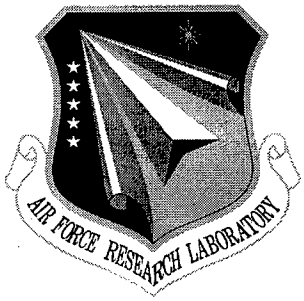
AFRL-IF-RS-TR-2000-105
Final Technical Report
July 2000



SPACE-TIME ADAPTIVE PROCESSING ON COMMERCIAL HIGH-PERFORMANCE COMPUTERS

Cornell University

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. C253

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-105 has been reviewed and is approved for publication.

APPROVED:

RALPH KOHLER
Project Engineer

FOR THE DIRECTOR:

NORTHRUP FOWLER, Technical Advisor
Information Technology Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTC, 26 Electronic Pky, Rome, NY 13441-4514. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

SPACE-TIME ADAPTIVE PROCESSING ON COMMERCIAL HIGH-PERFORMANCE COMPUTERS

Adam W. Bojanczyk

Contractor: Cornell University
Contract Number: F30602-95-1-0016
Effective Date of Contract: 12 April 1995
Contract Expiration Date: 30 June 1999
Short Title of Work: Space-Time Adaptive Processing on Commercial High-Performance Computers
Period of Work Covered: Apr 95 – Jun 99

Principal Investigator: Andrew W. Bojanczyk
Phone: (607) 255-4296
AFRL Project Engineer: Ralph Kohler
Phone: (315) 330-2016

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Ralph Kohler, AFRL/IFTC, 26 Electronic Pky, Rome, NY.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 2000	Final Apr 95 - Jun 99

4. TITLE AND SUBTITLE
SPACE-TIME ADAPTIVE PROCESSING ON COMMERCIAL HIGH-PERFORMANCE COMPUTERS

5. FUNDING NUMBERS
G -F30602-95-1-0016
PE - 62301E
PR - C253
TA - 01
WU - P1

6. AUTHOR(S)
Adam W. Bojanczyk

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Cornell University
School of Electrical Engineering
335 Frank H.T. Rhodes Hall
Ithaca, NY 14853-3801

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/IFTC
26 Electronic Parkway
Rome NY 13441-4514

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2000-105

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer: Ralph Kohler/IFTC/(315)330-2016

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release, Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This project consisted of building a portable parallel library for the space-time adaptive processing (STAP) problem. Portability was achieved by using standard like BLAS, LAPACK, SCALAPACK and MPI. The library simplifies implementation of STAP applications on different high-performance parallel computers, allowing rapid prototyping of parallel STAP systems. The library includes common to STAP communication and computation procedure. All library routines take as input 3D data cubes, and produce as outputs also 3D cubes. Two user manuals, one describing the data distributions library, and the other describing the STAP algorithms constraction were developed for this project.

14. SUBJECT TERMS
Parallel C Algorithms, ALPS library, SplitStaggeredCube function, Pri-Staggered SubCubes, FullUpdateStap, Data Cube, Block Cyclic Data Distribution

15. NUMBER OF PAGES
92

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

I. Administrative Information

Agreement No:	F30602-95-1-0016
Darpa order number	C253
Total Amount	$264,347.00
Start Date:	April 12, 1995
End Date:	April 11, 1998
No cost extension:	December 31, 1999
Principal Investigator	Adam W. Bojanczyk
Level of PI participation:	
Billed:	10%
Unbilled:	10%

Address

Cornell University
School of Electrical Engineering
335 Frank H.T. Rhodes Hall
Ithaca, NY 14853-3801
(607) 255 4296
(607) 255 9072
adamb@ee.cornell.edu
http://www.ee.cornell.edu/~adamb/STAP.html

II. Technical Report

(a) Description of the project.

In this project a portable parallel library for the space-time adaptive processing (STAP) problem was built. Portability was achieved by using standard like BLAS, LAPACK, SCALAPACK and MPI. The library simplifies implementation of STAP applications on different high-performance parallel computers, allowing rapid prototyping of parallel STAP systems.

The library includes common to STAP communication and computation procedure. These procedures form basic building blocks from which a parallel STAP application can be built. The building blocks are divide into data redistribution and computational blocks. They are implemented with standard BLAS, ScaLAPACK, LAPACK, and MPI routines. Each building block has several implementations corresponding to different parallel data distributions and machine-specific parameters.

All library routines take as input 3D data cubes, and produce as outputs also 3D cubes. STAP systems are built as sequences of calls to the redistribution and computation routines, all operating on 3D data cubes.

Two manuals, one describing the data distributions library, and the other describing the STAP algorithms constraction, are included with this final report. The software packaged as a tar file can be downloaded from our web site.

(b) Publications.

Towards a Portable Parallel Library for Space-Time Adaptive Methods, J. Lebak, R. Durie and A.W. Bojanczyk, Cornell Theory Center Technical Report, Cornell University, June 1996. (Available in a postscript format from `http://www.tc.cornell.edu/Research/Tech.Reports/`).

Portable Parallel Subroutines for Space-Time Adaptive Processing, PhD Dissertation, J.M. Lebak, January 1997.

Automated Modeling of Parallel Algorithms for Performance Optimization and Prediction, R. Durie and A. Bojanczyk, in the Proccedings of the Eighth SIAM Conference for Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997, (Also available in a postscript format from `http://www.ee.cornell.edu/~adamb/STAP/STAP.html`).

Multi-Instance Parallel Libraries for Three Dimensional Data Redistribution, MS thesis, W. Kostis, August 1998.

Some Improvements to Parallel Three Dimensional Data Redistribution Library, MEng thesis, R. Weitkunat, January 1999.

Design and Performance Evaluation of a Portable Parallel Library for Space-Time Adaptive Methods, J. Lebak and A. Bojanczyk, IEEE Transactions on Parallel and Distributed Systems, March 2000, to appear.

(c) Professional personnel.

In addition to partial support for the Principal Investigator, the budget included full support for one graduate student. Over the course of the project, the following students were partially supported by this grant:

James Lebak, PhD in January 1997
Bob Durie, PhD in progress
Will Kostis, MSc in August 1998
Richard Weuikenat, MEng in January 1999

(d) Interactions.

Meetings:

(1) PI meeting, Florida, March 1995, organizer R. Parker of ARPA, presentation by A. Bojanczyk, *"Space-Time Adaptive Processing on High-Performance".*

(2) Kick-off meeting, Ithaca, June 1995, M. Linderman and V. Vannicola of Rome Lab.

(3) General ARPA CSTO PI meeting, Florida, July 1995, organizer H. Frank of ARPA.

(4) PI meeting, Boston, November 1995, organizer V. Vannicola of Rome Lab, presentation by A.W. Bojanczyk *"STAP on High Performance Computers, Progress Report".*

(5) PI meeting, Atlanta, January 1996, organizer R. Parker of ARPA, presentation by graduate student G. Adams, *"Tools for building parallel application-specific libraries".*

(6) Six months review, Ithaca, January 1996, M. Linderman and V. Vannicola of Rome Lab.

(7) ASAP workshop, Boston, March 1996, presentation by graduate student R. Durie *" STAP on HPCs, Benchmarking Tools".*

(8) PI meeting, San Diego, June 1996, organizer J. Munoz of ARPA, presentation by graduate student

R. Durie, *"Modeling Parallel Libraries".*

(9) Six months review, Ithaca, September 1996, M. Linderman and V. Vannicola of Rome Lab.

(10) General ARPA CSTO PI meeting, Dallas, Texas, October 7-8, 1996, organizer H. Frank of DARPA.

(11) 8th SIAM Conference for Parallel Processing for Scientific Computing, Minneapolis, MN, March 14-17, 1997. Presentation on "Automated Modeling of Parallel Algorithms for Performance Optimization and Prediction".

(12) Fifth Annual Workshop on Adaptive Sensor Array Processing (ASAP '97), Lexington, MA, March 13, 1997, presentation on "Automated Application Synthesis for High-Performance Sensor Array Processing".

(13) DARPA Embeddable Systems PI Meeting, Santa Fe, NM, March 19, 1997, presentation by graduate student W. Kostis on *"Parallel Libraries for Space-Time Adaptive Processing ".*

(14) DARPA Embeddable Systems PI Meeting, Ft. Lauderdale, FL, March 25-27, 1998. The project progress report, "Space-Time Adaptive Processing on High-Performance Computers".

Consultative and advisory functions:

We helped to install several pieces of software on the Rome Lab Paragon system (tcsh, scalapack, lapack).

(e) New discoveries and inventions.

A benchmarking harness for the building block suite was developed. The benchmarking tools measure the performance of computational and communication subroutines on a variety of parallel machine configurations.

Tools for automatic performance modeling of parallel routines were developed. The tools facilitate complete performance characterization of individual library modules and entire STAP implementations.

Multi-instance libraries were created. A multi-instance library is constructed from multiple implementations of functionally identical routines. These routines can operate on different data distributions and utilize different algorithms however. When a routine is called from a multi-instance library, optimization tools are free to choose any implementation. Multi-instance libraries aid in performance optimization of STAP methods on parallel architectures.

(f) Patent disclosures.

There were none.

(g) Technology Transition.

The full versions of the redistribution library is available from our web site.

(h) Comparison of accomplishments with goals.

SCHEDULED WORK	ACTUAL ACCOMPLISHMENTS
YEAR 1	
• Examine various STAP methods and identify the complete set of major computational components needed. Assess available portable computation libraries.	• Pre- and Post-Doppler Element Space STAP methods were implemented. Major computational modules were identified. LAPACK and SCALAPACK computation libraries were selected.
• Examine possible communication needs. Assess available portable communication needs.	• Communication primitives were benchmarked. In addition to native communication libraries, a portable MPI communication library has been selected.
• Develop benchmarking suite.	• FFT, QR, triangular solve, I/O and "corner turn" problems were selected as benchmarking modules.
• Develop efficient methods for each computational module. Implement each module for various data distributions.	• Several computational modules were implemented with standard BLAS, ScaLAPACK, LAPACK, and MPI routines. Each building block has several implementations corresponding to different parallel data distributions and machine-specific parameters.
• Experiment with and analyze above implementations.	• The performance of each building block was experimentally determined. The building blocks were used to implement several variations of the higher-order post -Doppler STAP computation on the Intel Paragon and IBM SP2.
• Apply previous work in recursive least squares as one solution to the STAP problem.	• A new implementation of the "sliding hole" strategy was proposed.

Scheduled Work	Actual accomplishments
Year 2	
• Complete initial development, experimentation, and analysis of all computational modules.	• MATLAB routines were written for library modules to verify numerical correctness of parallel codes. • Data redistribution routines were added to the library.
• Port implementations to target architectures.	• All modules were run on the Intel Paragon and the IBM SP2.
• Experiment and analyze implementations for all target architectures.	• Analytic models based on benchmarked machine parameters were developed.
• Optimize implementations where possible.	• Several implementations of basic library modules were developed. Multiple implementations were collected in *Multi-instance libraries.*
• Experiment with new implementations.	• A PRI-Staggered method was implemented on both Paragon and SP2. • A hybrid fine-coarse grain implementation of the HOPD method was developed. The hybrid method can use more processors than the number of PRIs in the data cube. It was often faster than methods that exploit only coarse grain parallelism. • A parallel version of MITRE STAP benchmarks was built from library modules.

Scheduled Work	Actual accomplishments
Year 3	
• Consider overall computational flow for each STAP method. • Build implementations for each STAP method considered using modular building blocks. • Experiment and analyze STAP implementations. • Optimize STAP implementations where possible.	• Two ways of partitioning the STAP input datacube were implemented using library modules. Two different algorithms were built from library modules which together with the two different partitioning scheme gave four distinct STAP systems. • The code was verified on the Intel Paragon and the IBM SP2 parallel computers. • User manuals were written and the code was made available for downloads from the PI's web site.

STAP Algorithm Construction

A.W. Bojanczyk and R.H. Weitkunat

Chapter 1

Introduction

This document illustrates how one can construct parallel STAP applications in the C language utilizing the MPI-based ALPS redistribution library and LAPACK, a third-party linear algebraic math library. Two examples of STAP implementations are presented, as well as information on installing and configuring the libraries for inclusion in parallel C algorithms.

Chapter 2 described the operation of the STAP algorithms including the organization and distribution of the input data. Chapter 3 explains how to obtain the ALPS and LAPACK libraries, and provides instruction for executing the sample STAP programs described in chapter 2.

Chapter 2

Space Time Adaptive Processing (STAP) algorithms

Two sample STAP implementations are presented in this section to demonstrate the use of the ALPS library in constructing parallel algorithms. It is assumed that the reader is familiar with STAP processing and has familiarity with the ALPS library as described in the ALPS manual [1]. The performance of some of the ALPS procedures were measured and analyzed in [2].

Each implementation is organized into two components: first, the pre-STAP preparation of the datacube and the set of steering vectors; and second, the actual STAP processing. See Figure 2.1 for a general depiction of implementation.

The pre-STAP steps include distribution, duplication, and optionally performing doppler processing (a one dimensional FFT along all `Pri`'s for each value of `Range` and `Channel`). This creates a number of independent subcubes that each have the same length of `Range` and `Channel` dimensions, with shortened `Pri` dimensions, and are distributed across the set of parallel processors.

The actual STAP processing is then performed on each subcube, independently of the others. Each STAP operation produces output data which are joined together to comprise a distributed datacube for output.

Two methods of data distribution are combined with two STAP algorithms to perform four different operations, both with the option of applying the FFT. Each program processes two sets of data, the radar data and the set of steering vectors.

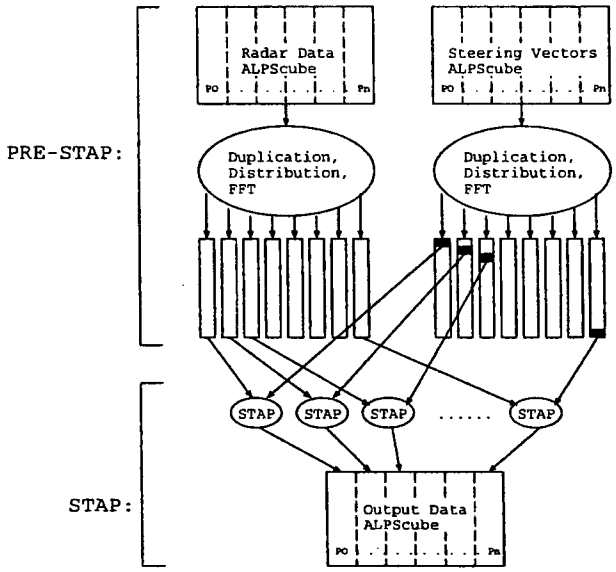


Figure 2.1: General depiction of STAP algorithm implementation

2.1 Pre-STAP processing

Two schemes of data distribution in our examples are shown in Figures 2.2 and 2.4. Each operation is accomplished by an appropriate ALPS library function and is labeled with the names of the corresponding procedural call. The input data must be initially stored as files in the standard ALPS pdc format; see [1] for the description of the pdc format. Some examples of creating datacubes in pdc format are discussed in section 3.3.1.

2.1.1 Pri-Overlap SubCubes

The first scheme in Figure 2.2, dubbed the 'pri-overlap' technique, divides both the radar data and steering vectors into smaller *subcubes* with overlapping regions in the `Pri` dimension.

The number of subcubes that are created, along with the length of each subcube in the `Pri` dimension, are controlled by two parameters: *offset* and *overlap*. These parameters are directly supplied to the ALPS **SplitStaggeredCube** function which results in the creation of the overlapping subcubes. See Figure 2.3 - the user is also refered to the ALPS library manual

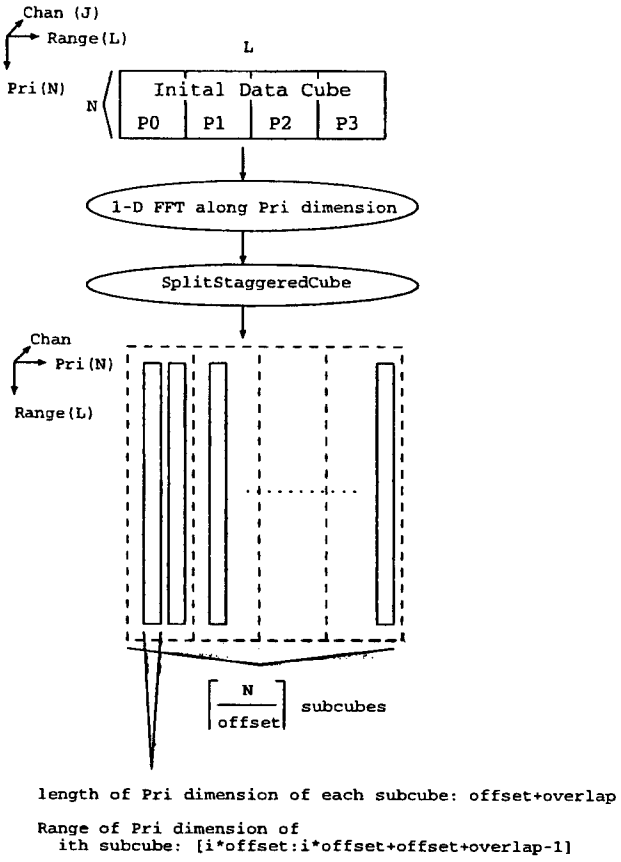


Figure 2.2: Pri-Overlap distribution

[1] for further details on the usage of **SplitStaggeredCube**.

The number of new subcubes will be equal to the number of Pri's in the original datacube divided by the offset, or $\lceil \frac{\texttt{Pri}}{offset} \rceil$. Each process will contain $\lceil \frac{\texttt{Pri}}{offset*P} \rceil$, except possibly for the last processor which will contain $\lceil \frac{\texttt{Pri}}{offset} \rceil \bmod P$ subcubes, where P are the number of processors.

The starting Pri index of each subcube will be a multiple of the *offset* parameter: namely, the ith subcube's Pri dimension will start at index $(i * offset)$, and end at $(i * offset + (offset + overlap) - 1)$.

Doppler processing is optionally performed before the **SplitStaggered-Cube** operation, as indicated in Figure 2.2.

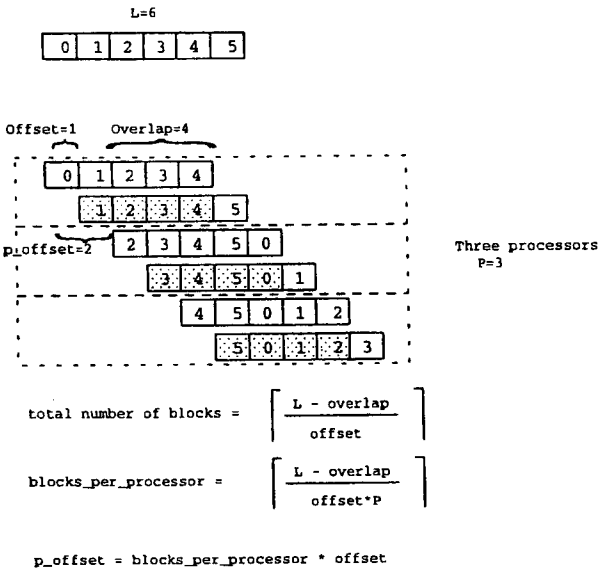


Figure 2.3: Example of SplitStaggeredCube's output in overlapping dimension

2.1.2 Pri-Staggered SubCubes

The second scheme, dubbed the 'pri-staggered' technique, reorganizes the data in a different fashion for the sake of performing the Doppler processing on overlapping sets of data and then recombining the data as indicated in Figure 2.4.

The number of cubes, k, produced as a result of the **SplitStaggered** operation, is a parameter set by the user. This parameter also determines the length of the `Pri` dimension for each resulting subcube after the **ReCube** operation is performed. The user is also refered to the ALPS library manual [1] for further details on the usage of **ReCube**. See Figure 2.5 for an illustration of the operation performed by these two operations in the staggered algorithm.

If no doppler processing is performed, then the result of this distribution is exactly the same as for the 'overlap' technique with parameters *offset* = 1 and *overlap* = k - 1.

2.1.3 Dividing up the Steering Vectors

After the above repartitioning of data has been accomplished for both the radar data and steering vectors, each subcube is ready to be processed by the STAP algorithm.

Each cube containing radar data is taken along with its corresponding steering vector cube, as shown in Figure 2.6. However, only a subset of the steering vectors are needed from each steering vector subcube. In general, if there are N subcubes and p steering vectors in total, , then each subcube of steering vectors is divided up into N subsets of $\lceil \frac{p}{N} \rceil$ vectors, and only the ith subset will be used from the ith subcube. The remainders of the subcubes are ignored.

As currently implemented, if there are fewer number of steering vectors than subcubes ($p < N$), only the first p subcubes will be processed. Since the subcubes are distributed consecutively, only the first $\lceil \frac{p}{P} \rceil$ processors will have work to do (where P is number of processors).

2.2 The Stap Algorithms

STAP processing relates to the minimization problem

$$\min_{d^H w=1} ||Xw||$$

where X and d represent the radar data and the steering vector, respectively.

In STAP it is of interest to evaluate influence of each individual row of X in the minimization. This importance can be assessed by measuring the magnitudes of corresponding residual elements.

The relevant residuals are defined as follows. Let X_i denote an $(L-1)\times N$ matrix composed of all rows of X with the exception of the row x_i^H,

$$X = P_i \begin{pmatrix} X_i \\ x_i^H \end{pmatrix} \tag{2.1}$$

where P_i denotes an appropriately chosen row permutation matrix, and let $w^{(i)}$ be the minimum norm solution to

$$\begin{array}{l} \min \\ d^H w = 1 \\ w \in \mathcal{C}^N \end{array} \quad ||X_i w||_2 \tag{2.2}$$

Then

$$r^{(i)} = X_i w^{(i)} \text{ and } \hat{r}_i = x_i^H w^{(i)} \tag{2.3}$$

denote the residual vector in (2.2) and a *predicted* residual element, respectively. The quantity

$$\hat{\hat{r}}_i = \frac{\hat{r}_i}{||r^{(i)}||_2} = \frac{|x_i^H w^{(i)}|}{|(w^{(i)})^H X_i^H X_i w^{(i)}|^{\frac{1}{2}}} \tag{2.4}$$

provides a measure of importance of the observation x_i^H in the set X. We will refer to $\hat{\hat{r}}_i$ as a *scaled residual element.*

Scaled residuals can be computed in various ways. In the following section we present two STAP algorithms: FullUpdateStap and PredictRes.

Both algorithms work in serial fashion: that is, each processor does its own computations with the data residing locally. These algorithms were written with the aid of BLAS and LAPACK linear algebra functions. Each takes as input a local ALPScube of radar data, and a local cube of steering vectors; these cubes are the subcubes produced by the redistribution described above.

2.2.1 FullUpdateStap

The FullUpdateStap solves $\min_{d^H w=1} ||Xw||$ by computing the weight vector w according to the formula

$$w = \frac{(X^H X)^{-1} d}{d^H (X^H X)^{-1} d}$$

If the QR factorization of X is $X = QR$, then $X^H X = R^H R$. Thus

$$w = \frac{(R^H R)^{-1} d}{d^H (R^H R)^{-1} d}$$

In FullUpdateStap the R factor is downdated after removing a row from the matrix X. Let R_i be the triangular factor of X_i where X_i is X with the ith row removed. From the Sherman Morrison formula we have

$$(R_i^H R_i)^{-1} = (R^H R)^{-1} + \frac{(R^H R)^{-1} x_i x_i^H (R^H R)^{-1}}{1 - x_i^H (R^H R)^{-1} x_i}$$

Thus the formula for the optimal $w^{(i)}$ becomes

$$w^{(i)} = \frac{(R_i^H R_i)^{-1} d}{d^H (R_i^H R_i)^{-1} d} = \frac{R^{-1}\hat{d} + \frac{R^{-1}\hat{x}_i \hat{x}_i^H \hat{d}}{1 - \hat{x}_i^H \hat{x}_i}}{\hat{d}^H \hat{d} + \frac{\hat{d}^H \hat{x}_i \hat{x}_i^H \hat{d}}{1 - \hat{x}_i^H \hat{x}_i}}$$

where $\hat{x}_i = R^{-H} x_i$, and $\hat{d} = R^{-H} d$.

Once $w^{(i)}$ are known , the vector $t = (\hat{\hat{r}}_i)$ of scaled residual elements 2.4 can be computed.

This process is repeated for each different direction vector d_k producing a matrix T of residual vectors, $T = [t_1, t_2, ...t_p]$.

The basic steps for the FullUpdateStap algorithm implement the algebraic formula above, and are described below and illustrated in Figure 2.7. The main computations are vector and matrix operations and are realized by BLAS and LAPACK functions.

Algorithm 2.1: FullUpdateStap

Input: Matrix X containing the radar data, and matrix D containing the set of steering vectors.
Intermediate quantities: matrix W containing weight vectors W_i
Output: matrix T containing residual vectors

LAPACK functions expect matrices to be in column-wise order.
The required orientation must have `Channels` in
fastest-varying order in memory, and the `Range` in slowest-varying order.

I. Receive datacubes X and D with an orientation of `Range`, `Pri`, `Chan`.
Map datacube to 2-d $n \times m$ matrix X.
Number of rows is $n =$ `Pri` $\cdot$ `Chan`,
and number of columns is $m =$ `Range`.

Map steering cube to 2-d $n \times m$ matrix D
Number of rows is $n =$ `Pri` $\cdot$ `Chan`, and number of columns
p is equal to the number of steering vectors.

II. Create hermitian of X: X^H. Required for obtaining QR factorization.
Simple `for` loop.
III. Do QR factorization of X^H: $X^H = QR$
LAPACK function call:

```
zgeqrf(m, n, r, m, tau, work, lwork, info);
```

IV. Copy D and X matrices into buffer A: $A \leftarrow [D\ X]$
Simple `memcpy`.

V. Solve for $\hat{A}$: $R^H \hat{A} = A$, where $\hat{A} = [\hat{D}\ \hat{X}]$
LAPACK function call:

```
ztrtrs("U", "C", "N", orderR, nrhs, r, m, a, n, info);
```

VI. Compute vectors N^D and N^X of squared norms of columns of the matrices $\hat{D}$ and $\hat{X}$:
$N_j^D = \sum_{i=0}^{n} |\hat{D}_{ij}|^2$,
$N_j^X = \sum_{i=0}^{n} |\hat{X}_{ij}|^2$

VII. Compute product $B \leftarrow \hat{X}^H\ \hat{D}$
LAPACK function call:

```
zgemm("C", "N", m, p, n, alpha, dx+sizeD, n, dx, n, beta, Xhd, m);
```

VIII. Solve for $\hat{\hat{A}}$: $R\hat{\hat{A}} = \hat{A}$, where $\hat{\hat{A}} = [\hat{\hat{D}}\ \hat{\hat{X}}]$.
LAPACK function call:

```
ztrtrs("U", "N", "N", orderR, nrhs, r, m, dx, n, info);
```

IX. For each direction vector $\hat{\hat{D}}_j$ (j = 1:p):
a) Compute columns W_i of matrix W:

```
for i = 1 to m,
```

$W_i = \frac{((1-N_i^X)*\hat{\hat{D}}_j + B_{i,j}*\hat{\hat{X}}_i)}{((1-N_i^X)*N_j^D + B_{i,j}^H B_{i,j})}$

where $\hat{\hat{D}}_j$ is jth column of matrix $\hat{\hat{D}}$.
$\hat{\hat{X}}_i$ is ith column of matrix $\hat{\hat{X}}$.
N_i^D is ith element of vector N^D.
N_i^X is ith element of vector N^X.
$B_{i,j}$ is (i,j) element of matrix $\hat{X}^H \hat{d}$.

b) Compute product $C \leftarrow X^H W$
LAPACK function call:

```
zgemm("C", "N", m, nrhs, n, alpha, x, n, w, n, beta, xhw, m);
```

c). Compute elements of matrix $T = (T_{i,j})$:
$C_{i,j}$ is (i,j)th element of matrix $X^H W$.
N_i = square norm of ith column of $X^H W$ excluding (i,i)th element,
or || (ith column of $X^H W$ excluding (i,i)th element) $||^2$.

for i = 1 to m,
$T_{i,j} = \frac{|C_{i,i}|^2}{N_i}$

X. Return matrix T as datacube

2.2.2 Predicted Residuals

The PredictedRes algorithm determines the residuals in a different fashion than FullUpdateStap. The first step in the PredictedRes method is to eliminate the constraint from the minimization $\min_{d^H w=1} ||Xw||$. This is achieved by mapping the steering vector d onto the direction e_n by a unitary transformation H. The data matrix X must be transformed in the same manner, resulting in the transformed data $X_H = XH$. The vector $t = (\hat{\hat{r}}_i)$ of predicted residuals can now be computed from the orthogonal factor $Q = (q_{i,j})$ of the QR decomposition of X_H according to the following formula:

$$\hat{\hat{r}}_i = \frac{q_{i,n}}{q_{i,n+1}} \cdot \frac{1}{\sqrt{q_{i,n}^2 + q_{i,n+1}^2}}$$

The PredictedRes algorithm is summarized in the psuedocode below.

Predicted Residuals

- determin a reflection H (or a sequence of rotations G) such that $d^H H = (0, .., 0, 1)$
- transform to unconstrained, $X_H \longleftarrow XH$
- get QR, $X_H = QR$
- calculate $d = \text{diag}(QQ^H)$, the diagonal of QQ^H
- calculate $q_{i,n+1} = \sqrt{1 - d_i}$, $i = 1, 2, ..., m$
- calculate the predicted scaled residual elements $\hat{\hat{r}}_i = \frac{q_{i,n}}{q_{i,n+1}} \cdot \frac{1}{\sqrt{q_{i,n}^2 + q_{i,n+1}^2}}$

The basic steps in the PredictRes algorithm are vector matrix operations and can be implemented with BLAS and LAPACK function calls, as illustrated below.

Algorithm 2.2: PredictRes

Input: Matrix X containing the radar data, and matrix D containing the set of steering vectors.
Output: matrix T containing residual vectors

LAPACK functions expect matrices to be in column-wise order.
The required orientation must have `Channels` in
fastest-varying order in memory, and the `Range` in slowest-varying order.

I. Receive datacubes X and D with an orientation of `Range`, `Pri`, `Chan`.
Map datacube to 2-d n-by-m matrix X.
Number of rows is $n =$ `Pri` $\cdot$ `Chan`, and number of columns
is $m =$ `Range`.

Map steering cube to 2-d n-by-m matrix D
Number of rows is $n =$ `Pri` $\cdot$ `Chan`, and number of columns
is $m =$ number of vectors.

II. Compute Householder vectors $H = (h_1, h_2, ...h_p)$:
for each direction vector d in matrix D (jth column where `j=1:p`)
$h \leftarrow \frac{d}{\delta_n(1+\frac{||d||}{|\delta_n|})}$
and $\eta_n \leftarrow 1$.
where δ_n is the last element in vector d.
and where η_n is the last element in vector h.

III. For each Householder vector h in matrix H:
a) Create hermitian of X: X^H. Required for obtaining QR factorization.

b) Compute $X_h \leftarrow X^H + \frac{2}{||h||}X^H hh^H$. This is done in two steps:
i) $Z \leftarrow \frac{2}{||h||}X^H h$, using LAPACK function:

```
zgemv("N", numrows, numcols, beta, data, numrows, hvec, incx, zero, buf,
```

ii) $X_h \leftarrow X^H + Zh^H$, using LAPACK function:

```
zgerc(numrows, numcols, one, buf, incx, hvec, incx, data, numrows);
```

c) Do QR factorization of X_h: $X_h = QR$
where Q has dimensions (m,n).

LAPACK function calls (Obtaining Q requires two steps):

QR factorization: `zgeqrf( m, n, q , m, tau, work, lwork, info );`

Obtain Q: `zungqr( m, n, n, q, m, tau, work, lwork, info);`

d) Compute Predicted Res of Q:

where $Q_{i,j}$ is ith row, jth column element of matrix Q,
N_i is square norm of ith row of Q ($N_i = \sum_{j=1}^{n} |Q_{i,j}|^2$)
$R_{i,j}$ is ith row, jth column element of output matrix R.

for each row i in Q: `for i = 1:m,`

$$R_{i,j} = \sqrt{\frac{|Q_{i,n}|^2}{(1-N_i)(1-N_i+|Q_{i,n}|^2)}}$$

IV. Return R as datacube.

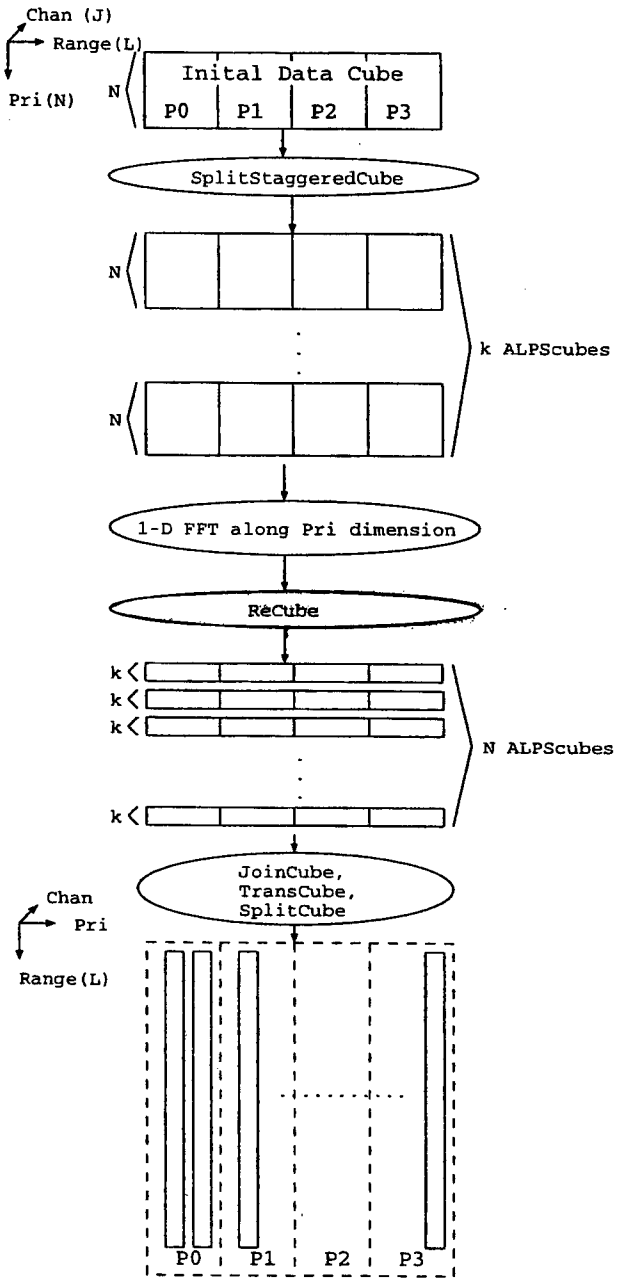


Figure 2.4: Pri-Staggered distribution

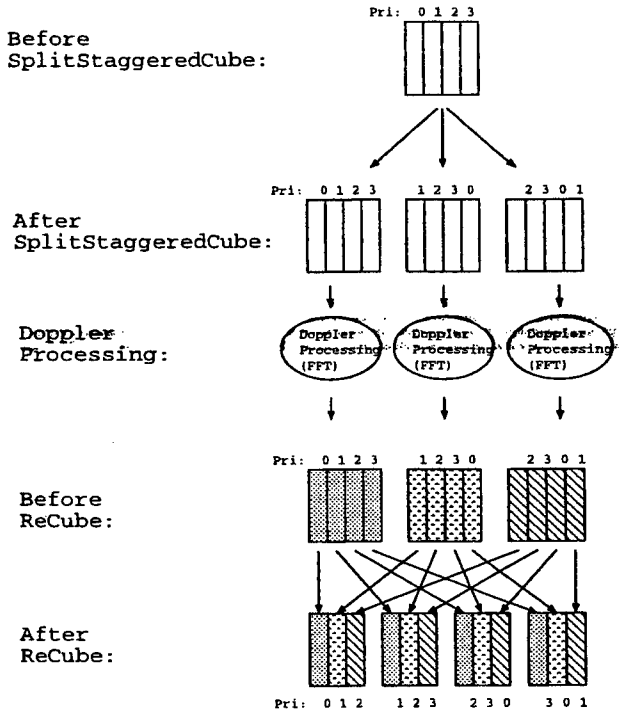


Figure 2.5: Example of Staggered Algorithm's operation

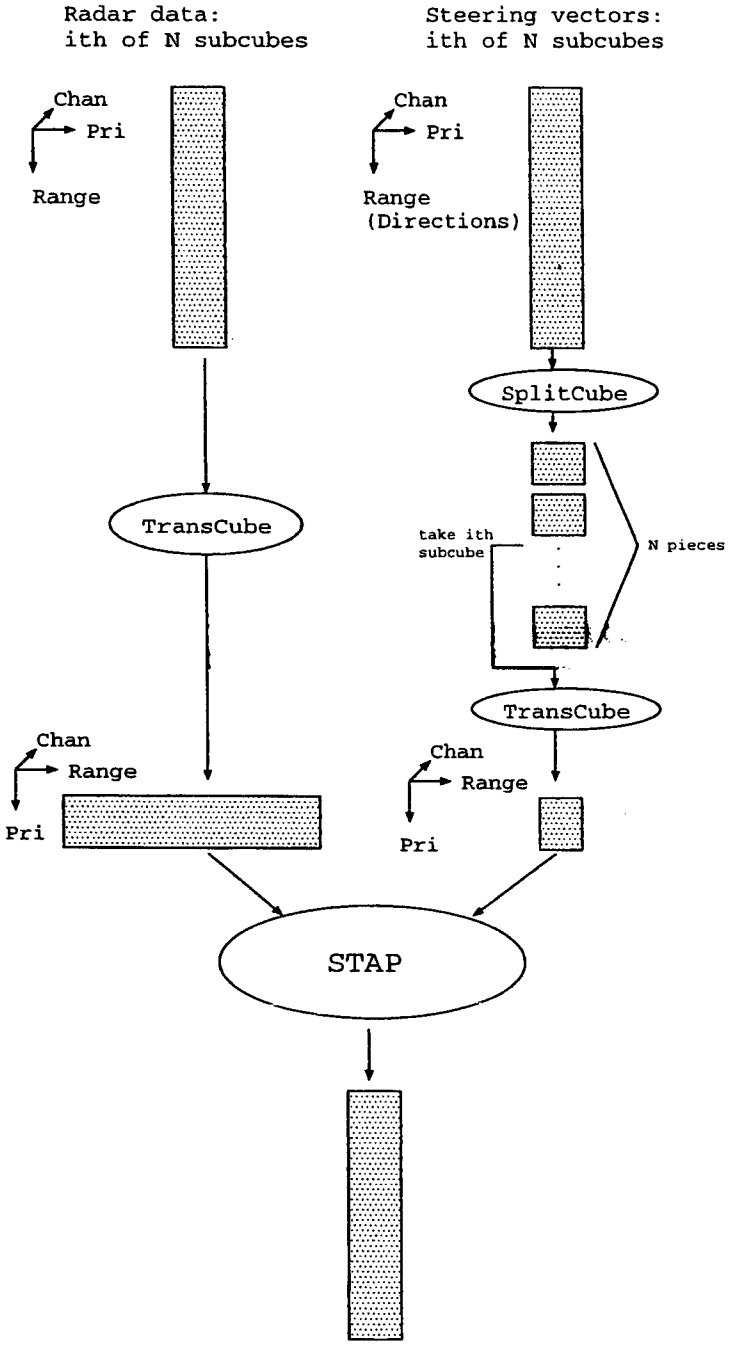


Figure 2.6: Input for the single-processor STAP operation

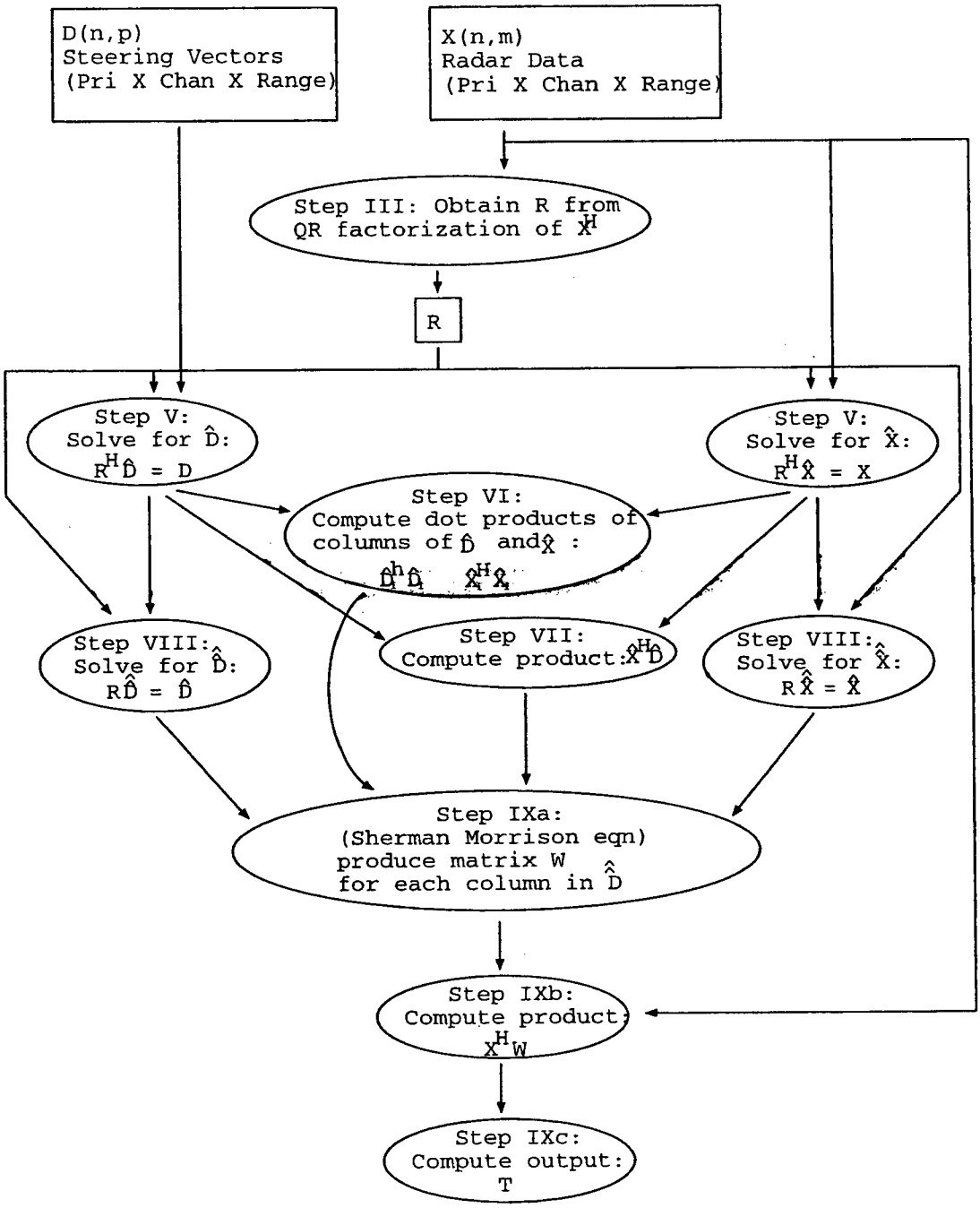


Figure 2.7: Illustration of FullUpdateStap algorithm

Chapter 3

Software Installation, Configuration, and Demonstration

3.1 Configuration

The LAPACK library of linear algebraic functions is public domain. The users guide is located on the web at:

`http://www.netlib.org/lapack/lug/lapack_lug.html`

The ALPS library is available at:

`http://www.ee.cornell.edu/~adamb/STAP/software/ALPScomm.tar.gz`

in compressed tar format. Please refer to the ALPS manual [1] for information on installing the ALPS library.

3.2 Compiling your own programs with the ALPS library

Users may want to modify existing example STAP programs, or create their own STAP programs from LAPACK and ALPS modules.

3.2.1 Modify existing programs

Example programs such as `overlap` and `staggered` are located in the subdirectory `examples`. To recompile these programs or any programs within the ALPScomm library, the user must issue the `make World` command from within the ALPScomm directory.

3.2.2 Compiling new programs

When compiling your own programs utilizing the ALPS library, you will need to include the `alpscube.h` file in your C program. This file resides in the `ALPScomm/include` subdirectory. When compiling your program you must specify the pathname of the subdirectory, containing the include file, as a compiler option (i.e. `-Ipathname/ALPScomm/include`).

In order to link the ALPS library, there are two library files you must link: libcube.a and libcomm.a. These files reside in the ALPScomm/lib subdirectory. To link these libraries you must include the proper linker options (i.e. `-Lpathname/ALPScomm/lib -lcomm -lcube`).

If your program utilizes LAPACK functions then you must also link the proper LAPACK libraries (i.e. for example, on Cornell's SP2 the LAPACK libraries are located under the subdirectory `/usr/local/lib`, so the corresponding linker options are `-L/usr/local/lib -llapack -lblas -lxlf90`).

The ALPScomm/examples subdirectory contains implementations of the STAP algorithms described in section 2. These programs require the ESSL math library package, which is linked using the `-lessl` linker option.

3.3 Running the example STAP programs

It is assumed that the user will create STAP datacubes from his own source. Users must present radar datacubes in the ALPS pdc format described in the ALPSmanual [1]. If the data is created synthetically in MATLAB, the ALPS library provides functionality for creating files in the pdc format.

The steps below describe how to prepare your own data in MATLAB for processing by the demonstration STAP programs, and how to execute the programs.

3.3.1 Step 1 - Creating ALPScube data files in MATLAB

It is possible to create synthetic STAP data in MATLAB. Our software requires that the datacube is a three dimensional MATLAB array with the dimensions corresponding to `Range`, `Chan`, and `Pri`, in some order. After starting MATLAB, first you must do

addpath *path*/ALPScomm/matlab

where *path* is the path specifying the location of the `ALPScomm/matlab` subdirectory.

Ascertain which dimensions of the MATLAB matrix correspond to which physical parameters (`Range`, `Chan`, and `Pri`). If the data generated by MATLAB is a two dimensional array, then two of the parameters are likley combined into a single dimension. You must determine which of the two parameters are ordered consecutively, or fastest-varying, in that dimension. Once this is determined, the order of the dimensions in the three dimensional array can be established.

For example, in Figure 3.1, a two-dimensional matrix is illustrated, with the rows spanning the set of Ranges, and the columns spanning the combined parameters of `Pri` and `Chan`. The correct labeling of the dimensions is '`Range`, `Pri`, `Chan`'. The first MATLAB index corresponds to `Range` and is easily determined. The 2nd dimension parameter label is '`Pri`' because the `Pri`'s are grouped together consecutively for each `Chan`, and thus the `Pri`'s vary faster than the `Chan`'s.

In Figure 3.2, the dimensions are ordered as (`Pri`, `Chan`,`Range`) following the same reasoning as in the previous example.

Once the dimensions are ordered then the matrix can be written to a datacube file.

Writing MATLAB matrix to ALPScube file

In order to write a parallel datacube file, the user can run the following MATLAB command in the ALPS MATLAB subdirectory:

```
makecube(data, 'filname', 'orientation', 'datatype', Range, Chan, Pri)
```

where:

- `data` is the data matrix

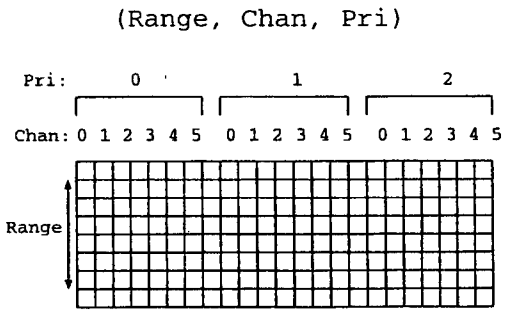


Figure 3.1: Example 2-d MATLAB matrix (Range, Chan, Pri)

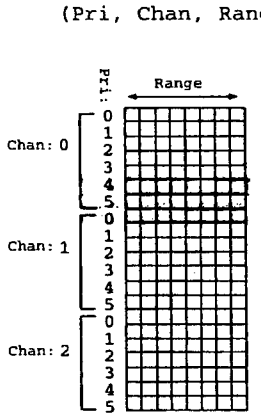


Figure 3.2: Example 2-d MATLAB matrix (Pri, Chan, Range)

- `filename` is a string specifying the filename of the ALPScube
- `orientation` is a string listing the order of the dimension names as determined in step 2 above (i.e. 'range chan pri', 'pri chan range', 'range pri chan', etc)
- `datatype` is a string specifying the datatype: 'complex' or 'double_complex'
- `Range` specifies length of range
- `Chan` is number of channels
- `Pri` is number of pri's

If the data matrix is already in 3-d format then the last 3 parameters are optional.

This function writes the datacube to a parallel data cube file (.pdc format).

Creating a set of Standard Steering Vectors

The `steercube` MATLAB command creates a set D of standard steering vectors where `D = kron(dftmtx(Pri)',dftmtx(Chan'))`.

To create a cube of steering vectors with `Chan`·`Pri` direction vectors D, the user should execute: `steercube('filename', chan, pri, datatype )`

where:

- `filename` is a string specifying the filename of the ALPScube,
- `chan` is number of channels
- `pri` is number of pri's.
- `datatype` is a string specifying the datatype: 'complex' or 'double_complex'

This function writes the steering cube to a file in .pdc format.

3.3.2 Step 2 - executing demo STAP implementations

Two programs, **overlap** and **staggered**, reside in the `ALPScomm/examples` subdirectory. These programs implement the two different methods of data distribution described in section 2.1. Each program takes command line parameters which specify whether to use the FullUpdateStap or PredictRes STAP algorithm. The command line parameters are specified below.

The steps below outline the procedure for execution on the Cornell SP2. There are two modes of execution, interactive and batch. Interactive mode entails running the programs at the command prompt with immediate results. However this limits the user to at most 4 processors.

Batch mode entails submission a request for processor allocation and execution to a queue and waiting for the program to be executed at some unspecified future time. Sample batch files have been provided in the ALPScomm/examples subdirectory for this purpose and are specified below.

Interactive Mode

Before execution, the ALPS library requires that the environmental variable **CUBEDEFINITIONS** be set to the full pathname of the `ALPScomm/dataformat` subdirectory.

I) "Overlap" distribution (described in section 2.1.1)

```
overlap datafile steerfile outputfile offset overlap fft alg
```

`datafile`: name of data file (excluding .pdc extension)

`steerfile`: name of steercube file (excluding .pdc extension)

`outputfile`: name of output data file (excluding .pdc extension)

`offset`: offset value

`overlap`: overlap value

`fft`: 0=no doppler processing, 1= doppler processing

`alg`: 0=FullUpdateStap , 1=PredictRes

II) "Staggered" distribution (described in section 2.1.2)

```
staggered datafile steerfile outputfile numcubes fft alg
```

`datafile`: name of data file (excluding .pdc extension)

`steerfile`: name of steercube file (excluding .pdc extension)

`outputfile`: name of output data file (excluding .pdc extension)

`numcubes`: parameter specifying number of overlapping cubes as described in Figure 2.4.

`fft`: 0=no doppler processing, 1= doppler processing

`alg`: 0=FullUpdateStap , 1=PredictRes

Batch Mode

The commands below submit batch jobs on cornell's SP2 machine (splong.tc.cornell.edu) for each of the indicated operations. All the batch files specified below are present in the ALPScomm/examples subdirectory; they expect that the input data file be named `data.pdc` and the steering vector file be named `steer.pdc` and they allocate 4 processors for execution.

NOTE: The files must first be edited to set the pathname of the CUBE-DEFINITIONS environmental variable. You may also modify the number of allocated processors or the program command-line parameters as desired.

These commands must be execeuted from the ALPScomm/examples sub-directory.

I) "Overlap" distribution (with parameters: offset = 1, overlap = 3)

a) with FullUpdateStap algorithm

without doppler processing: `llsubmit overlap_fu.batch`

with doppler processing: `llsubmit overlap_dp_fu.batch`

b) with PredictRes algorithm

without doppler processing: `llsubmit overlap_pr.batch`

with doppler processing: `llsubmit overlap_dp_pr.batch`

II) "Staggered" Distribution (with parameters: numcubes = 3)

a) with FullUpdateStap algorithm

without doppler processing: `llsubmit staggered_fu.batch`

with doppler processing: `llsubmit staggered_dp_fu.batch`

b) with PredictRes algorithm

without doppler processing: `llsubmit staggered_pr.batch`

with doppler processing: `llsubmit staggered_dp_pr.batch`

3.3.3 Step 3 - Read output of algorithm

To read the output file of a STAP program in MATLAB, start MATLAB, and do

addpath *path*/ALPScomm/matlab

where *path* is the path specifying the location of the `ALPScomm/matlab` sub-directory. To read the output file of a STAP program, run:

`[data,format] = mat_readcube('filename')`

where `filename` is filename of output file. `data` is the 2-d matrix containing the results. The number of rows are equal to the number of direction vectors, and the number of columns are equal to the length of `Range`.

Bibliography

[1] Richard H. Weitkunat, *ALPS Parallel Library for Three-Dimensional Data Redistribution,* Department of Electrical Engineering, Cornell University, August 1999

[2] Richard H. Weitkunat, *Improvements To Parallel Libraries For Three-Dimensional Data Redistribution,* Department of Electrical Engineering, Cornell University, January 1999

ALPS PARALLEL LIBRARIES FOR THREE-DIMENSIONAL DATA REDISTRIBUTION

A.W. Bojanczyk, B. Durie, W. Kostis and R.H. Weitkunat

Contents

List of Figures

List of Tables

Chapter 1

Data Distribution and Redistribution

We start by presenting an example illustrating functionality of the ALPS library. Let us consider a *distributed* three-dimensional matrix of data (hereafter referred to as a "data cube") residing on a mesh of parallel processors that are logically (although not necessarily physically) organized into a three dimensional rectangular topology, or "process cube." We would like to reorganize this data in a series of reconfigurations, as graphically illustrated in Figure 1.1. Each configuration corresponds to a computational module in the ALPS library. A sample program which implements this series of reconfigurations is shown in section 1.1.

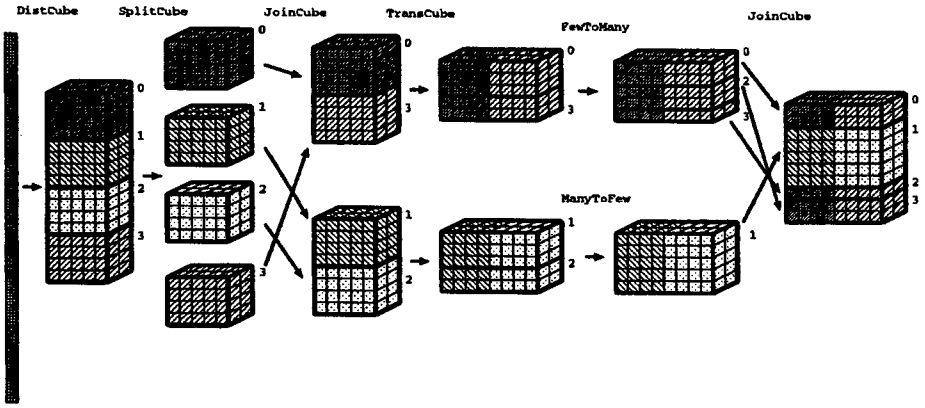


Figure 1.1: Graphic illustration of data distribution

In the example shown, we are operating on four processors. The processors are labeled 0 to 3, and the 0th processor is referred to as the *root* processor.

First we would like to create a data cube that occupies all four 4 processors.

We start with a three dimensional matrix of data stored on the root processor in a linear array, *data*.

```
startCube    = DistCube(data, dims, blocksizes, startType, startComm);
```

Each processor simultaneously issues the **DistCube** command, which copies and distributes the data residing on the root processor so that it is divided up and distributed to the four processors. The function returns a handle, *startCube*, for identifying the newly created ALPScube.

In Figure 1.1, the 3-dimensional data cube is depicted as a single 3-d matrix. It is divided into portions residing on separate processors, outlined by the dark heavy lines. The light lines illustrate the individual data elements. (The data is distributed in a particular pattern known as Block-Cyclic distribution, discussed further in section 2.2.1.)

Besides the data itself, the ALPScube specifies the context in which the data resides, namely which processors belong to the process cube upon which the data is distributed. All operations henceforth performed on this data cube must be performed simultaneously by all the processors that belong to it. In order to allow each processor to perform separate operations on its portion of the data cube, an ALPScube must be created for each processor which encapsulates only the portion of data residing on the local processor. That is accomplished in the next step.

```
NumberOfPieces = 4;
SplitDimension = 2;
retCubeList = SplitCube(startCube, NumberOfPieces, SplitDimension);
```

All the processors next employ the **SplitCube** command to copy and repartition the single ALPScube into four new smaller ALPScubes. **SplitCube** is used to divide an ALPScube into equal sub-cubes, along a single dimension. No data redistribution takes place, but the data is duplicated for the new ALPScubes. It is the process cube that is subdivided.

The result of this particular operation is to encapsulate each processor's portion of the data as a separate ALPScube. This is illustrated in Figure 1.1 by the physical separation of the blocks of data residing on separate processors.

The functional difference between the new ALPScubes and their progenitor is purely one of context. The four new ALPScubes each encompass a single processor, and each processor may perform its own separate operations on the smaller ALPSCube, independently of each other. The distributed portions of the original data matrix can be managed as separate matrices in this fashion.

The **SplitCube** command only works on process cubes that have a linear shape, or have only one dimension with a length greater than one. The **SplitCube3d** command was created to overcome this particular restriction, although it also only can split cubes along a single dimension at a time.

Besides creating new ALPScubes which are subsets of a larger ALPScube, it is also possible to join ALPScubes together, as shown next.

```
JoinDimension = 2;
joinedCubes = JoinCube(retCubeList, JoinDimension, joinComm);
```

The **JoinCube** command is next utilized to create two new ALPScubes from the four smaller ones. **JoinCube** is used to join cubes along a single dimension. Again, as in **SplitCube**, no data are redistributed between processors. All four processors execute the **JoinCube** command; however, in this case, processors 0 and 3 join to form an ALPScube with their resident data, separate from the one formed by processors 1 and 2.

The means of organizing processors into process cubes involve utilizing MPI commands and *communicators*, and is discussed further in section 2.2.7. In this step, the two groups of processors ({0, 3}, and {1, 2}) are in effect acting independently of each other, since the ALPScubes involved do not span between the two groups.

The **JoinCube** command only works on process cubes that have a linear shape, or have only one dimension with a length greater than one. The **JoinCube3d** command was created as an extended version to overcome this particular restriction, although it also only joins cubes along a single dimension at a time.

So far, three different frameworks have been created for managing three copies of the same data matrix on the same processors. The first portrays the data as a single matrix distributed over 4 processors, the second as four matrices residing on separate processors, and the third as as two matrices each distributed over two processors.

Now we will actually redistribute the data cubes of the two ALPScubes created by the **JoinCube** command.

```
transposedCube = TransCube(joinedCube, newCubeType, newBlockSizes);
```

Processors 1 and 2 apply the **TransCube** command to the ALPScube that they have in common, and likewise for Processors 0 and 3. As shown in the figure, the data cubes of the new ALPScubes have had two of their axes transposed. The dimension that previously spanned over multiple processors now resides within a single processor, and vice versa. The process cubes themselves are not transposed.

In this example only two dimensions were transposed, although **TransCube** is capable of transposing all three simultaneously, as well as redistributing data in a block-cyclic pattern. See section 2.2.1 for further details.

So far, all the processors have executed the same sequence of ALPS functions in the same order. In the next stage of redistribution the sizes of the process cubes themselves are affected.

```
if(myid == 2)
        inputCube = NULL;
else
        inputCube = transposedCube;
```

```
ftmCube = FewToMany(inputCube, ftmComm);
```

Processors 0, 2, and 3 perform the **FewToMany** operation. (Processor 2 has to pass a NULL value for the input cube, instead of transposedCube). The purpose of this operation is to increase the number of processors over which an ALPScube is distributed. In the example shown, the ALPScube residing on processors 0 and 2, previously created by the **TransCube** command, is copied and evenly redistributed over processors 0, 2, and 3. But for the newly created ALPScube, processor 2 is logically ordered to come after processor 3. The processors 0, 2,and 3 have been reordered utilizing the same methods as before, during the previous **JoinCube** operation.

While this operation takes place, a different operation takes place on processors 1 and 2 (processor 2 takes part in both operations, but it does so sequentially, first participating in one operation, then the other).

```
NumProcsDim0 = 1;
NumProcsDim1 = 1;
NumProcsDim2 = 1;
mtfCube = ManyToFew(transposedCube, NumProcsDim0, NumProcsDim1, NumProcsDim2, NULL);
```

Processors 1 and 2 perform the **ManyToFew** command on their ALPSCube. The purpose of this command is to reduce the number of processors over which a data cube is distributed. In the example, a new ALPScube is created that resides on only one processor. The 2nd processor does not hold any data from the new ALPScube, nor does it receive a handle for it. Instead it is returned a NULL value.

In the last stage of this example, the resulting two ALPScubes are joined together. Notice that for the new ALPScube created, the processors are back in their original order. This is again due to the specified MPI communicator, which is the same one used by the first ALPScube.

```
finalCube = JoinCube(newCubeList, JoinDimension, startComm);
```

This new cube is not distributed evenly: processor #1 now has twice as much data as the other processors. Also, the data on processor #3, which was logically in the middle of the previous data cube, is now logically at the end of the resultant data cube. This cube can later be redistributed evenly using a redistribution function such as **TransCube, FewToMany, or ManyToFew**.

Although this example depicts a linear process cube, the example can effortlessly be extended to a 3-dimensional process cube.

1.1 Sample Code

```
#include <stdio.h>
```

```
#include "alpscube.h"

/* LOCAL PROTOTYPES */
ALPScube  FillCube(ALPScube cube);

main(int argc, char **argv)
{

/* VARIABLES */
int pri, chan, range;
int dim[3];
int blocksizes[3]= {0,0,0};
int newBlockSizes[3]= {0,0,0};
int myid;
int newid;
int JoinDimension;
int NumberOfPieces, SplitDimension;
int NumProcsDim0, NumProcsDim1, NumProcsDim2;
int color1;
int color2;

char *startType    = "Pri_Chan_Range_SC";
char *newCubeType  = "Range_Pri_Chan_SC";

MPI_Comm startComm;
MPI_Comm tempComm1;
MPI_Comm tempComm2;
MPI_Comm joinComm;
MPI_Comm ftmComm;

ALPScube startCube  = NULL;
ALPScube *retCubeList = NULL;
ALPScube joinedCube = NULL;
ALPScube transposedCube = NULL;
ALPScube inputCube = NULL;
ALPScube mtfCube = NULL;
ALPScube ftmCube = NULL;
ALPScube newCubeList[2];
ALPScube finalCube = NULL;
ALPScube finalCube2 = NULL;


/* Initialize argc, argv */
Initialize(&argc, &argv);

/* Set dimensions of ALPScube */
dim[0]=pri=16;
dim[1]=chan=32;
dim[2]=range=1024;
```

```
/* get processor id */
MPI_Comm_rank(MPI_COMM_WORLD, &myid);

/* Make Cube communicator */
startComm    = MakeCubeComm(MPI_COMM_WORLD, 1, 1, 4);

/* Distribute data among processors and create data cube */
startCube = DistCube(data, dim, blocksizes, startType, startComm));

/* Split cube into 4 pieces along 2nd dimension */
NumberOfPieces = 4;
SplitDimension = 2;
retCubeList = SplitCube(startCube, NumberOfPieces, SplitDimension);

/* Make communicator for 1st upcoming join */
if(myid ==0 || myid ==3) /* I am the 1st or the 4th processor */
color1 = 0;
else /* I am the 2nd or 3rd processor */
color1 = 1;

MPI_Comm_split(startComm, color1, myid, &tempComm1);

/* new process cubes are half as long as original */

joinComm    = MakeCubeComm(tempComm1, 1, 1, 2);

JoinDimension = 2;
joinedCube = JoinCube(retCubeList, JoinDimension, joinComm);

transposedCube = TransCube(joinedCube, newCubeType, newBlockSizes);

/* Do ManyToFew  on processors 1 and 2 */
if(myid == 1 || myid == 2) /* If I am Processor 1 or 2 */
{
NumProcsDim0 = 1;
NumProcsDim1 = 1;
NumProcsDim2 = 1;
mtfCube = ManyToFew(transposedCube, NumProcsDim0, NumProcsDim1, NumProcsDim2, NULL);
}

/* Make communicator for FewToMany */
/* Switch order of processor 2 and 3 */

if(myid == 2) /* Rerank 2 as 3 for new communicator */
newid = 3;

else if(myid == 3) /* Rerank 3 as 2 for new communicator */
newid = 2;
else
newid = myid;
```

```
if(myid == 1 ) /* Processor 1 not included */
color2 = 0;
else
color2 = 1;

MPI_Comm_split(startComm, color2, newid, &tempComm2);

/*Do FewToMany  on processors 0, 3, and 2 */
if(color2 == 1)
{
ftmComm = MakeCubeComm(tempComm2, 1, 1, 3);

/* if myid==2, we dont have any cubedata to submit to FTM */
if(myid == 2)
inputCube = NULL;
else
inputCube = transposedCube;

ftmCube = FewToMany(inputCube, ftmComm);
}

/* Join cubes together: create list for JoinCube */

if(mtfCube) {
newCubeList[0] = mtfCube;
newCubeList[1] = NULL;
}
else if(ftmCube) {
newCubeList[0] = ftmCube;
newCubeList[1] = NULL;
}
else
newCubeList[0] = NULL;

finalCube = JoinCube(newCubeList, JoinDimension, startComm);

finalCube2 = TransAnyCube(finalCube, finalCube->definition->name, blocksizes);
WriteCube("example", finalCube2);

Finalize();
}

/* ALPScube  FillCube(ALPScube cube) */
/* fill cube with test data that indiactes coordinates. */
/*  Only for block-distributed cubes. */
ALPScube  FillCube(ALPScube cube)
{
int i, j, k;
```

```
CMPX ***scData = (CMPX ***)cube->data;
CMPX scNum ;
DCMPX ***dcData = (DCMPX ***)cube->data;
DCMPX dcNum ;
int factor[3]={10,10, 0};
int dim;


for(i=0; i<2; i++)
{
dim = cube->dim[i+1];
while(dim > 10)
{
dim /= 10;
factor[i] *= 10;
}
}
factor[0]*=factor[1];

for(i=1; i<= cube->ldim[0]; i++)
  for(j=1; j<= cube->ldim[1]; j++)
    for(k=1; k<= cube->ldim[2]; k++)
{
switch(cube->definition->Datatype) {
case MPI_COMPLEX:
scNum.re = scNum.im =
  (i+cube->firstId[0])*factor[0]
+ (j+cube->firstId[1])*factor[1]
+ k+cube->firstId[2];

scData[i-1][j-1][k-1] = scNum;
break;
case MPI_DOUBLE_COMPLEX:
dcNum.re = dcNum.im =
  (i+cube->firstId[0])*factor[0]
+ (j+cube->firstId[1])*factor[1]
+ k+cube->firstId[2];

dcData[i-1][j-1][k-1] = dcNum;
break;
default:
parError("FillCube : Datatype not supported");

}

}

return (cube);
}
```

Chapter 2

ALPScube C functions

2.1 Introduction

This document describes a collection of C functions that are designed to facilitate the organization and distribution of three dimensional data matrices across parallel processors.

2.2 The ALPScube

An ALPScube, from the programmer's point of view, is an identifier used to refer to a distributed data matrix, and to distinguish one ALPScube from another. But this identifier serves a hidden purpose as well.

In actuality, the ALPScube is a pointer to a data structure. Each processor that belongs to the distributed cube's processer mesh maintains a local data structure which contains information describing the global data matrix, as well as information relevant to its portion of the data.

The ALPScube data structure contains the following information about the data cube:

1. a pointer to a 3-d array of the local portion of the data matrix
2. the global data matrix's dimensions
3. the global data matrix's blocksizes
4. the dimensions of the processor's local portion of the data matrix
5. the global coordinates of the local cube's origin
6. the dimensions of the process cube
7. the local processor's coordinates in the process matrix.
8. information from the ALPStype (see section 2.2.6 below).

2.2.1 The Data Cube and Block Cyclic Data Distribution

The data of an ALPSCube is initially distributed according to a set pattern.

The data distribution functions **CollCube**, **DistCube**, **TransCube**, **ManyToFew**, and **FewToMany** all employ the commonly used *block-cyclic* data distribution. The ALPSCube that is both created and read by these functions is distributed in a three-dimensional block-cyclic fashion. The **TransCube** and **DistCube** functions take block sizes for all three dimensions as a parameter.

2.2.2 The Process Cube

The process cube refers to the actual processors that contain a portion of the distributed data matrix. This group of processors is identified as a whole by an MPI "Communicator" (an integer handle). The communicator serves the same purpose for the processors as the ALPSCube serves for the distributed data matrix.

The processors are logically organized into a three-dimensional rectangular mesh, and each processor is assigned a set of three cartesian coordinates. See figure 2.2 for an illustration of a process cube. In this document, the x dimension of the process cube always refer to the dimension that corresponds to the slowest-varying dimension of the data cube, while the z-dimension refers to the fastest-varying dimension, as shown in figure 2.3.

2.2.3 Block Cyclic Distribution in one dimension

The distribution pattern is easily described in the one-dimensionsal case. Figure 2.1 illustrates an array which is to be distributed in block-cyclic fashion amongst a group of three processors with a block size of 2.

To put it more formally, given a linear array of N elements and a blocksize of b, the array is split into $\lceil \frac{N}{b} \rceil$ blocks, and the blocks are cyclicly distributed across P processors in round-robin order.

Processors 0 thru $P_\alpha = \lfloor \frac{N}{b} \rfloor \bmod P$ each receive $\lceil N \bmod (Pb) \rceil$ blocks. If $P_\alpha < P-1$, then processors $P_\alpha + 1$ thru $P-1$ receive $\lceil N \bmod (Pb) \rceil - 1$ blocks. If N is not a multiple of b, then the very last block, received by processor P_α, will contain $N \bmod b$ elements.

2.2.4 Block Cyclic Distribution in three dimensions

The extension to three dimensions follows easily. The pattern is applied to each dimension independently. Each dimension is partitioned by it's own blocksize, and each block is assigned to a processor index along its respective dimension. By applying this to all three dimensions, the cube is effectuvely partitioned into 3-d blocks, and each block's assigned processor is identified by the three processor indicies which match the processor's cartesian coordinates.

Each processor stores its locally assigned data as a three dimensional matrix. Figure 2.2 illustrates the data matrix distributed over the processor mesh, or process cube.

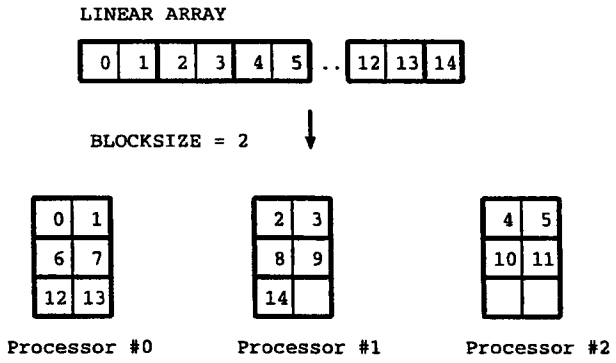


Figure 2.1: Example of block-cyclic distribution in one dimension

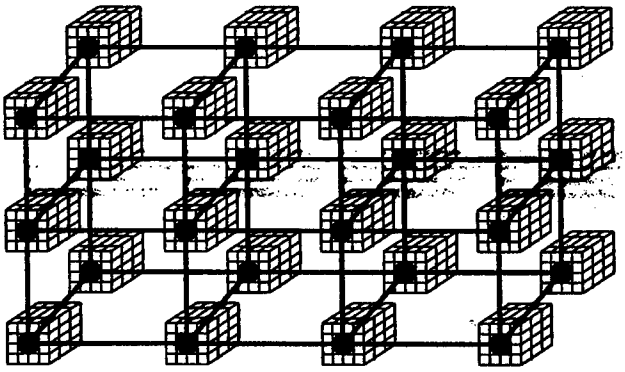


Figure 2.2: Illustration of processor mesh and distributed data cube

2.2.5 Exceptions to the Block-Cyclic distribution

ALPScubes do not have to be distributed in a block-cyclic format . The final ALPScube created in the introductory example is not distributed in block-cyclic fashion. The function **TransAnyCube** can in fact read such an ALPScube. The data is assumed only to be contiguously distributed along each dimension.

2.2.6 ALPScube Types

When an ALPSCube is created by the functions **DistCube** and **TransCube**, a cube type must be specified as a parameter. A cube type descibes certain properties of the ALPScube:

- the relative orientation of the data cube's axes
- the data type of the data elements

MPI Datatype	Equivalent C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_COMPLEX	struct { float re; float im; }
MPI_DOUBLE_COMPLEX	struct { double re; double im; }

Table 2.1: Supported Datatypes

- restrictions on other characteristics of the ALPScube

ALPScube types are defined in an ascii file named "stap.fmt". The evironmental variable `CUBEDEFINITIONS` must contain the name of the directory in which the file resides (see section 4.3 for more details).

Several pre-defined types may exist in the file, but the user can modify the file and create new ALPScube types as desired. The format of an ALPStype definition is as follows:

```
{ALPScube_type_name} {
        MPI_Datatype {datatype}
        permute {permute vector}
        restriction (condition)
}
```

The string *ALPScube_type_name* is an arbitrary alphanumeric label. The string *datatype* specifies the datatype of the data's elements. It must be a MPI datatype: supported types are listed in table 2.1.

The *permute vector* is of the format [x y z] where x, y, and z specify the orientation of the axes of the data cube (e.g. [0 1 2], [2 0 1], [2 1 0], etc) with respect to an arbitrary 'standard' orientation of [0 1 2], the *standard form*. The z-axis is always the fastest-varying dimension, regardless of its orientation with respect to the standard form. The x-axis is the slowest varying dimension. Figure (2.3) llustrates the actual layout of data in memory for a 3-d matrix with dimensions (2,2,2).

The permute vector only has practical meaning in the context of transposition. For example, when a data cube with a permute vector of [0 1 2] is transposed into a cube with permute vector [2 1 0], the effective result is to transpose the x and z dimensions (a 2-d transpose) while leaving the y-dimension as is. The source and destination permute vectors specify the two matrices' orientation with respect to each other. In order to transpose a cube's axes as desired, an ALPStype cube definition must exist with the necessary permute vector to achieve the new desired orientation of the axes.

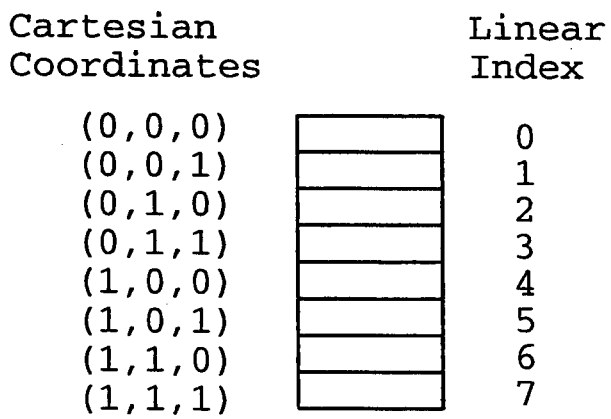


Figure 2.3: Illustration of data arranged in memory

Such an ALPScube is distributed over a group of processors (technically, the data is distributed amongst processes, not processors. These processes could all exist on the same physical processor, or each process could exist on its own separate processor. For the remainder of this document, we do not distinguish between processes and processors - the assumption is that each process runs on its own separate processor).

2.2.7 Creating a Cube Communicator

MPI_Comm MakeCubeComm(MPI_Comm *parentComm*, int x, int y, int z)

This function is used to organize a set of processors into a three dimensional rectangular mesh and create a new MPI communicator to be used when creating an ALPScube. All the processors to be included in the new communicator must already belong to a pre-existing parent communicator (specified as *parentComm*), and all members of *parentComm* must participate.

Parameters:

- *parentComm*: the MPI communicator for the group of processors.
- x: desired length of the process cube's x-dimension.
- y: desired length of the process cube's y-dimension.
- z: desired length of the process cube's z-dimension.

The total number of processors belonging to *parentComm* must equal the number of processors in the desired three-dimensional process cube, equal to $x \times y \times z$.

Return Values:

The function returns a new MPI communicator with the same processors as *parentComm* and has each processor associated with cartesian coordinates. In case of error, program execution is halted.

2.3 Creating an ALPSCube

This section describes functions that create an ALPSCube

2.3.1 Create an ALPSCube from a linear data buffer

ALPScube DistCube(void *_linArray_, int *_dim_, int *_bs_, char *_cDefStr_, MPI_Comm _totalComm_)

This function distributes a linear data array into a distributed ALPScube. The data is stored in memory as illustrated in figure 2.3.

Parameters:

- *linArray*: pointer to data buffer.
- *dim*: an array of three integers specifying the lengths of the global matrix's three dimensions.
- *bs*: an array of three integers specifying the blocksizes for the three dimensions, for block-cyclic distribution.
- *cDefStr*: pointer to string specifying the desired ALPScube type of the new ALPScube.
- *totalComm*: MPI communicator created by **MakeCubeComm**.

If a blocksize $bs[d]$ for any given dimension d is zero, then the actual blocksize used will be $bs[d] = \lceil \frac{dim[d]}{P_d} \rceil$, where P_d is the length of the process cube in the dth dimension.

The fastest-varying dimension (z-dimension) of the data cube as it is stored in the linear buffer will be the fastest-varying dimension of the ALPScube ; no transposition is performed by this function (i.e. the ALSPcube's permute vector has no effect on the distribution of the data in this function).

Return Values:

Returns the new ALPScube. In case of error, program execution is halted.

2.3.2 Creating a "blank" cube

ALPScube MakeCube(int *x*, **int** *xb*, **int** *y*, **int** *yb*, **int** *z*, **int** *zb*, **char** ***cDefStr*, **MPI_Comm** *totalComm*)

The **MakeCube** function will return a pointer to a new ALPScube with the necessary memory space allocated. The

Parameters:

- *x*: length of the global data cube's x dimension.
- *y*: length of the global data cube's y dimension.
- *z*: length of the global data cube's z dimension.
- *xb*: blocksize for the x-dimension.
- *yb*: blocksize for the y-dimension.
- *zb*: blocksize for the z-dimension.
- *cDefStr*: pointer to string specifying the desired ALPScube type of the new ALPScube.
- *totalComm*: MPI communicator created by **MakeCubeComm**.

If a blocksize for any given dimension is zero, then the actual blocksize used will be $\lceil \frac{x}{P} \rceil$, where x is the length of the data matrix and P is the number of processes in that dimension.

Return Values:

Returns the new ALPScube. In case of error, program execution is halted.

2.4 Retrieve an ALPScube into a linear array

void *CollCube(ALPScube *totalCube*)

This function collects the distributed data into a linear unblocked array on the root processor (the processor with a rank of 0). The array buffer space is allocated by the function.

Parameters:

- *totalCube*: the ALPScube to be collected into a single buffer.

Return Values:

On the root processor, the function returns a pointer to the linear array.
On the remaining processors, the function returns the NULL value.

2.5 Reading and Writing ALPScubes from/to disk

int WriteCube(char **filestem*, ALPScube *totalCube*)

This function saves the distributed data in an ALPScube into a single file. The file format is is a proprietary format (Parallel Data Cube, or PDC) for storing information about the ALPScube as well as the data itself.

The function automatically appends the extension ".pdc" to the specified filename.

Parameters:

- *filestem*: pointer to string specifying the filename.
- *totalCube*: the ALPScube to be written to a file.

ALPScube ReadCube(char **filestem*, int *bx*, int *by*, int *bz*, MPI_Comm *totalComm*)

This function reads the ALPScube stored in the specified file (excluding the ".pdc" extension) and distributes the data in block-cyclic format to all the calling processes in *totalComm*.

Parameters:

- *filestem*: pointer to string specifying the filename.
- *xb*: blocksize for the x-dimension.
- *yb*: blocksize for the y-dimension.
- *zb*: blocksize for the z-dimension.
- *totalComm*: MPI communicator that specifies the processor mesh.

If a blocksize for any given dimension is zero, then the actual blocksize will be computed as $\lceil \frac{x}{P} \rceil$, where x is the length of the data matrix and P is the number of processes in that dimension.

Return Values:

Returns the new ALPScube. In case of error, program execution is halted.

2.6 Redistributing an ALPScube over a different number of processors

The process cube upon which an ALPScube is distributed can be decreased in number of processors using **ManyToFew1d** for linear process cubes (only one of the three dimensions of the process cube has a length greater than one). For general 3-d process cubes, **ManyToFew** can be utilized.

2.6.1 Reducing the number of processors

ALPScube ManyToFew1d(ALPScube *totalCube*, **int** *numProcs*)

The **ManyToFew1d** function creates a new ALPScube that occupies a smaller subset of the processor topology occupied. Both the old and new process cubes must be linear, i.e. have length only in one dimension. The new length of the long dimension is shortened to the value of *numProcs*. The data cube will be redistributed with maximal blocking (insert equation) in all dimensions.

All processors that belong to *totalCube* must call the **ManyToFew1d** function.

Parameters:

- *totalCube*: ALPScube to be redistributed.
- *numProcs*: The new (shorter) length of the processor array.

Return Values:

Those processors that belong to the new subset of processors will be returned the new ALPScube. This ALPScube will have a new communicator that only includes the processors belonging to the new subset of processors. The processors that do not belong to the new subset will be returned the NULL value.

ALPScube ManyToFew(ALPScube *totalCube*, **int** *numProcsX*, **int** *numProcsY*, **int** *numProcsZ*, **MPI_Comm** ***pnewcomm*)

The **ManyToFew** function creates a new ALPScube that occupies a subset of the occupied process cube. The parameters *numprocsX*, *numProcsY*, and *numProcsZ* specify the lengths of the new, smaller, process cube. The data cube will be redistributed in block distribution format in all dimensions. The optional parameter *pnewcomm* is a pointer to an MPI_Comm variable that will contain the new communicator of which the calling process is a member. All processors that belong to *totalCube* must call the **ManyToFew** function.

Parameters:

- *totalCube*: ALPScube to be redistributed.

- *numProcsX*: The new length of the process cube's x-dimension.
- *numProcsY*: The new length of the process cube's y-dimension.
- *numProcsZ*: The new length of the process cube's z-dimension.
- *pnewcomm*: optional pointer to MPI_Comm variable.

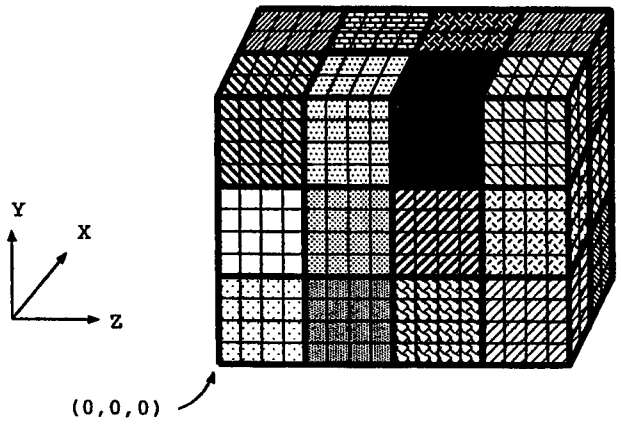
Return Values:

Those processors that belong to the new subset of processors will be returned the new ALPScube. This ALPScube will have a new communicator that only includes the processors belonging to the new subset of processors.

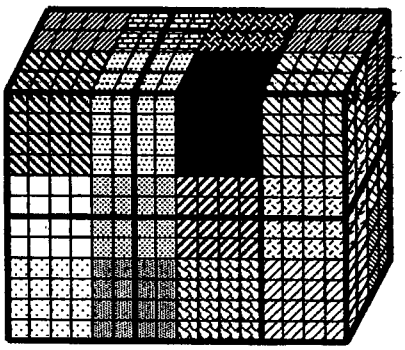
The processors that do not belong to the new subset form a separate subset which is assigned a new communicator (this communicator will not have Cartesian coordinates associated with its members). These processors are returned the NULL value. In any case, if the parameter *pnewcomm* is supplied, this variable will contain the new communicator that the processor belongs to.

Example of ManyToFew

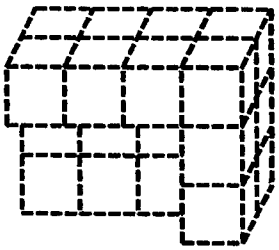
Figure 2.4 illustrates the use of **ManyToFew** on a data cube with a process cube of dimensions (2, 3, 4). Figure 2.4(b) shows the new process cube with dimensions (1, 2, 3). The ordering of the data is contiguous along each dimension. Figure 2.4(c) illustrates the processors from the original process cube that are not members of the new process cube.



(a) Initial Data Cube



(b) ManyToFew used to create smaller process cube



(c) Excluded empty processors

Figure 2.4: Graphic illustration of ManyToFew

2.6.2 Increasing the number of processors

The process cube upon which an ALPScube is distributed can be increased in size using **FewToMany1d** for linear process cubes only (only one of the three dimensions of the process cube has a length greater than one). For general 3-d process cubes, **FewToMany** can be utilized, with much faster performance.

ALPScube FewToMany1d(ALPScube *origCube*, **MPI_Comm** *newComm*)

The **FewToMany1d** function does the opposite of **ManyToFew1d** - it creates a new ALPScube by redistributing *origCube* onto a greater number of processors. Both the old and new process cubes must be linear, i.e. have length only in one dimension. The new topology is given by the new communicator *newComm*. All the processors that belong to *newComm* must call this function. Processors that do not belong to the original ALPScube must set *origCube* to NULL. All processors that belong to the original cube must belong to the new communicator *newComm* as well, and their processor coordinates must be the same in *origCube*'s communicator as in *newComm*.

Parameters:

- *origCube*: ALPScube to be redistributed.
- *newComm*: MPI Communicator specifying new processor array.

Return Values:

The function returns the new ALPScube.

ALPScube FewToMany(ALPScube *origCube*, **MPI_Comm** *newComm*)

The **FewToMany** function can expand in all dimensions simultaneously, and is not restricted to linear process cubes. The new process cube is specified by the communicator *newComm*. All the processors that belong to *newComm* must call this function. Processors that do not belong to the original ALPScube must set *totalCube* to NULL. All processors that belong to the original cube must belong to the new communicator *newComm* as well, and their processor coordinates must be the same in *totalCube*'s communicator as in *newComm*.

Parameters:

- *origCube*: ALPScube to be redistributed.
- *newComm*: MPI Communicator specifying new process cube.

Return Values:

The function returns the new ALPScube.

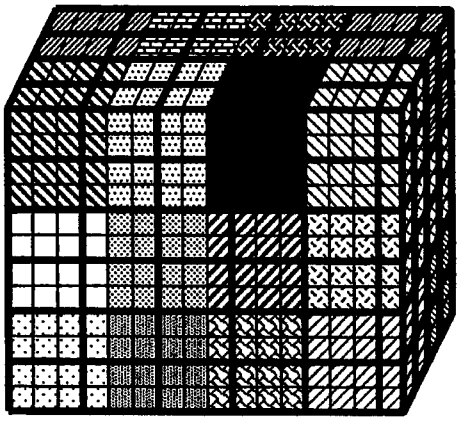


Figure 2.5: FewToMany used to create larger process cube

Example of FewToMany

Figure 2.5 illustrates the use of **FewToMany** to enlarge the process cube of the data cube shown in Figure 2.4(a). The new process cube has dimensions (4, 6, 6). Note that the X-Y plane of processes at the end of the Z axis has less data than the others, due to the nature of the block distribution pattern.

2.7 Transposition, Reblocking, and conversion between cube types

2.7.1 TransCube for block-cyclic ALPScubes

ALPScube TransCube(ALPScube *totalCube*, char **newType*, int **bs*)

The **TransCube** function creates a new ALPScube with ALPScube type *newType* (see section 2.2.6) and the data from *totalCube*.

This effectively allows the user to perform transposition on an ALPScube and/or reblocking on *totalCube*. Transposition is accomplished by specifying the new cube type *newType* that has the same datatype as *totalCube*, but a different orientation of the axes with respect to *totalCube*'s axes.

If *newType* has the same permute vector as the original, then no transposition will be performed. If *newType* is NULL, *totalCube*'s cube type will be used. If the user wishes to reblock without transposing, *newType* should be NULL.

Reblocking is accomplished by specifying the desired new blocksizes as an array *bs*, a pointer to an array of 3 integers, where each blocksize corresponds to the new cube's axes (i.e. bs[0] is the blocksize for the new cube's x-axis, and so forth). If *bs* is NULL, then *totalCube's* block sizes will be used instead. In this case, the blocksize vector will **not** be permuted to correspond to the transposed matrices axes; i.e. the x dimension's blocksize in *totalCube* will be the x dimension's blocksize in the transposed cube regardless of it's axis permutation.

If a blocksize $bs[d]$ is zero for any given dimension d, then the actual blocksize used will be $bs[d] = \lceil \frac{dim[d]}{P} \rceil$, where P is the number of processes in that dimension.

Parameters:

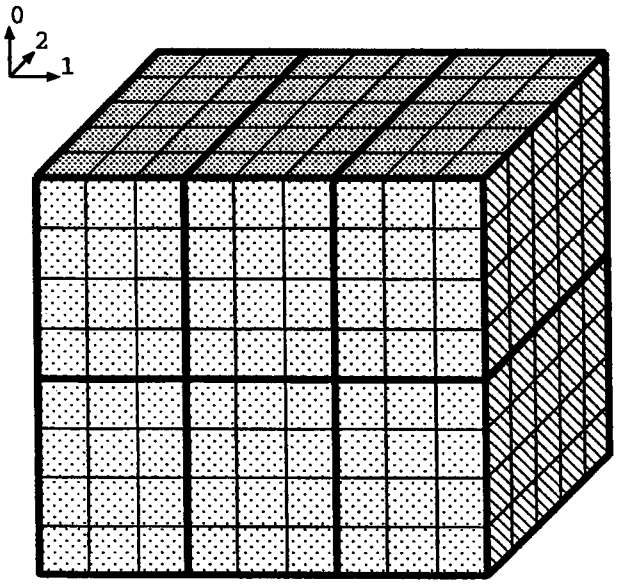
- *totalCube*: ALPScube to be transposed and/or reblocked.
- *newType*: string specifying the desired ALPScube type of the new ALPScube.
- *bs*: pointer to array of three integers specifying new blocksizes.

Return Values:

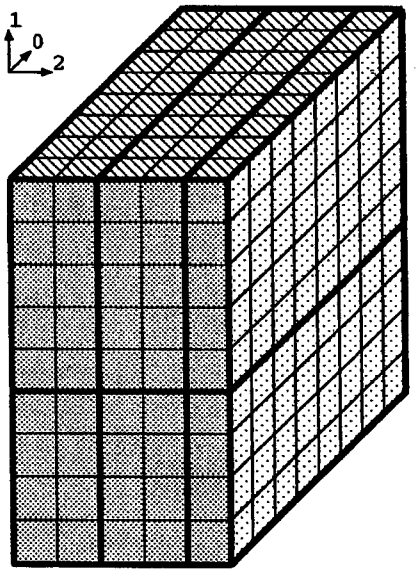
The function returns the new ALPScube.

Example of TransCube

Figure 2.6 illustrates the use of **TransCube** on a data cube with dimensions (8, 9, 5) on a process cube of dimensions (2, 3, 1). Figure 2.6(a) shows the new data cube with dimensions (9, 5, 8). The data is distributed in block-distribution fashion for both cubes.



(a) Initial Data Cube



(b) TransCube used to transpose data cube

Figure 2.6: Graphic illustration of TransCube

2.7.2 TransAnyCube for non-block-cyclic ALPScubes

ALPScube TransAnyCube(ALPScube *totalCube*, **char** ***newType*, **int** ***bs*)

This function behaves the same as **TransCube**, except that it can take as input an ALPScube which is not distributed in block-cyclic fashion, but in a contiguous yet irregular fashion in each dimension. Such an ALPScube may possibly be created by **JoinCube**. This function requires more overhead processing than **TransCube**, and should be used only for ALPScubes which require it.

The new ALPScube will be distributed in block-cyclic format.

Parameters:

- *totalCube*: ALPScube to be transposed and/or reblocked.
- *newType*: string specifying the new ALPScube type of the new ALPScube.
- *bs*: pointer to array of three integers specifying new blocksizes.

Return Values:

The function returns the new ALPScube.

2.7.3 TransCubeResize for resizing the process cube

ALPScube TransCubeResize(ALPScube *totalCube*, **char** ***newType*, **int** ***bs*, **MPI_Comm** *expandedComm*, **int** ***numProcs*, **MPI Comm** ***pnewcomm* **)**

This version of **TransCube** has several extra features. It allows the user to expand or reduce the number of processors that the new ALPScube will occupy. Its behavior is determined as follows:

1. If the MPI communicator *expandedComm* is provided, then *expandedComm* serves exactly the same role as *newComm* in **FewToMany**, and the result is the same as for **FewToMany**, except that transposition and reblocking can also be accomplished simultaneously.
2. If *expandedComm* is NULL, but *numProcs* is not NULL, then *numProcs* is an array of three integers specifying the dimensions of the process cube. The result is exactly the same as that of **ManyToFew**, except that transposition and reblocking can also be done simultaneously.

Parameters:

- *totalCube*: ALPScube to be transposed and/or reblocked.
- *newType*: string specifying the new ALPScube type of the new ALPScube.
- *bs*: pointer to array of three integers specifying new blocksizes.

- *expandedComm*: MPI Communicator specifying new (larger) process cube.
- *numProcs*: array of three integers specifying the lengths of the new (smaller) process cube's dimensions.
- *pnewcomm*: optional pointer to MPI_Comm variable.

Return Values:

Those processors that belong to the new set of processors will be returned the new ALPScube. This ALPScube will have a new communicator that only includes the processors belonging to the new set of processors.

If the new ALPScube occupies a smaller process cube, then he processors that do not belong to the new subset form a separate subset which is assigned a new communicator (this communicator will not have Cartesian coordinates associated with its members). These processors are returned the NULL value. In any case, if the parameter *pnewcomm* is supplied, this variable will contain the new communicator that the processor belongs to.

2.8 Dividing an ALPScube into smaller ALPScubes

ALPScube *SplitCube(ALPScube *totalCube*, int *numPieces*, int *sdim*)

SplitCube divides *totalCube* into equal-sized new ALPScubes, along a single dimension *sdim* of the ALPScube *totalCube*. The number of subpieces is given by *numPieces*. The cube is "cut" by planes that are orthogonal to the specified dimension *sdim*. The parameter *sdim* is either 0, 1, or 2, specifying the slowest-varying, middle, and fastest varying dimensions, respectively.

numPieces must be either an integral multiple or factor of the number of processors in the process cube. The function returns a pointer to a NULL-terminated array of ALPScubes. Each new ALPScube will have the same blocking parameters as *totalCube*.

Let P be the length of the process cube along the *sdim* dimension. There are two cases:

1. In the case that numPieces is an integer multiple of the number of processors P, then

$$numPieces = k * P$$

where k is an integer. In this case, each processor's local portion of the data cube will be subdivided into k pieces of equal length along the *sdim* dimension, and thus create k separate ALPScubes on *each processor*. Each ALPScube will reside completely within the confines of its process. **SplitCube** will return a NULL-terminated pointer to an array of k ALPScubes.

2. In the case that *numPieces* is an integral factor of P, then

$$numPieces * k = P$$

where k is an integer. In this case, each new ALPScube will occupy k processors along the *sdim* dimension. **SplitCube** will return a NULL-terminated array containing only the one ALPScube that the process belongs to.

Parameters:

- *totalCube*: ALPScube to be split. Process cube must be linear.
- *numPieces*: number of pieces to split ALPScube into.
- *sdim*: An integer value of 0, 1 or 2, specifying the x, y, or z dimension, respectively, that will be cut by the subdivisions.

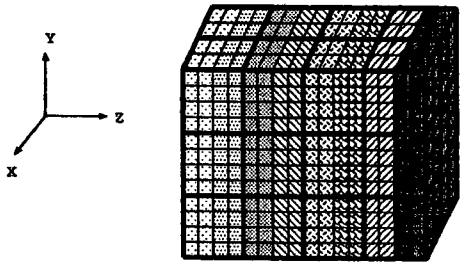
Return Values:

The function returns a NULL-terminated array of ALPScubes (even if there is only one ALPScube in the array).

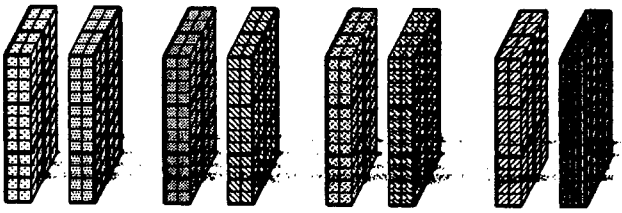
Example

In figure 2.7, an illustration of a data cube is shown being split into its subparts. Figure 2.7(a) shows the initial data cube. The dark heavy lines indicate process boundaries, while the light thin lines indicate individual data element boundaries. The data cube has dimension lengths (4, 12, 16) and occupies a process cube with dimensio ns (2, 3, 4). **SplitCube** is first used to split the cube into 8 new cubes alon g the second axis (figure 2.7(b)). The resulting cubes each occupy a process cube of (2, 3, 1), with two cubes occupying each processor. The actual data still resides on the same processors as in the original cube.

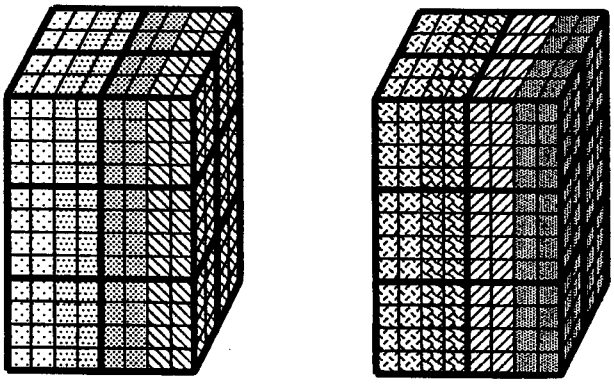
Figure 2.7(c) shows the results of creating 2 new cubes along the Z axis. Each new cube occupies a process cube with dimensions (2, 3, 2).



(a) Initial Data Cube



(b) SplitCube3d used to create 8 new Cubes along Z axis



(c) SplitCube3d used to create 2 new Cubes along Z axis

Figure 2.7: Graphic illustration of SplitCube3d

2.9 Combine Multiple ALPScubes into a single ALPScube

ALPScube JoinCube(ALPScube ****cubeList*, **int *jdim*, **MPI_Comm** *totalComm*)

This function creates a new ALPScube by copying the data from smaller ALPScubes into a single matrix. *cubeList* must be a NULL-terminated array of ALPScubes. The local portions of each ALPScube in the array are joined to form the local portion of the new ALPScube.

The parameter *jdim* specifies the axis corresponding to the direction in which the ALPScubes are joined.

The MPI communicator *totalComm* specifies the process cube of the new ALPScube. This ultimately specifies how the individual ALPScubes will be joined. The Cartesian coordinates of each process can be specified utilizing MPI functions, thus allowing an arbitrary ordering of processors.

The new ALSPcube will have the blocksizes of the input ALPSCube that is occupying the root processor.

However, there is a possible exception to this rule. If the new ALPScube does not happen to fit the standard block-cyclic distribution pattern defined by the blocksizes obtained as mentioned above, then each process will set the new blocksize to its local dimension length. This is a special case that signifies that the new ALPScube is not distributed in block-cyclic fashion, and each process's portion of the data matrix may possibly vary in length from process to process. The assumption is made that the data is not cyclic, but contiguously distributed. The alternative command **JoinAnyCube** is the same as **JoinCube** except that it takes an extra parameter, which when set to 1, will always set the new blocksize on each processor to the local dimension length. This is useful to avoid the possibility of accidently having an incorrect blocksize resulting by joining together datacubes that were created independently of each other.

In this case, the resultant data cube must first be reblocked. A special version of **TransCube**, **TransAnyCube**, should be used to reblock the resultant cube as desired before any other function is allowed to process it. **TransAnyCube** takes the same parameters as **TransCube**.

Parameters:

- *cubeList*: ALPScube to be split.
- *jdim*: An integer value of 0, 1 or 2, specifying the x, y, or z dimension, respectively, along which the ALPScubes will be joined.
- *totalComm*: MPI communicator specifying the new ALPScube's process cube. Must be linear (have length in only one dimension).

Return Values:

The function returns the new ALPScube.

ALPScube JoinCube(ALPScube **cubeList*, int *jdim*, MPI_Comm *totalComm*)

Parameters:

- *cubeList*: ALPScube to be split.
- *jdim*: An integer value of 0, 1 or 2, specifying the x, y, or z dimension, respectively, along which the ALPScubes will be joined.
- *totalComm*: MPI communicator specifying the new ALPScube's process cube. Must be linear (have length in only one dimension).
- *IgnoreBlockSize*: there are two choices:

  0: works the same as **JoinCube**.

  1: sets blocksize on each processor to local dimension length (where dimension is *jdim*).

2.10 Split ALPScube into overlapping ALPScubes

SplitStaggeredCube and **RRSplitStaggeredCube** split an ALPScube into a sequence of subcubes that overlap in the 0th dimension. They work by first transposing the input cube in the same fashion that **TransCube** would if it were called with the *newType* parameter and with new blocksizes of {0,0,0}, but with the important difference that some data elements are duplicated in order to create subcubes that overlap in the 0th dimension of the new subcubes. **TransStaggeredCube** is utilized to accomplish the duplication and distribution of data; the two functions mentioned above extend this functionality by also splitting the data into subcubes. They differ in the order in which the subcubes are distributed.

The new cubes overlap in the 0th dimension (slowest-varying or outermost) of the new cubetype's orientation (see section 2.2.6).

2.10.1 Consecutive distribution of subcubes

ALPScubeList SplitStaggeredCube(ALPScube *totalCube*, char **newType*, int *offset*, int *overlap*, int *duplicate*, int *wraparound*)

The subcubes are distributed such that consecutive cubes on each processor are ordered consecutively by their dimension indices. This is illustrated in figure 2.8.

Each subcube will have dimensions {(overlap+offset), DimLength[1], DimLength[2]}, where DimLength[1] and DimLength[2] are the lengths of the non-staggered dimensions. The outermost dimension of each subcube in the sequence overlaps the following subcube by *overlap* elements.

See figure 2.9 for an illustrated example. As shown, the sequence of new ALPScubes are distributed across the length of the process cube in the 0th dimension.

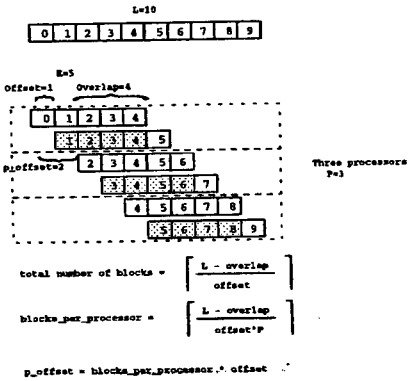


Figure 2.8: Consecutive overlapping distribution

Parameters:

- *totalCube*: ALPScube to be redistributed and split.
- *newType*: string specifying the new ALPScube type of the new ALPScube.
- *offset*: integer specifying distance between the first elements in the outermost dimension of consecutive new subcubes.
- *overlap*: integer specifying the number of overlapping elements in the outermost dimension of consecutive subcubes.
- *duplicate*:

  0 if subcubes on the same processor are to share data in memory. Data elements that are shared by two or more subcubes utilize a single copy of the data in memory. Overwriting such elements in one subcube will affect all subcubes that share data.

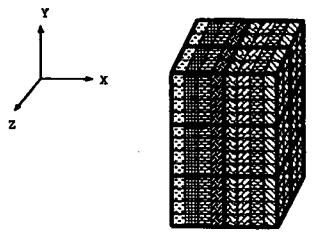
  1 if subcubes are to have separate copies of shared data elements. This option consumes more memory space but allows the data elements in a subcube to be overwritten without affecting other subcubes.

- *wraparound*:

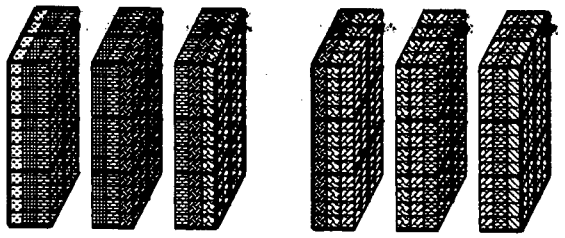
0 denotes no wraparound of outermost dimension.

-1 causes the outermost dimension to wrap around, such that the last subcube's outermost dimension begins with the last element in the outermost dimension.

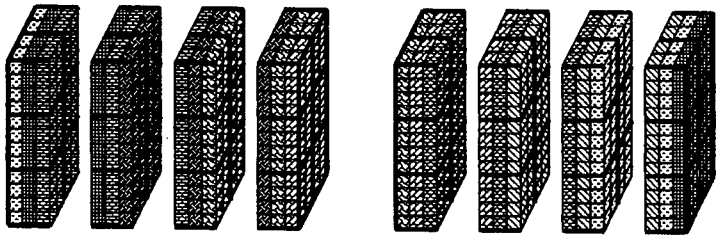
Any value greater than zero causes the outermost dimension to wrap around by the value specified.



(a) Initial Data Cube



(b) Offset=1, overlap=2, no wraparound



(c) Offset=1, overlap=2, with wraparound

Figure 2.9: Illustration of SplitStaggerCube

Return Values:

The function returns a structure of type `ALPScubeList` which contains three elements:

- **cube**: pointer to NULL-terminated array of ALPScubes (even if there is only one ALPScube in the array).
- **numCubes**: integer specifying number of ALPScubes in array.
- **comm**: MPI communicator of original ALPScube.

2.10.2 Round-robin distribution of subcubes

ALPScubeList RRSplitStaggeredCube(ALPScube *totalCube*, **char** **newType*, **int** *offset*, **int** *overlap*, **int** *wraparound*)

This function works identical to **SplitStaggeredCube** except that the subcubes are distributed in round-robin fashion across the processors, rather than consecutively. See figure 2.10. This function lacks the *duplicate* parameter since it is not applicable to this manner of distribution.

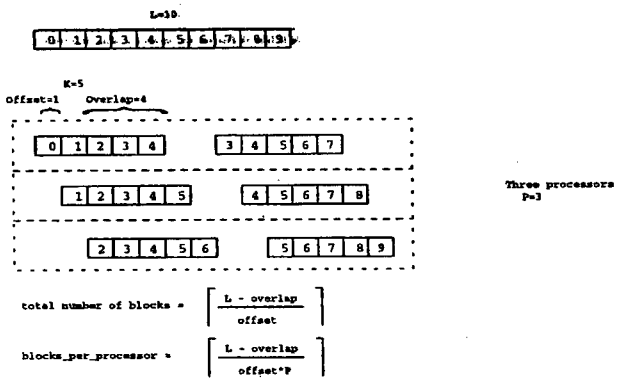


Figure 2.10: Round-robin overlapping distribution

Parameters:

- *totalCube*: ALPScube to be redistributed and split.
- *newType*: string specifying the new ALPScube type of the new ALPScube.
- *offset*: integer specifying distance between the first elements in the outermost dimension of consecutive new subcubes.
- *overlap*: integer specifying the number of overlapping elements in the outermost dimension of consecutive subcubes.

- *wraparound*:

  0 denotes no wraparound of outermost dimension.

  -1 causes the outermost dimension to wrap around, such that the last subcube's outermost dimension begins with the last element in the outermost dimension.

  Any value greater than zero causes the outermost dimension to wrap around by the value specified.

Return Values:

The function returns a structure of type `ALPScubeList` which contains three elements:

- **cube**: pointer to NULL-terminated array of ALPScubes (even if there is only one ALPScube in the array).
- **numCubes**: integer specifying number of ALPScubes in array.
- **comm**: MPI communicator of original ALPScube.

2.10.3 overlapping distribution of data

ALPScube TransStaggeredCube(ALPScube *totalCube*, **char** **newType*, **int** **newBS*, **int** *overlap*, **int** *wraparound*)

This function redistributes the data in the same fashion as **RRSplitStaggered-Cube**, except that the result is contained in a single distributed datacube rather than split into subcubes. This function provides the data duplication and redistribution functionality for the **SplitStaggeredCube** and **RRSplitStaggered-Cube** functions. Note that this function accepts an array of new blocksizes for all dimensions: the blocksize for the oth dimension is in fact the *offset* parameter. The other two dimensions are distributed in the same manner as for **TransCube**. See figure 2.11 for an illustration in the single dimension.

Parameters:

- *totalCube*: ALPScube to be redistributed.
- *newType*: string specifying the new ALPScube type of the new ALPScube.
- *newBS*: pointer to array of three integers specifying new blocksizes. *newBS[0]* specifies the offset for the 0th dimension. in the outermost dimension of consecutive new subcubes.
- *overlap*: integer specifying the number of overlapping elements in the outermost dimension of consecutive subcubes.
- *wraparound*:

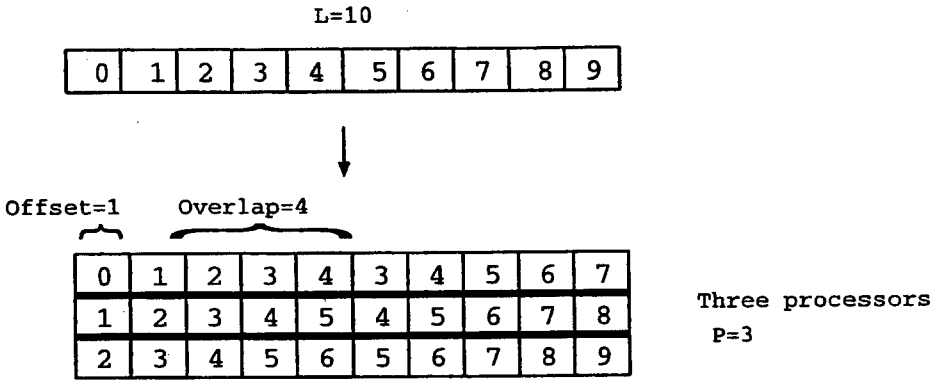


Figure 2.11: Pattern of overlapping distribution

0 denotes no wraparound of outermost dimension.

Any value greater than zero causes the outermost dimension to wrap around by the value specified.

Return Values:

The function returns a new ALPScube.

2.11 Reorganize Cube Data

ALPScube *ReCube(ALPScube **cubeList*, int *dim*)

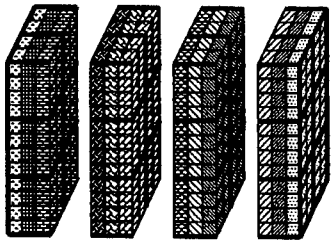
ReCube reorganizes a list of ALPScubes as follows: First it does the equivalent of **SplitCube** on each input cube, splitting it along dimension *dim* into *DimLength* pieces, where *DimLength* is the length of dimension *dim*. Each piece is effectively a two-dimensional ALPScube, or a "plane."

The firstmost planes resulting from all the **SplitCube** operations are then joined together along dimension *dim* to form the first output ALPScube; next, the secondmost plane from each **SplitCube** operation is joined together, and so forth. In this manner a list of ALPScubes are created and returned as output. Refer to the illustrated example in figure 2.12.

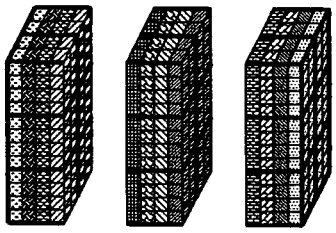
- *cubeList*: pointer to array of ALPScubes.
- *dim*: dimension along which to split and join cubes.

Return Values:

The function returns a NULL-terminated array of ALPScubes (even if there is only one ALPScube in the array).



(a) Initial Data Cubes



(b) Result of ReCube

Figure 2.12: ReCube

2.12 Duplicate ALPScube

ALPScube CopyCube(ALPScube *_origCube_)

This function creates a duplicate of the specified ALPScube.

Parameters:

- *origCube*: ALPScube to be duplicated.

Return Values:

The function returns the new ALPScube.

Chapter 3

ALPScube Matlab functions

3.1 Introduction

This chapter describes a collection of Matlab functions for reading, writing, and creating ALPS data cubes.

3.2 Read an ALPScube pdc file into Matlab

[*cube*, *cdef_str*, *cond*] = **mat_readcube**(*file*)

In order to return the results of ALPS-based applications on parallel machines to a uniprocessor workstation for verification, mat_readcube.m allows the user to read a .pdc file generated on a multiprocessor platform and create the appropriate parallel data cube in MATLAB. The use of this function has allowed many results on the Intel Paragon and IBM SP2 to be verified using the original MATLAB prototypes.

A key feature of mat_readcube.m is that it performs data permutation on the data contained in the .pdc file such that it returns to "canonical form" (orientation = [0 1 2]). When data are read into MATLAB, they are permuted into this form so that a single MATLAB prototype can operate on data from any orientation on the parallel machine.

Input Parameters:

- it file: a string specifying the file name. The '.pdc' suffix will automatically be added.

Output Parameters:

- *cube*: the 3D data cube.

- *cdef_str*: a string containing the cube type definition of the cube. It must refer to a valid entry in the parallel cube definition file (e.g. stap.fmt).
- *cond*: 0 on successful completion, 1 on failure.

3.3 Write an ALPScube pdc file from Matlab

cond = **mat_writecube**(*cube*, *file*, *cdef_str*)

The counterpart to mat_readcube.m, mat_writecube.m allows a three-dimensional data array in MATLAB to be written as a .pdc file for transfer to a parallel platform. The ALPStype must be specified to ensure the correct orientation of the data on the parallel machine.

Input Parameters:

- it file: a string specifying the file name. The '.pdc' suffix will automatically be added.
- *cube*: the 3D data cube.
- *cdef_str*: a string containing the cube type definition of the cube. It must refer to a valid entry in the parallel cube definition file (e.g. stap.fmt).

Output Parameters:

- *cond*: 0 on successful completion, 1 on failure.

3.3.1 MATLAB Canonical Orientation

The canonical orientation for ALPS datacubes in MATLAB is [0 1 2] (Range_Chan_Pri). When .pdc files are read using mat_readcube the resultant data matrix is transposed to this orientation. This provides the convenience that a single matlab routine can process data from any .pdc file, regardless of orientation. When mat_writecube is called, the orientation for the .pdc file is specified in the cube definition, resulting in a different orientation for ALPS parallel routines. Should one want to defeat the convenience of mat_readcube reading all files into canonical form, one may use mat_readtrue, which does not perform the permutation, and thus reads a non-transposed image of what is in a .pdc file.

3.4 Display contents of ALPScube

mat_printcube(*cube*, *index_mode*, *wait*)

It is desirable to have a standard output format for the contents of an ALPScube. In MATLAB, mat_printcube.m provides this output. It supports either the C (0 to (n-1)) or MATLAB (1 to n) numbering conventions.

Input Parameters:

- *cube*: The data cube to be printed.
- *index_mode*: boolean argument determining indexing mode used in output. 0 for C indexing (dims start at 0), 1 for actual MATLAB indexing (dims start at 1).
- *wait*: A boolean argument determining whether to pause between elements. 0 for no pause, 1 for pause.

cpr(*num*, *pre_str*)

rpr(*num*, *pre_str*)

ipr(*num*, *pre_str*)

Pretty-printing functions for complex, real, and integer numbers are handled by cpr.m, rpr.m, and ipr.m, respectively. They also form the basis for mat_printcube.m.

Input Parameters:

- *num*: The number to be printed.
- *pre_str*: A string to precede the value of the number. (" will print no string).

Output Parameters:

- None.

3.5 Create ALPScube with data entries that identify coordinates of each entry

cond = **mat_labelcube**(*file*, *cdef_str*, *dim*)

This function creates a three-dimensional parallel datacube of specified size with the following property: The value at each element is an encoded representation of its three-dimensional global index. Six digits are used as a triplet of two-digit indices, one for each of the three dimensions. For example, in a Range_Chan_Pri cube (of type real), the element at Range 12, Channel 4, and PRI 7 would contain the value 0.120407. This method of created labeled data is particularly useful in examining data flow in communication algorithms, as the original location of data can be deduced from its value. As two-digit fields are used to encode the indices, the maximum extent of any dimension is 99. This should be sufficient to produce datacubes for algorithmic development and verification.

Input Parameters:

- it file: a string specifying the file name. The '.pdc' suffix will automatically be added.
- *cdef_str*: a string containing the cube type definition of the cube. It must refer to a valid entry in the parallel cube definition file (e.g. stap.fmt).
- *dim*: an array of three integers specifying the dimension lengths of the global three dimensional data matrix.

Output Parameters:

- *cond*: 0 on successful completion, 1 on failure.

3.6 Create ALPScube with random data entries

cond = **mat_randcube**(*file*, *cdef_str*, *dim*)

As a means of testing both computational and communication modules in ALPS, it is important to have a means of producing randomized datasets. The function mat_randcube.m creates a parallel datacube of arbitrary size containing randomized data of the type appropriate to the cube (real, complex, integer, etc.).

Input Parameters:

- it file: a string specifying the file name. The '.pdc' suffix will automatically be added.
- *cdef_str*: a string containing the cube type definition of the cube. It must refer to a valid entry in the parallel cube definition file (e.g. stap.fmt).
- *dim*: an array of three integers specifying the dimension lengths of the global three dimensional data matrix.

Output Parameters:

- *cond*: 0 on successful completion, 1 on failure.

3.7 Retrieve information about an ALPScube type definition

dt_str = **lookup_def**(*def_str*, *attrib*)

In order to manage the variety of ALPStypes defined in the libraries, a central repository of information is kept. This specifies the data type, cube orientation,

and other information about the ALPStype. Many ALPS modules (both MATLAB and parallel C) need to consult this repository. The MATLAB function lookup_def.m is provided for this purpose.

Input Parameters:

- *def_str*: A string containing the name of the cube type definition.
- *attrib*: A string containing the requested attribute. It must be one of the following:
  1. **MPI_Datatype** (type of data stored)
  2. **permute** (permutation from standard form)
  3. **desc** (description of datatype)
  4. **local** (boolean; 1 if vector is local)

Output Parameters:

- *dt_str*: A string containing the datatype associated with the cube definition *def_str*. If the cube definition cannot be found in the file, a null string is returned.

cpr(*num*, *pre_str*)

rpr(*num*, *pre_str*)

ipr(*num*, *pre_str*)

Pretty-printing functions for complex, real, and integer numbers are handled by cpr.m, rpr.m, and ipr.m, respectively. They also form the basis for mat_printcube.m.

Input Parameters:

- *num*: The number to be printed.
- *pre_str*: A string to precede the value of the number. (" will print no string).

Output Parameters:

- None.

Chapter 4

Installation and Configuration

4.1 Obtaining the Software

The source code can be obtained at the web address:

`http://www.ee.cornell.edu/~adamb/ALPScomm.tar.gz`

This file can be downloaded with the aid of a web browser. Simply point your browser to the address shown and download the file to your hard drive.

The file must first be decompressed using the **gunzip** utility:

```
gunzip ALPScomm.tar.gz
```

This results in a new file being created called **ALPScomm.tar**. Next, use the **tar** command:

```
tar cvf ALPScomm.tar
```

This command will create a new subdirectory named **ALPScomm** with all the relevant subdirectories and files within.

4.2 Creating the library files

The first step in installing the ALPS communication libraries is to confirm the site-specific paths for running **imake** in the **site.def** file stored in the directory **ALPScomm/config**. Several current examples have been provided, including:

- site.def.CTCSP2 (IBM SP2 at the Cornell Theory Center)
- site.def.RomeParagon (Intel Paragon at the USAF Rome Laboratory)

Once the default file **site.def** is correct, follow the procedures outlined below.

4.2.1 Compilation on the SP2

To compile the on the IBM SP2, use the following imake command:

```
imake -I<CONFIGDIR> -DTOPDIR=<TOPDIR> -DSPXMIES -DCURDIR=.
```

where <CONFIGDIR> is the full pathname of the **ALPScomm/config** directory and <TOPDIR> is the full pathname to the **ALPScomm** subdirectory. (With the current directory being the one in which the **ALPScomm** subdirectory was installed, use the unix command **pwd** to determine the full pathname of the current directory.)

Next, type the following command:

```
make World
```

This should produce both the **libcube.a** and **libcomm.a** libraries in the directory **ALPScomm/lib**, as well as a group of test programs in the **ALPScomm/demo** directory.

4.2.2 Compilation on the Intel Paragon

Similarly, to compile on the Intel Paragon, the imake command is:

```
imake -I<CONFIGDIR> -DTOPDIR=topdir -DPARAGON -DCURDIR=.
```

again, where <CONFIGDIR> is the full pathname to the **ALPScomm/config** directory and <TOPDIR> is the path to the **ALPScomm** directory, followed by the command:

```
make World
```

This should produce both the **libcube.a** and **libcomm.a** libraries in the directory **ALPScomm/lib**, as well as a group of test programs in the **ALPScomm/demo** directory.

4.3 Setting the Environment

When running the ALPS communication libraries, a special environment variable must be set to indicate where the ALPStype information is stored, as it is checked during runtime. This environment variable is **CUBEDEFINITIONS**, and it would typically be set in a .cshrc file as follows:

```
setenv CUBEDEFINITIONS <DFPATH>
```

where <DFPATH> is the full pathname to the **ALPScomm/dataformat** directory in the ALPScomm distribution.

The ALPS communication libraries should be installable on any message-passing parallel platform that has **imake**, **make**, a parallel compiler, and a current MPI distribution. The critical installation differences should be confined to the **site.def** configuration file.

4.4 Writing C programs

In your C files, you must include the header file `alpscube.h`. The file is located in the `ALPScomm/include` subdirectory.

The very first function call in your `main(int argc, char argv)` routine should be `Initialize(argc, argv)`. Also, before the program exits, the `Finalize()` function should be called.

When compiling, you must include the path of the above include directory as a compiler option.

When linking, include the path of the library directory `ALPScomm/lib`, and also the link options `-lcomm` and `-lcube`.

*U.S. GOVERNMENT PRINTING OFFICE: 2000-510-079-10001

DISTRIBUTION LIST

addresses	number of copies
RALPH KOHLER AFRL/IFTC 26 ELECTRONIC PKWY ROME NY 13441-4514	2
CORNELL UNIVERSITY SCHOOL OF ELECTRICAL ENGINEERING 335 FRANK H.T. RHODES HALL ITHACA NY 14853-3801	1
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
ATTN: SMDC IM PL US ARMY SPACE & MISSILE DEF CMD P.O. BOX 1500 HUNTSVILLE AL 35807-3801	1

TECHNICAL LIBRARY D0274(PL-TS) 1
SPAWARSYSCEN
53560 HULL ST.
SAN DIEGO CA 92152-5001

COMMANDER, CODE 4TL000D 1
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE CA 93555-6100

CDR, US ARMY AVIATION & MISSILE CMD 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

REPORT LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

USAF/AIR FORCE RESEARCH LABORATORY 1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.