Insert* command specifies which of the three types of objects (Small, Medium, and Large) is being inserted by the operation. Table 4.2.5-2 gives the character/byte code and the number of non-key attributes for each data object type

Table 4.2.5-2: Data Object Types

Object Type	Code	No. of Non-Key Attributes
Small	1	10
Medium	2	17
Large	3	43

Data objects differ by the number of non-key attributes assigned to each. Each non-key data attribute has an identical format that primarily consists of an 8-bit NULL-terminated ASCII character sequence of maximum length 1024. Table 4.2.5-2 gives the number of non-key attributes assigned to each object type. The maximum sizes for the Small (*overhead* + 10 * 1024), Medium (*overhead* + 17 * 1024), and Large (*overhead* + 43 * 1024) data objects are known beforehand, but the total size of the database is not determined until the input is set. The database should be able to handle all three types of data object, in any permutation. Note that the object type specification is placed at the beginning of the *Insert* command as a convenience, since the number of attributes can be determined by reading until the next carriage return, i.e., the end of the command.

The *Delete* and *Query* commands each reference a specified attribute by means of an Attribute Code. The following table gives the attribute code sequence for both the key and non-key attributes. Also, each attribute is assigned a type and, if applicable, a name and units.

Table 4.2.5-3 Attribute Codes and Descriptions

		Attribute Code	Name	Type
		0	T	float
	Lower	1	X	float
	Point	2	Y	float

Key Attributes		3	Z	float
	Upper Point	4	T	float
		5	X	float
		6	Y	float
		7	Z	float
Non-Key Attributes	Small	8	property	char string
		9	property	char string
		:	:	:
		16	property	char string
		17	property	char string
	Medium	18	property	char string
		19	property	char string
		:	:	:
		23	property	char string
		24	property	char string
	Large	25	property	char string
		26	property	char string
		:	:	:
		49	property	char string
		50	property	char string

Only the *Query* commands result in a response from the database. The response is the set of data objects that are appropriate for the corresponding *Query*. A description of the response is given in the Output section of this benchmark specification.

The formal definition of each command input line is described in the following sections. Each command line represents separate pieces of data, which are ASCII space-delimited unless otherwise stated. References to integers and floats indicate 32-bit IEEE standards. The first piece of data for every command line is the command code, which indicates the specific operation. The rest of the line is relative to the command type and is described in detail below.

4.2.5.1.1 Initialization

The *Initialization* command appears only once per data set and is always the first command. Only two pieces of information is provided for the command where the first is the command type, which in this case is '0'. The second piece of data is the fan size, which is an integer. A diagram of the *Initialization* command line is given below:

int	*int*
code	fan size

4.2.5.1.2 Insert

The *Insert* command is different from the *Delete* and *Query* commands, in that the *Insert* command does not reference data attributes by the appropriate attribute code given in Table 4.2.5-3. The *Insert* command does not need to specify the attribute code, since all attributes are provided in the command line and are in the proper order. Thus, the *Insert* command input line does not use the attribute codes in order to reduce the size of the input data sets and to simplify the command line input.

The first piece of data in the command line is the command type, which is '1' for the *Insert* operation. The next piece of data represents the object type, and can be the character 1, 2, or 3, for Small, Medium, or Large, respectively. The next eight pieces of data make up the index key for the object as the floating-point values for T, X, Y, and Z, for the "lower" and "upper" hyper-points, respectively. The remaining data are the individual non-key attribute character sequences of the new object. Each attribute is space-delimited, and is of variable length. The number of attributes on the line is dependent upon the object type being read, and is given in Table 4.2.5-2. The maximum size for any attribute on the command line is 1024, although in practice the attributes will be smaller. A diagram of the *Insert* command line is given below:

int	*int*	*float*	*float*	*float*	*float*	*float*	*float*	*float*	*float*	*char*	*char*	⋯	*char*
code	type	key attributes								non- key attributes			

4.2.5.1.3 Query

The *Query* command returns all data objects within the current database that are consistent with the provided key and non-key attribute values. A data object is consistent when the object's attribute values "intersect" with the *Query* attribute values. The definition of intersection is different for the key and non-key attributes. The attributes that make up the index key are hyper-cubes and the definition for an intersection is an intersection of the respective hyper-cubes, i.e., an intersection occurs whenever the input hyper-cube and the stored hyper-cube share any of the hyper-space. The non-key attributes consist of character sequences and the definition of an intersection is when any part of the stored character sequence matches the entire input sequence.

The first piece of data is the command code, which is '2' for *Query*. The rest of the command line is a list of attribute code and value pairs. A diagram of the *Query* command line is given below:

int	*int*	*float / char*	…	*int*	*float / char*
code	attribute code	attribute value		attribute code	attribute value

The number of attribute code-value pairs ranges from 1 to 50, where an attribute code is never repeated in a single command line. The attribute value type depends upon the at-

tribute code where an attribute code between zero and seven indicates a float and an attribute code between eight and 50 indicates a character sequence. It is possible that a *Query* will specify an attribute code that is not applicable to a specific data object. For example, any attribute code greater than 17 for a Small object, or any attribute code greater than 24 for a Medium object. The query search values for these cases should be ignored, and should not prevent the candidate data object from inclusion in the *Query* solution.

A key query, a search which uses the R-Tree index to search the database, requires a full index key, i.e., all eight floating point values specifying the search hyper-cube. The *Query* command input line need not contain all eight values for the search. If so, the search hyper-cube uses "wild-card" values for the rest of the search hyper-cube that match all possible stored values. An example of an incomplete key *Query* is

2 3 0.0 7 10.0

which searches the current database for all data objects which were between the Z-positions of 0.0 and 10.0 for any values for the T-, X-, and Y-positions. Similarly, ad hoc queries also use wild-card values for missing values of the search hyper-cube.

4.2.5.1.4 Delete

The *Delete* command removes all data objects in the database that are consistent with the provided attributes. A data object is consistent when the object's attribute values "intersect" with the *Query* attribute values. The definition of intersection is different for the key and non-key attributes. The attributes that make up the index key are hyper-cubes and the definition for an intersection is an intersection of the respective hyper-cubes, i.e., an intersection occurs whenever the input hyper-cube and the stored hyper-cube share any of the hyper-space. The non-key attributes consist of character sequences and the definition of an intersection is when any part of the stored character sequence matches with the input sequence. This description of consistency is identical to the one given for the *Query* command in the previous section.

The first piece of data is the command code, which is '3' for the *Delete* operation. The rest of the command line is a list of attribute code and value pairs and is identical to the *Query* command given in the previous section. A diagram of the *Delete* command line is given below.

int	*int*	*float / char*	…	*int*	*float / char*
code	attribute code	attribute value		attribute code	attribute value

The description of the attribute code-value pairs and the use of wild-card values is identical to the *Query* command given in the previous section.

4.2.5.2 Algorithmic Specification

The database consists of various combinations of the three types of data objects. The algorithm requires the maintenance of an R-Tree index structure. The program shall respond to the command operations: *Insert*, *Query*, and *Delete*. This section has three

parts: the data object description, R-Tree index structure and description, and R-Tree variant discussion.

4.2.5.2.1 Data Object Description

Each entry in the database will be one of three types as discussed in section 4.2.5.1. A data object has a set list of attributes, which are the sum of the key and non-key attributes. The first eight attributes represent the index key and is specified as eight 32-bit IEEE floating-point numbers representing the T, X, Y, and Z-positions of both the "lower" and "upper" points of a hyper-cube, respectively. Finally, each object has a constant number of attributes or parts. A non-key data object attribute is an 8-bit NULL-terminated ASCII character sequence of maximum length 1024. The number of attributes assigned to each data object type is given in Table 4.2.5-2. The attributes for a given data object reference each other as a single-linked list and the data object holds a reference to the first attribute in the list which is defined as the head. Separate indices that would use the non-key attributes are not permitted for this benchmark. Thus, a *Query* operation which contains no key search information will search the entire database for consistent entries.

4.2.5.2.2 R-Tree

This benchmark requires the implementation of a simplified object-oriented database and an attendant R-Tree indexing structure. A general R-tree has the following properties:

1. All leaves are at the same level (height-balanced).
2. Every node contains between kM and M index entries unless it is the root. (M is the order of the tree).
3. For each entry in an intermediate node, the sub-tree rooted at the node contains a hyper-cube if and only if the hyper-cube is "covered" by the node, i.e., containment.
4. The root has at least two children, unless it is a leaf.

This benchmark requires the R-Tree structure be maintained during execution of the database implementation. However, the particular method used to maintain the R-Tree (search, tree compacting, etc.) is left to the user.

This section gives a brief description of the R-Tree algorithm and two variants. One variant allows for concurrency assurance without a full index list update (R-Link tree); the other seeks to minimize the overlap of the R-Tree with the offset of increasing the tree's height (R+-Tree). The user is encouraged to implement the standard R-Tree, variant described here, or other variant, as is most suitable for the hardware being tested. Descriptions given here are for illustrative purposes; they are not intended to dictate implementation strategy.

The R-Tree index provides a multi-dimensional data indexing scheme. It is a direct extension of the B-Tree in k dimensions (where k = 4 for this benchmark). The structure is a height-balanced containment tree, which consists of intermediate and leaf nodes. The data objects are stored as leaf-nodes (requiring four-dimensional position information). The intermediate nodes are built up by grouping all of the hyper-cubes at the lower level. The grouping hyper-cube of the intermediate node completely encloses all of the lower hyper-cubes and/or points. An example is the placement of rectangles in a Cartesian plane given in Figure 4.2.5-1.

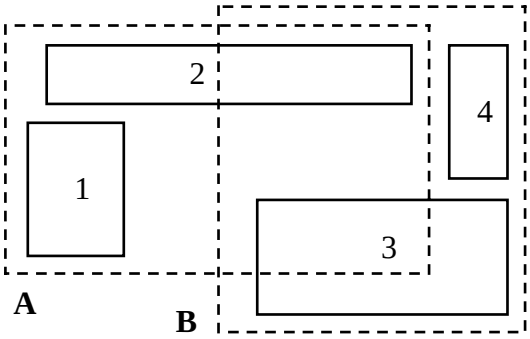


Figure 4.2.5-1: R-Tree 2D Example

This layout of rectangles would produce an index given in Figure 4.2.5-2.

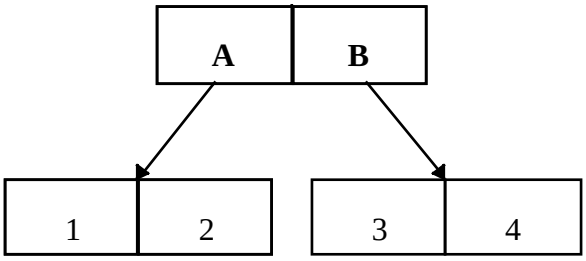


Figure 4.2.5-2: R-Tree Structure Example

The first command operation to be detailed is *Insert* . The *Insert* is the method used to place new data objects into the index and is the primary index management method, and thus the most complex. The command operation *Insert* for a generic R-Tree is given in Figure 4.2.5-3.

Method Insert(R, E)

Input: An R-Tree rooted at R and new data object with input hyper-cube E

Output: The new R-Tree after insertion of data object.

Method: Find where object should go and add to leaf nodes, splitting if necessary.

1. **Find leaf for insertion**. If R is not a leaf, recursively descend R and find L which is defined as the leaf node of R which gives the minimum penalty. The penalty of "change in area" proposed by Guttman is defined as the difference between the union of the hyper-cubes L and E and the area of L. The union of two hyper-cubes is itself a hyper-cube which minimally spans its components.
2. **Insert**. If L is not full, install E on L. Otherwise split L. Splitting L consists of separating L into two groups according to a similar "change in area" penalty. One group is placed on the new node, and the other is Inserted into the parent, splitting again if necessary.
3. **Adjust Keys**. Check the immediate parent of new node. If the key is already accurate or if there is no parent, stop. Otherwise, modify the parent to be the union of its children. Recursively ascend tree until root, R.

Figure 4.2.5-3 Insert

The "change in area" penalty is only one of several methods in choosing the leaf for insertion and for splitting. The user is referred to [Guttman], [Kornacker], and [Sellis], for a sampling of the different methods.

The second command operation detailed is the *Query* command, which is described in Figure 4.2.5-4. The *Query* command recursively descends all paths of the R-Tree which are consistent with the input search key returning all data objects which are consistent with the same search key.

Method Query (R, K, A)

Input: An R-Tree rooted at R, search key K, and non-key search values A

Output: The set of objects which are consistent with search key K.

Method: Recursively descend all paths of R which are consistent with K.

1. **Search**. Check each subtree of R to see if K is consistent. If so, search on subtree until leaf
2. **Check Key Attributes.** If current node is a leaf, check if K is consistent with data object. If so, add data object to solution set
3. **Check Non-Key Attributes.** Remove all entries in current solution set which are not consistent with non-key attributes of input values A.
4. **Return**. Return complete solution set

Figure 4.2.5-4: Query

The *Query* command detailed in Figure 4.2.5-4 is for key, non-key, and ad hoc queries. Note that a non-key query will check the entire database for all consistent objects. The method given in Figure 4.6 will work for a non-key query but will yield poor performance. Because of the default "wild-cards" for the non-specified search hyper-cube, the second step of the method will return the entire database as a list, which will then be searched by the third step. The creation of that list using the R-Tree index is not efficient and the benchmark implementors are encouraged to have auxiliary lists or parallel searches to improve the performance for non-key queries.

The final command operation detailed is the *Delete* operation, described in Figure 4.2.5-5. The purpose of the *Delete* command is to measure the performance of index management when entries are removed. The *Delete* command presented here uses the *Query* operation to determine the data objects that need to be removed, and so, the *Delete* performance is also a measure of the *Query* performance.

Method Delete (R, K, A)

Input: An R-Tree rooted at R, search key K, and non-key search values A

Output: The new R-Tree after deletion of all consistent data objects.

Method: Remove all data objects consistent with both K and A.

1. **Search** . Query index for all entries, L, consistent with K and A.
2. **Delete** . Remove all entries in L from R
   - 2.1. **Remove:** Remove entry in L from parent leaf, P.
   - 2.2. **Condense** . Recursively ascend tree, from P, adjusting the keys to minimize the penalty until the root, R.
3. **Clean-up:** If the root node has only one child, make child the new root.

Figure 4.2.5-5: Delete

4.2.5.2.3 R-Tree Variants

The user is constrained to implement the R-Tree structure as the indexing scheme for this benchmark application. However, the user is encouraged to select any implementation or variant of the R-Tree algorithm. Two variants, which may improve performance for a specific hardware architecture, are the R-link tree and the R+-Tree; these are discussed here.

The R-link tree variant uses a technique corresponding to the B-link tree to develop a scheme that does not require parent node locking for concurrent operations on the tree. Two differences from the base R-Tree are introduced for R-link trees. The first requires that all nodes within a level are right-linked together for a singly linked list. The second difference adds a logical sequence number (LSN) to each node which is unique within the tree/partition. The R-link algorithm uses the LSN to provide a mechanism for determining when an operation's understanding of a given node is obsolete, i.e., a node split has

occurred. If a split has occurred, the right-link is used to traverse the tree until a correct or expected LSN is found.

The R+-Tree eliminates overlap by reducing any overlapping hyper-cubes into sub-cubes and redistributing the tree. This provides a marked increase in search performance. The increase is offset by a more complicated index maintenance and by an increase of approximately 10% for the space required for the index.

4.2.5.3 Output

The output of the database will be the responses to each *Query* operation.

The response to a *Query* operation consists of a set of data objects that are consistent with the *Query*. Each data object in a response is represented as the list of its attributes in order defined by Table 4.2.5-3. The list of attributes shall be written to an 8-bit ASCII character file where each attribute is space delimited and with each list carriage return delimited. The format is very similar to the *Insert* operation format for the input data sets with the only difference being the omission of the command code. The set of data objects returned by a *Query* must be placed in the output file continuously, i.e., in adjacent lines, although the order of the set is not constrained.

4.2.5.4 Acceptance Test

A given data set will be considered successfully executed when the command operation query responses match with the corresponding data set query responses provided by the baseline results.

4.2.5.5 Metrics

The primary metric associated with the Data Management benchmark is total time for accurate completion of a given input data set. A series of secondary metrics are the individual times of the command operations: *Insert*, *Delete*, and *Query*. Best, worst, and average times should be reported for all operations for each data set.

The time for a non-response command operation to complete is defined as the difference between the time immediately before the command is placed in the database input queue and the time immediately before the next command is placed in the same input queue. This time difference is essentially the rate at which each line of the input data set is read. This definition is applied to the *Insert* and *Delete* command operations. The time for a *Query* command operation to complete is defined as the difference between the time immediately before the command is placed in the input queue to the time immediately after the response is placed in the output queue.

4.2.5.6 Baseline Source Code

Baseline source code is available at http://www.aaec.com/projectweb/dis.

4.2.5.7 Baseline Performance Figures

Baseline performance figures are available at http://www.aaec.com/projectweb/dis.

4.2.5.8 Test Data Sets

Test data sets are available at http://www.aaec.com/projectweb/dis.

4.2.5.9 References

[Guttman] Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOID*, pp. 47-57, June 1984.

[Kornacker] Kornacker, Banks, "High-Concurrency Locking in R-Trees," *Proceedings of 21st International Conference on Very Large Data Bases*, pp. 134-145, September 1995.

[Sellis] Sellis , Roussopoulos, and Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 13th International Conference on Very Large Data Bases*, pp. 507-518, Brighton, September 1987.

5 Contact Information

For questions about…	Contact…
• Data-Intensive Systems program	Dr. José Mu•oz DARPA / ITO 3701 North Fairfax Drive Arlington, VA 22203
• This Document • Benchmarks • Procedures • Input/Output Data • Baseline Performance • Reporting of Results • Multidimensional Fourier Transform Benchmark • Image Understanding Benchmark • Data Management Benchmark	Joseph F. Musmanno Atlantic Aerospace Electronics Corporation 470 Totten Pond Road Waltham, MA 02451 Telephone: 781-890-4200x3218 Fax: 781-890-0224 Email: joe@aaec.com
• Method of Moments Benchmark	Joseph W. Manke, Ph.D. The Boeing Company PO Box 3707 MC 7L-21 Seattle, WA 98124-2207 Telephone: 425-865-3163 Fax: 425-865-2966 Email: joseph.w.manke@boeing.com
• Ray-Tracing Benchmark	Jon W. Harris ERIM International, Inc. PO Box 134008 Ann Arbor, MI 48113-4008 Telephone: 734-994-1200x3313 Fax: 313-994-5124 Email: jharris@erim-int.com

6 References

DIS Program

[DARPA] DARPA/ITO website http://www.darpa.mil/ito.

[Mu□oz] Dr. Jos□ Mu□oz, presentation at Data-Intensive Systems Principal Investigators' meeting, 1 October, 1998, http://www.darpa.mil/ito/research/pdf_files/dis_approved.pdf.

Benchmarking

[Honeywell] Honeywell, Inc., *Benchmarking Tools and Assessment Environment for Configurable Computing: Benchmark Definition Methodology Document*, submitted to USA Intelligence Center and Fort Huachuca under contract number DABT63-96-C-0085, 19 February 1998.

[Weems] Weems, Riseman, and Hanson, The DARPA Image Understanding Benchmark for Parallel Computers, *Journal of Parallel and Distributed Computing*, 11, 24 January 1991.

[ASCII] *Information Systems - Coded Character Sets - 7-bit American National Standard Code for Information Interchange*, ITI (NCITS), ANSI x3.4-1986 (R1997).

[Float] *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std* 754-1985 (IEEE 754), published by the Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, 1986.

Method of Moments

[Rokhlin-1] V. Rokhlin, "Diagonal Forms of Translation Operators for the Helmholtz Equation in Three Dimensions", Research Report YALEU/DCS/RR-894, Dept. of Comp. Sci., Yale Univ., March, 1992.

[Rokhlin-2] R. Coifman, V. Rokhlin and S. Wandzura, "The Fast Multipole Method for the Wave Equation: A Pedestrian Prescription", IEEE Antennas and Propagation Magazine, 35, No. 3, June 1993, pp. 7-12.

[Dembart-1] B. Dembart and E. L. Yip, "A 3-d Fast Multipole Method for Electromagnetics with Multiple Levels", ISSTECH-97-004, The Boeing Company, December, 1994.

[Dembart-2] M. A. Epton. and B. Dembart, "Multipole Translation Theory for the 3-D Laplace and Helmholtz Equations", SIAM J. Sci. Comput. 16, No. 4, pp. 865-897, July, 1995.

[Dembart-3] M. A. Epton and B. Dembart, "Low Frequency Multipole Translation Theory for the Helmholtz Equation", SSGTECH-98-013, The Boeing Company, August, 1998.

[Dembart-4] M. A. Epton and B. Dembart, “Spherical Harmonic Analysis and Syntheses for the Fast Multipole Method”, SSGTECH-98-014, The Boeing Company, August, 1998.

[Saad] Yousef Saad, “Iterative Methods for Sparse Linear Systems”, PWS Publishing Company, Boston, MA, 1996.

Simulated SAR Ray Tracing

Ray Tracing References

[1] K. Bouatouch and T. Priol. Parallel space tracing: An experience on an iPSC hypercube. In N. Magnenat-Thalmann and D. Thalmann, editors, *New Trends in Computer Graphics (Proceedings of CG International ’88)*, pages 170–187, New York, 1988. Springer-Verlag.

[2] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti. Multiprocessor ray tracing. *Computer Graphics Forum*, pages 3–12, 1986.

[3] F. C. Crow, G. Demos, J. Hardy, J. McLaugglin, and K. Sims. 3d image synthesis on the connection machine. In *Proceedings Parallel Processing for Computer Vision and Display*, Leeds, 1988.

[4] M. A. Z. Dipp´ e and J. Swensen. An adaptive subdivision algorithm and parallel architecture for realistic image synthesis. *ACM Computer Graphics*, 18(3):149–158, jul 1984.

[5] S. A. Green and D. J. Paddon. Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics and Applications*, pages 12–27, nov 1989.

[6] H. Kobayashi, S. Nishimura, H. Kubota, T. Nakamura, and Y. Shigei. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer*, 4(4):197–209, 1988.

[7] A. J. F. Kok. *Ray Tracing and Radiosity Methods for Photorealistic Image Synthesis*. PhD thesis, Delft University of Technology, jan 1994.

[8] T. T. Y. Lin and M. Slater. Stochastic ray tracing using SIMD processor arrays. *The Visual Computer*, 7:187–199, 1991.

[9] D. J. Plunkett and M. J. Bailey. The vectorization of a ray-tracing algorithm for improved execution speed. *IEEE Computer Graphics and Applications*, 5(8):52–60, aug 1985.

[10] T. Priol and K. Bouatouch. Static load balancing for a parallel ray tracing on a MIMD hypercube. *The Visual Computer*, 5:109–119, 1989.

[11] E. Reinhard. Hybrid scheduling for parallel ray tracing. TWAIO final report, Delft University of Technology, jan 1996.

[12] I. D. Scherson and C. Caspary. A self-balanced parallel ray-tracing algorithm. In P. M. Dew, R. A. Earnshaw, and T. R. Heywood, editors, *Parallel Processing for Computer Vision and Display*, volume 4, pages 188–196, Wokingham, 1988. Addison-Wesley Publishing Company.

[13] L. S. Shen, E. Deprettere, and P. Dewilde. A new space partition technique to support a highly pipelined parallel architecture for the radiosity method. In *Ad-*

vances in Graphics Hardware V, proceedings Fifth Eurographics Workshop on Hardware. Springer-Verlag, 1990.

[14] E. R. Frederik, W. Jansen . Rendering Large Scenes Using Parallel Ray Tracing. *Parallel Computing*, pages 873-885, 1997

[15] T. Wilson, N. Doe. Acceleration Schemes for Ray Tracing. Report Number: CS-TR-92-22, Department of Computer Science, University of Central Florida, September 1992.

[16] R.L. Cook, T. Porter, L. Carpenter. Distributed Ray Tracing. *Computer Graphics* (Proceedings of SIGGRAPH 1984), 18(3), 137-145, July 1984.

[17] R.L. Cook. Stochastic sampling in computer graphics, ACM Transaction in Graphics 5(1), 51-72, January 1986.

[18] A. S. Glassner (Editor), *An Introduction to Ray Tracing*, Academic Press 1989.

[19] Ray Tracing Bibliography, http://www.cm.cf.ac.uk/Ray.Tracing/RT.Bibliography.html

Simulated SAR References

[1] D.J. Andersh, M. Hazlett, S.W. Lee, D.D. Reeves, D.P. Sullivan and Y. Chu, "Xpatch: A high fre-quency electromagnetic-scattering prediction code and environment for complex three-dimensional objects," *IEEE Antennas & Propagation. Magazine*, vol. 36, pp.65-69, 1994.

[2] J. Baldauf, S.W. Lee, L. Lin, S.K. Jeng, S.M. Scarborough, and C.L. Yu, "High frequency scattering from trihedral corner reflectors and other benchmark targets: SBR vs. experiment," *IEEE Transacrions on Antennas and Propagation*, vol. 39, pp. 1345-1351, 1991.

[3] R. Bhalla and H. Ling, *Image-domain ray tube integration formula for the shooting and bouncing ray technique*, University of Texas Report, NASA Grant NCC 3-273, July 1993.

[4] R. Bhalla and H. Ling, "A fast algorithm for signature prediction and image formation using the shooting and bouncing ray technique," to appear in *IEEE Transactions on Antennas and Propagation*, 1995.

[5] G. Franceschetti, M. Migliaccio, D. Riccio, and G. Schirinzi, "SARAS: A Synthetic Aperture Radar (SAR) Raw Signal Simulator," *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 30, No. 1, January 1992.

[6] G. Franceschetti, M. Migliaccio, and D. Riccio, "SAR Raw Signal Simulation of Actual Ground Sites in Terms of Sparse Input Data," *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 32, No. 6, November 1994.

[7] D.E Herrick and I.J. LaHaie, *SRIM Polarimetric Signature Modeling*, ERIM IR&D Final Report 675805-1-F, December 1988.

[8] D.E Herrick and B.J. Thelen, "Computer Simulation of Clutter in SAR Imagery," *Proceedings of the Progress in Electromagnetics Research Symposium*, Cambridge, MA, July 1991

[9] D.E Herrick, "Computer Simulation of Polarimetric Radar and Laser Imagery," in *Direct and Inverse Methods in Radar Polarimetry,* W.-M. Boerner *et al.* (eds), Klumer Academic Publishers, The Netherlands 1992.

[10] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Modeling of Foliage Effects in UHF SAR", *Proceedings qfthe Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[11] D.E Herrick, M.A. Ricoy, and W.D. Williams, "Synthesizing SAR Signatures of Ground Vehicles with Complex Scattering Mechanisms", *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[12] E.R. Keydel, D.E *Henick,* and W.D. Williams, "Interactive Countermeasures Design and Analysis Tool," *Proceedings of the Ground Target Modeling and Validation Conference,* Houghton, MI, August 1994.

[13] S.W. Lee and D.J. Andersh, *On Nussbaum Method for Exponential Series,* Electromagnetic Laboratory Technical Report ARTI-92-11, University of Illinois, Urbana, November, 1992.

[14] H. Ling, R.C. Chou, and S.W. Lee, "Shooting and Bouncing Rays: Calculating the RCS of an arbitrarily shaped cavity," *IEEE Transactions on Antennas and Propagation,* vol. 37, pp. 194-05, 1989.

[15] J.M. Nasr and D. Vidal-Madjar, "Image Simulation of Geometric Targets for Spaceborne Synthetic Aperture Radar", *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 6, November 1991.

[16] N.D. Taket, S.M. Howarth, and R.E. Burge, "A Model For the Imaging of Urban Areas by Synthetis Aperture Radar," *IEEE Transactions on Geoscience and Remote Sensing,* Vol. 29, No. 3, May 1991.

[17] M.R. Wohlers, S.Hsiao, J. Mendelsohn, and G. Gerdner, "Computer Simulation of Synthetic Aperture Radar Images of Three-Dimensional Objects," *IEEE Transactions on Aerospace and Electronic Systems,* Vol. AES-16, No. 3, May 1980.

Image Understanding

[Castleman] Castleman, K., *Digital Image Processing*, Prentice-Hall, 1979.

[Maragos] Maragos, P., “Tutorial on advances in morphological image processing and analysis,” *Optical Engineering*, vol. 26, no. 7, pp. 623-632, July 1987.

[Parker 94] Parker, J., *Practical Computer Vision Using C*, Wiley, 1994.

[Parker 97] Parker, J., *Algorithms For Image Processing And Computer Vision*, Wiley Computer Publishing, 1997.

[Unser] Unser, M., “Sum and Difference Histograms for Texture Classification,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, 1:118-125, 1986

[Weeks] Weeks, A., *Fundamentals of Electronic Image Processing*, SPIE/IEEE series on imaging science & engineering, 1996.

Fourier Transform

[Duhamel90] Duhamel and Vetterli, “Fast Fourier transforms: a Tutorial Review and State of the Art,” *Signal Processing*, vol. 19, pp.259-299, April 1990.

[Cooley] Cooley and Tukey, “An Algorithm for Machine Computation of Complex Fourier Series,,” *Math. Comp.*, vol. 19, pp.297-301, April 1965.

[Ganapa] Ganapathiraju, Hamaker, Picone and Skjellum, "Analysis and Characterization of Fast Fourier Transform Algorithms," MS State High Performance Computing Laboratory, Oct. 1997.

[Duhamel84] Duhamel and Hoolomann, “Split Radix FFT Algorithm,” *Electronic Letters*, vol. 20, pp.14-16, Jan 1984.

[Rader] Rader, “Discrete Fourier Transforms when the Number of Data Samples is Prime,” *Proc. of the IEEE*, vol. 56, pp.1107-1108, June 1968.

[Frigo] Frigo and Johnson, The FFTW web page, http://theory.lcs.mit.edu/~fftw

[Frigo99] Frigo, “A Fast Fourier Transform Compiler,” MIT Laboratory for Computer Science, Feb. 16, 1999.

[Saidi] Saidi, “Decimation-In-Time-Frequency FFT Algorithm,” *Proc. of International Conference on Acoustics, Speech, and Signal Processing*, vol. III, pp.453-456, Adelaide, Australia, April 1994.

[Johnson] Frigo and Johnson, The BenchFFT web page, http://theory.lcs.mit.edu/~benchfft

Data Management

[Guttman] Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” *Proc. ACM SIGMOID*, pp. 47-57, June 1984.

[Kornacker] Kornacker, Banks, “High-Concurrency Locking in R-Trees,” *Proceedings of 21st International Conference on Very Large Data Bases*, pp. 134-145, September 1995.

[Sellis] Sellis , Roussopoulos, and Faloutsos, “The R+-Tree: A Dynamic Index for Multi-Dimensional Objects,” *Proc. 13th International Conference on Very Large Data Bases*, pp. 507-518, Brighton, September 1987