RL-TR-94-77
Final Technical Report
May 1994

PARALLEL SOFTWARE BENCHMARKS FOR HIGH PERFORMANCE BMC3/IS SYSTEMS

Syracuse University

Salim Hariri, Geoffrey C. Fox, Balaji Thiagarajan,
Manish Parashar

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

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

RL-TR-94-77 has been reviewed and is approved for publication.

APPROVED: Paul M. Engelhart

PAUL M. ENGELHART
Project Engineer

FOR THE COMMANDER John A. Graniero

JOHN A. GRANIERO
Chief Scientist for C3

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3CB) Griffiss AFB NY 13441. 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.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1994	Final Apr 92 - May 93

4. TITLE AND SUBTITLE
PARALLEL SOFTWARE BENCHMARKS FOR HIGH PERFORMANCE BMC3/IS SYSTEMS

5. FUNDING NUMBERS
C - F30602-92-C-0063
PE - 63728F
PR - 2527
TA - 03
WU - P9

6. AUTHOR(S)
Salim Hariri, Geoffrey C. Fox, Balaji Thiagarajan, Manish Parashar

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Syracuse University
Northeast Parallel Architectures Center
111 College Place
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Laboratory (C3CB)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-94-77

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Paul M. Engelhart/C3CB/(315) 330-4477

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
This final technical report summarizes research accomplished under the Expert Science and Engineering (ES&E) program by the Northeast Parallel Architectures Center at Syracuse University. This 13-month research endeavor, entitled "Parallel Software Benchmarks for BMC3/IS Systems," examined issues in developing parallel software and tools that can be used in a parallel/distributed computing environment. These issues were evaluated via the development and application of benchmarks over a wide range of applications. Moreover, a hierarchical set of tool criteria to evaluate and benchmark different tools for parallel/distributed computing was also developed. This report gives a description of the process, benchmarks, criteria and a multi-target tracker implementation using various parallel software tools.

14. SUBJECT TERMS
Parallel Processing, High Performance Computers, Software Benchmarks, Parallel Software Development

15. NUMBER OF PAGES
124

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

Executive Summary

We believe that the future of parallel computing lies in the integration of the plethora of architectures into a single Heterogeneous High Performance Computing (HHPC) environment that allows them to cooperate in solving complex problems. Such an environment can capitalize on existing architectures and on current advances in computing, networking and communication technology to provide efficient, cost-effective, scalable, high-performance distributed computing. Software development in any parallel/distributed environment is a non-trivial process and requires a thorough understanding of the application and architecture. This is apparent from the fact that applications are currently able to achieve only a fraction of peak available performance. The problem further intensifies as systems evolve into HHPC environments. This calls for a need to develop a software development process and tools that can be used in parallel/distributed computing and a systematic approach to evaluate and benchmark these tools for a wide range of applications. The research carried out in this project has been motivated by these requirements. The results of our research can be summarized as follows:

1. A software development process for parallel/distributed software development environment has been identified.

2. The Concurrent Multi-Target Tracking software has been implemented using four software tools. These tools include Express, Parallel Virtual Machine (PVM), Portable Instrumented Communication Library (PICL) and Portable Programming for Parallel Processors (p4).

3. Performance benchmarks of the Concurrent Multi-Target Tracker on the Intel iPSC(860), nCUBE, cluster of SUN and IBM workstations have been developed.

4. We developed a hierarchical set of tool criteria to evaluate and benchmark different tools used for Parallel/Distributed Computing. The set of tool criteria has the following levels:

   - **Level 0:** Hardware/Software requirements
   - **Level 1:** Tool Capability
   - **Level 2:** User Capability
   - **Level 3:** Software Development Capability

5. An approach to evaluate and select a tool for parallel/distributed software development is proposed. The approach is validated using the Multi-Target Tracker as a running example.

Contents

1 A Model for Software Development in a Heterogeneous High Performance Computing Environment 1

1.1 Introduction . . . 1

1.2 A Model for HHPC Software Development . . . 4

1.3 Parallel Modeling of Stock Option Pricing . . . 4

1.4 Model Inputs . . . 4

1.5 Application Analysis Stage . . . 6

1.6 Application Development Stage . . . 9

1.6.1 Algorithm Development Module . . . 9

1.6.2 System Level Mapping Module . . . 10

1.6.3 Machine Level Mapping Module . . . 11

1.6.4 Implementation/Coding Module . . . 11

1.6.5 Design Evaluator Module . . . 12

1.7 Compile-Time & Run-Time Stage . . . 12

1.8 Evaluation Stage . . . 12

1.9 Maintenance/Evolution Stage . . . 13

2 Tracker Description 14

2.1 Introduction . . . 14

2.2 Description of Concurrent Multi-Target Tracking Scenario . . . 17

2.3 Single Target Tracking . . . 17

2.3.1 Track System Initialization 17
2.3.2 Focal Plane Tracking 18
2.4 Multiple Target Tracking/3D Tracking 20
2.5 Concurrent Aspects of Multi-Target Tracker 21
2.6 Initial Implementation of the Tracker 22
2.6.1 Communication Primitives in the Concurrent tracker 23
2.6.2 Standard I/O in the Concurrent Tracker 25
2.6.3 Node Environment Information 25
2.6.4 Mapping a Decomposition Topology to the Concurrent Tracker 28

3 Tracker Implementation **29**

3.1 Porting the tracker on different tools 29
3.2 Performance of Concurrent Tracker 34
3.3 Express : Parallel Processing Toolkit 38
3.3.1 Express Implementation of Multi-Target Tracker 39
3.3.2 Performance of the Multi-Target Tracker implemented using Express 39
3.4 PVM (Parallel Virtual Machine) 41
3.4.1 PVM Implementation of the Multi-Target Tracker 43
3.4.2 Performance of the Concurrent Tracker implemented using PVM 43
3.5 PICL : Portable Instrumented Communication Library 48
3.5.1 PICL Implementation of Multi-Target Tracker 49
3.5.2 Performance of the Concurrent Tracker implemented using PICL 49
3.6 p4 : Portable Programs for Parallel Processors 51
3.6.1 Implementation of Multi-Target Tracker using p4 53
3.6.2 Performance of the Concurrent Tracker implemented using p4 54
3.7 Performance Comparison 56

4 Tool Evaluation Methodology **58**

4.1 Motivation 58
4.2 Proposed Approach for Tool Evaluation 59
4.3 Tool Evaluation Criteria 61
4.3.1 Level 0: Hardware/Software Requirements 61
4.3.2 Level 1: Tool Capability 62
4.3.3 Level 2: User Capability 64
4.3.4 Level 3: Application Development Capability 67
4.4 Methodology for Tool Evaluation 69
4.4.1 Tool Evaluation Algorithm 70
4.4.2 Step-by-Step Explanation of Tool Evaluation Process 72

5 Tool Evaluation Benchmarks **74**
5.1 Mapping the HHPC software development model to the Multi-Target Tracker porting process 75
5.2 Tool Evaluation Benchmarks 79
5.2.1 Tool Evaluation Benchmarks for PVM 79
5.2.2 Tool Evaluation Benchmarks for Express 84
5.2.3 Tool Evaluation Benchmarks for PICL 88
5.2.4 Tool Evaluation Benchmarks for p4 92

6 Summary and Conclusions **99**

A Glossary **106**

List of Figures

1.1 The Heterogeneous High Performance Computing Environment (HHPC) 2

1.2 A Model for HHPC Software Development 5

1.3 Stock Option Pricing Model: Application Specifications 7

1.4 Stock Option Pricing Model: Parallelization Specifications 8

2.1 Battle Management Command Control and Communication 15

2.2 Format of Input Data for Internal Generation of Threats 18

2.3 Cros III Primitives 23

2.4 Communication and Topology Mapping calls 24

2.5 Implementation of Global Communication Routines in the Tracker 26

2.6 Cube environment Structure Templates 27

3.1 Host Program Structure Template 32

3.2 Node Program Structure Template 33

3.3 Cshift Implementation 35

3.4 Hardware Environment for Multi-Target Concurrent Tracker 37

3.5 Changes to Host Program and Node Program Templates for Express Implementation 39

3.6 Plot of Main Tracking Tasks in Express 41

3.7 Changes to Host Program and Node Program Templates for PVM Implementation 44

3.8 Plot of Main Tracking Task in PVM 45

3.9 Plot of Main Tracking Task in PVM in a Heterogeneous Environment 47

LIST OF FIGURES

3.10 Changes to Host Program and Node Program Templates for PICL Implementation 50

3.11 Plot of Main Tracking Task in PICL 52

3.12 Changes to Host Program and Node Program Templates for p4 Implementation 53

3.13 Plot of Main Tracking Tasks in p4 56

4.1 Grading Scheme and Tool Hierarchy 60

4.2 Tool Algorithm and Implementation Example 71

5.1 PVM Host Makefile 77

5.2 PVM Node Makefile 78

List of Tables

1.1 Current utilization of parallel/distributed systems 3

3.1 Performance of Main Tracking Tasks on iPSC/860 using Express (8 Sites, 640 targets) 40

3.2 Speedup and Efficiency of Main Tracking Task on iPSC/860 using Express (8 Sites, 640 targets) . . . 40

3.3 Performance of Main Tracking Tasks on IBM RS/6000 using PVM (8 Sites, 640 targets) 45

3.4 Speedup and Efficency of Main Tracking Task on IBM RS/6000 (8 Sites, 640 targets) using PVM . . . 46

3.5 Performance of Main Tracking Task using PVM on Heterogeneous Nodes: SUN4 and IBM RS/6000 (8 Sites, 640 targets) 47

3.6 Multiple Tracking Task on Heterogeneous Nodes: SUN4 and IBM RS/6000 (8 Sites, 640 targets) using PVM 48

3.7 Performance of Main Tracking Task on iPSC/860 using PICL (8 Sites, 640 targets) 51

3.8 Multiple Tracking Task on iPSC/860 using PICL (8 Sites, 640 targets) 51

3.9 Performance of Main Tracking Task using p4 on IBM RS/6000 (8 Sites, 640 targets) 55

3.10 Performance of Main Tracking Task using p4 on IBM RS/6000 (8 Sites, 640 targets) 55

5.1 Example Evaluation of PVM 80

5.2 Example Evaluation of PVM (cont..) 81

5.3 Example Evaluation of PVM (cont..) 82

5.4 Example Evaluation of Express 85

5.5 Example Evaluation of Express (cont..) 86

5.6 Example Evaluation of Express (cont..) 87

5.7 Example Evaluation of PICL 89

LIST OF TABLES

5.8 Example Evaluation of PICL (cont..) 90

5.9 Example Evaluation of PICL (cont..) 91

5.10 Example Evaluation of p4 93

5.11 Example Evaluation of p4 (cont ..) 94

5.12 Example Evaluation of p4 (cont ..) 95

5.13 Process Evaluation Benchmarks for PVM, Express, p4 and PICL 98

5.14 Tool Evaluation Benchmarks for PVM, PICL, p4 and Express 98

Chapter 1

A Model for Software Development in a Heterogeneous High Performance Computing Environment

1.1 Introduction

The last few decades have seen an impressive development in every aspect of parallel computing technology; viz. processing and storage technology, interconnect technology and software technology. Advances in processing and storage technology can be characterized by advances in device and concurrency technology. Developments in device technology have resulted in faster, more powerful processors with larger storage support and increased functionality, while research in concurrency technology has explored new concurrency paradigms designed to exploit parallelism at different levels and in different ways (e.g. SIMD, shared memory MIMD (SM-MIMD), distributed memory MIMD (DM-MIMD), Dataflow, Vector, Pipelined, etc.). Advances in interconnect technology have introduced (a) high speed, reliable networks capable of providing high transfer rates (e.g. FDDI, DQDB, HIPPI, SONET, ATM, etc.), (b) new, more efficient communication protocols (e.g. NETBLT, VMTP, XTP, Ultranet, etc.) and (c) exotic interconnection topologies (e.g. FAT Tree, Hypercube, Mesh, Torus, etc.). Advances in software technology have explored new approaches to assist the user in developing parallel software and application. This has included the development of automatic parallelizing/vectorizing compilers, parallel programming languages and language extensions, parallel software development environments, etc. along with support tools such as performance analysis/monitoring tools, problem decomposition/mapping tools, parallel debugging tools, etc.

High performance computer systems today, include SIMD architectures like CM2 and DECmpp, shared memory MIMD, vector and pipelined architectures like the CRAY C90, NEC SX3, and IBM POWER/4, distributed memory MIMD machines like the Paragon XP/S and iPSC/860 from Intel, the CM-5 from TMC, and the KSR1, transputer based machines like the Parsytec GC, special purpose architectures like the BBN MP2000, etc. Each of the above architectures can be thought of as points in the state space of possible alternatives in the design of parallel computers and result from a unique set of trade-off's in system

parameters and design decisions. These design trade-off's cause specific architectures to favor certain computational models and thereby deliver maximum performance only to a specific set of applications which lend themselves to one of those computation models. Further, this narrow applicability of current architectures has prevented them from being cost-effective. As a result, although these architectures incorporate large amount of computing power, they are not general enough to efficiently support today's computation-intensive problems, that warrant multiple computational models and levels of parallelism. The U.S. Office of Science and Technology Policy's Committee on Physical, Mathematical, and Engineering Sciences has outlined a set of desired applications and their computing requirements in its report "Grand Challenges: High Performance Computing and Communication" (1991) [27]. The Grand Challenge applications include climate modeling, fluid turbulence, pollution dispersion, human genome, ocean circulation, quantum chormodynamics, semiconductor modeling, superconductor modeling, combustion systems and vision and cognition among others, and are estimated to requires Teraflops (10^{12} flops) of computing power. Tackling applications of this magnitude and diversity would require a general, cost-effective, scalable, yet powerful computing model which will be able to efficiently support its varied computational and communication requirement. It is this realization that has spurred intense research in heterogeneous computing environments [2, 1, 28, 29, 30, 27, 31, 32].

We believe that the future of parallel computing lies in the integration of the plethora of "specialized" architectures into a single Heterogeneous High Performance Computing (HHPC) environment that allows them to cooperate in solving complex problems (Figure 1.1). The HHPC environment will capitalize on

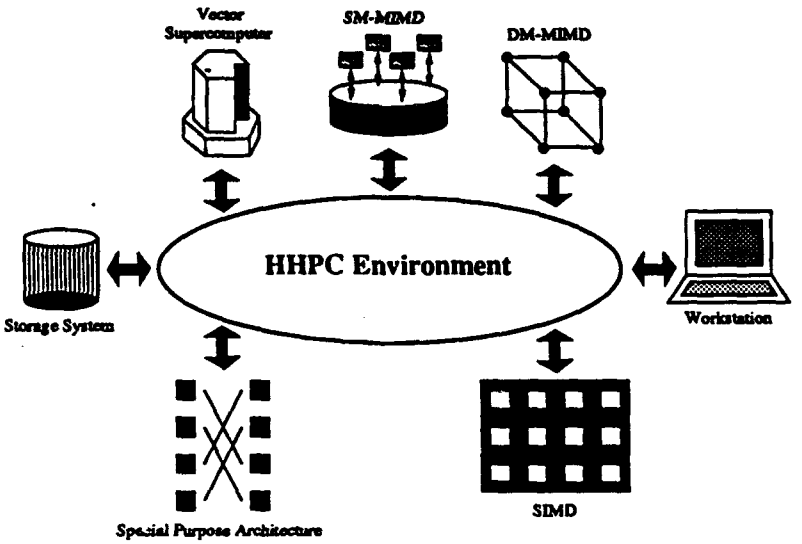


Figure 1.1: The Heterogeneous High Performance Computing Environment (HHPC)

existing architectures and on current advances in computing, networking and communication technology to provide efficient, cost-effective, scalable, high-performance distributed computing.

Software development in any Parallel/Distributed environment is a non-trivial process and requires a thorough understanding of the application and the architecture. This is apparent from the fact that, applications are currently able to achieve only a fraction of peak available performance [31, 27]. The percentage of the peak performance achieved by standard parallel benchmarks on current parallel/distributed systems

System Name	Configuration (# Processors)	Peak Speed (Gigaflops)	NAS Parallel Benchmark Efficiency (%)		
			EP (2^{28})	FFT ($256^{28} \times 128$)	CG (2×10^6)
Cray C90	16	16	50%	30%	16%
Intel iPSC/860	128	2.6	15%	20%	3%
TMC CM-200	64 K	20	12%	-	-
TMC CM-2	64 K	14	-	4%	-
TMC CM-2	16 K	3.5	-	-	3%

Note:
NAS = Numerical Aerodynamic Simulation
EP = Highly parallel Monte Carlo Simulation
FFT = 3-D Poisson PDE solver using FFT
CG = Conjugate Gradient linear equation solver for a banded system of equation

Table 1.1: Current utilization of parallel/distributed systems

is shown in Table 1.1 [27]. The software development problem further intensifies as systems evolve into HHPC environments. The HHPC environment, while increasing the computing power and design flexibility available to the user, provides increased degrees of freedom and therefore requires the developer to make a larger number of design choices. During the course of software development in an HHPC environment, the developer is required to select the optimal hardware configuration for a particular application, the best decomposition and mapping of the problem onto the selected hardware configuration, the best communication and synchronization strategy to be used, etc. Using conventional techniques, this would require extensive experimentation and data collection before these parameters can be resolved. The process is not always feasible since parallel/distributed systems are expensive resources and usually not freely available for such experimentation. Further, programming, running and data collection on most parallel/distributed systems is a tedious process and exhaustively evaluating the possible alternatives is usually not practical. Most existing evaluation tools post-process traces generated during an execution run. This implies instrumenting source code, executing the application on the actual hardware to generate trace files, post-processing these trace files to gain insight into the execution and overheads in the implementation, refining the implementation and then repeating the process. The process is repeated until all possibilities have been evaluated and the best options for the problem have been identified. Clearly, this development overhead explains the poor exploitation of existing parallel/distributed platforms.

Consequently, there is a need for a software development environment which can assist the developer in uncovering the inherent parallelism in the application, to make efficient use of the underlying computing resources and to exploit the heterogeneity in both, application and hardware. Such an environment should outline the stages involved in the software development process and incorporate tools to support the developer during each stage of application development starting from the specification and design formulation stages through the programming, mapping , distribution, scheduling phases, tuning and debugging stages upto the evaluation and maintenance stages.

1.2 A Model for HHPC Software Development

The HHPC software development model described in this chapter is defined as a set of stages which correspond to phases typically encountered in the software development process. At each stage, a set of support tools which can assist the developer are identified. The stages can be viewed as a set of filters in cascade (see Figure 1.2). The input to this system of filters is the application description and specification which is generated from the application itself (if it is a new problem) or from existing sequential code (porting of dusty decks). The final output of the model is a running application. All intermediate stages are managed within the environment, with user interaction. Feedback loops are present at some stages to enable step-wise refinement and tuning. Descriptions of models for traditional parallel computing environments spanning parts of the software development process can be found in [33, 36, 37]. The stages in the HHPC software development model are described in the following sections. To illustrate the validity of the proposed model and the application of the various stages of the model, we use the Modeling of Stock Option Pricing [39] as a running example through the discussion below.

1.3 Parallel Modeling of Stock Option Pricing

Stock options are contracts that give the holder of the contract the right to buy or sell the underlying stock at some time in the future for an agreed upon striking or exercise price. Option contracts are traded just as stocks and models that quickly and accurately predict their prices are valuable to the traders. Stock option pricing models estimate the price for an option contract based on historical market trends and current market information. The model required three classes of inputs: (1) **Market Variables** which include the current stock price, call price, exercise price and time to maturity. (2) **Model Parameters** which include the volatility of the asset (variance of the asset price over time), variance of the volatility and the correlation between asset price and volatility. These parameters cannot be be directly observed and must be estimated from historical data (using optimization techniques). (3) **User Inputs** which specify the nature of the required estimation; e.g. American/European call, constant/stochastic volatility, time of dividend payoff, and other constraints regarding acceptable accuracy and running times. A number of option pricing models have been developed using varied approaches, e.g. non-stochastic analytic models, Monte Carlo simulation models, binomial models, binomial models with forced recombination, etc. Each of these models involve a set of tradeoff's in the nature and accuracy of the estimation and suit different user requirements. In addition, these models make varied demands in terms of programming models and computing resources.

1.4 Model Inputs

The HHPC software development model presented in this chapter addresses two classes of applications:

1. **"New" Application Development:** This class of applications involves solving new problems using the resources of a HHPC environment. Developers of this class of applications have to start from scratch using a textual description of the problem.

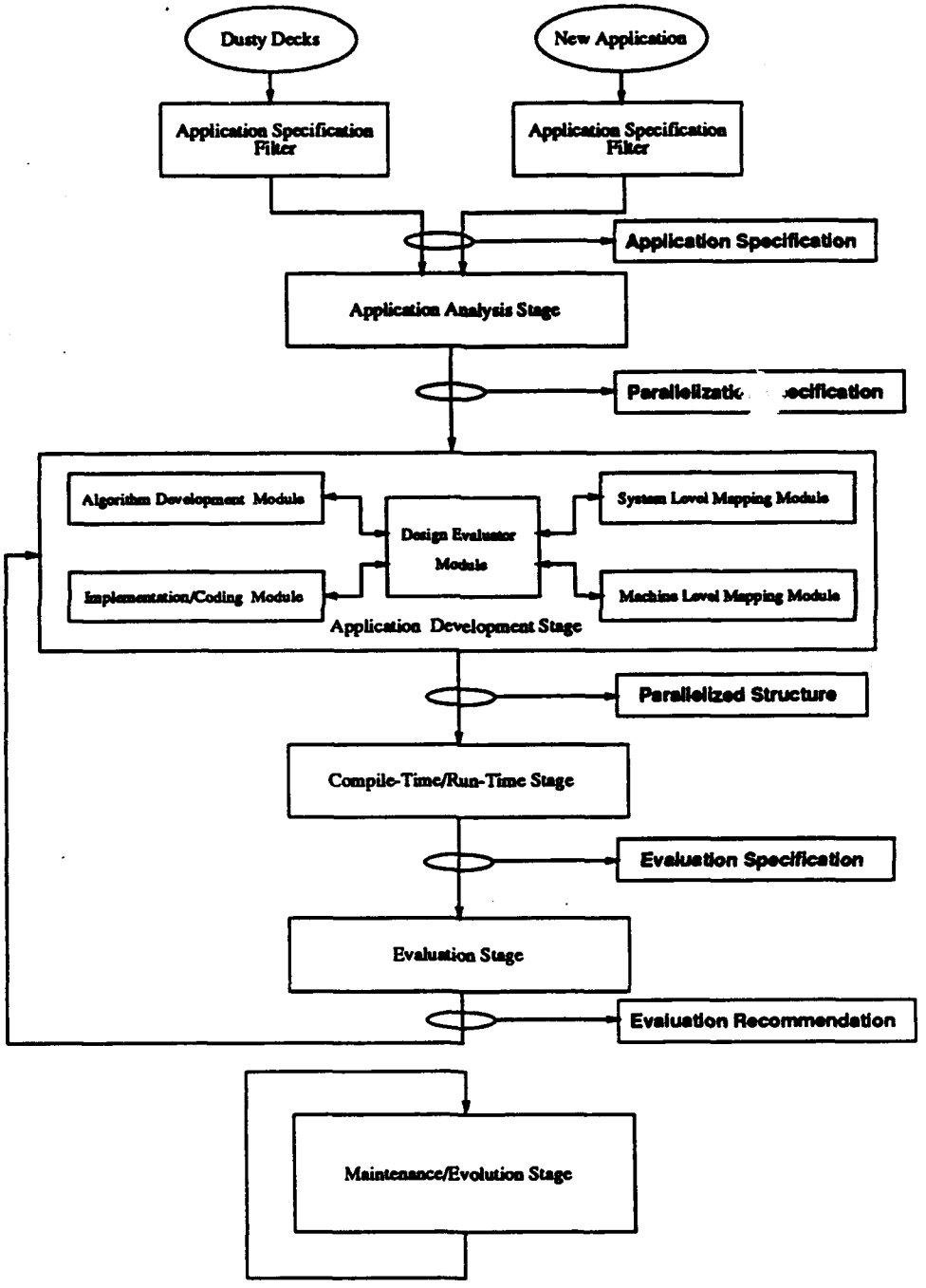


Figure 1.2: A Model for HHPC Software Development

2. **Porting of Exiting Applications (Dusty-Decks):** This class includes developers attempting to port existing codes written for single processor or closely-coupled multiprocessor systems, to a HHPC environment. Developer of this class applications start off with huge listings of (hopefully) commented source code.

The input to the HHPC software development model is an application specification in the form of a functional flow description of the application and its requirements. The functional flow description is a very high-level flow diagram of the application outlining the sequence of functions that have to be performed. Each node (termed as functional module) in the functional flow diagram is a black-box and contains information about (1) its input(s), (2) the function to be performed, (3) the desired output(s) and (4) the requirements at each node. Implementation issues like the approach or algorithm to be used to realize a function or the nature of data representation to be used, are not included in this specification. The application specification can be thought of as corresponding to the "user requirement document" in a traditional life-cycle models.

In the case of new applications, the inputs are generated from the textual description of the problem and its requirements. In the case of dusty decks code porting, the developer is required to analyze the existing source code. In either case, expert system based tools and intelligent editors, both equipped with a knowledge base to assist in analyzing the application, are required. In Figure 1.2, these tools are included in the "*Application Specification Filter*" module.

The stock price modeling application comes under the first class of applications. The application specifications based on the textual description presented in Section 1.3, is shown in Figure 1.3.

It consists of three functional modules: (1) The input module which accepts user specification, market information and historical data and generates the three classes of inputs required by the model. (2) The estimation module consists of the actual model and generates the stock option pricing estimates. (3) The output module provides a graphical display of the estimation to the user. The feedback from the output module to the input module represents tuning of the user specification based on the output displayed.

1.5 Application Analysis Stage

The first stage of the HHPC software development model is the application analysis stage. The input to this stage is the application specification as described in Section 1.4. The function of this stage is to thoroughly analyze the application with the sole objective of achieving the most efficient implementation. An attempt is made, in this stage, to uncover any parallelism inherent in the application. Functional modules which can be executed concurrently are identified and the dependencies between these modules are analyzed. In addition, the application analysis stage attempts to identify standard computational modules which can later be matched with a database of optimized templates in the application development stage (for example, nodes in the application specification performing a Fast Fourier Transform can be clustered and tagged so that they can be matched with an appropriate FFT template in the application development stage). The output of this stage is a detailed process flow graph called the "Parallelization Specification" where the nodes represent functional components and the edges represent interdependencies. Thus, the

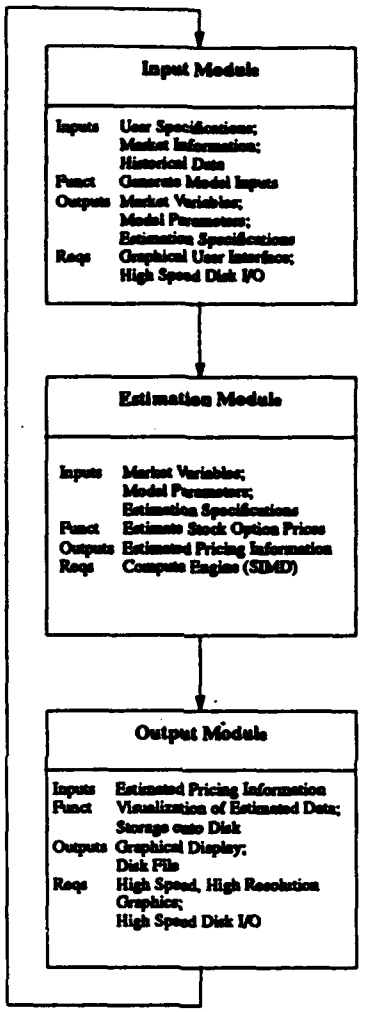


Figure 1.3: Stock Option Pricing Model: Application Specifications

problems dealt with in this stage can be summarized as: (1) module creation problem, i.e. identification of tasks which can be executed in parallel; (2) module classification problem i.e. identification of standard modules; and (3) module synchronization problem, i.e. analysis of mutual interdependencies. This stage corresponds to the "design phase" in standard software life-cycle models and its output corresponds to the "design document".

The tools which can assist the user at this stage of software development are: (1) smart editors which can interactively generate directed graph models from the application specifications; (2) intelligent tools with learning capabilities which can use the directed graphs to analyze dependencies, identify potentially parallelizable modules and attempt to classify the functional modules into standard modules; and (3) problem specific tools equipped with a database of transformations and strategies applicable to the specific problem.

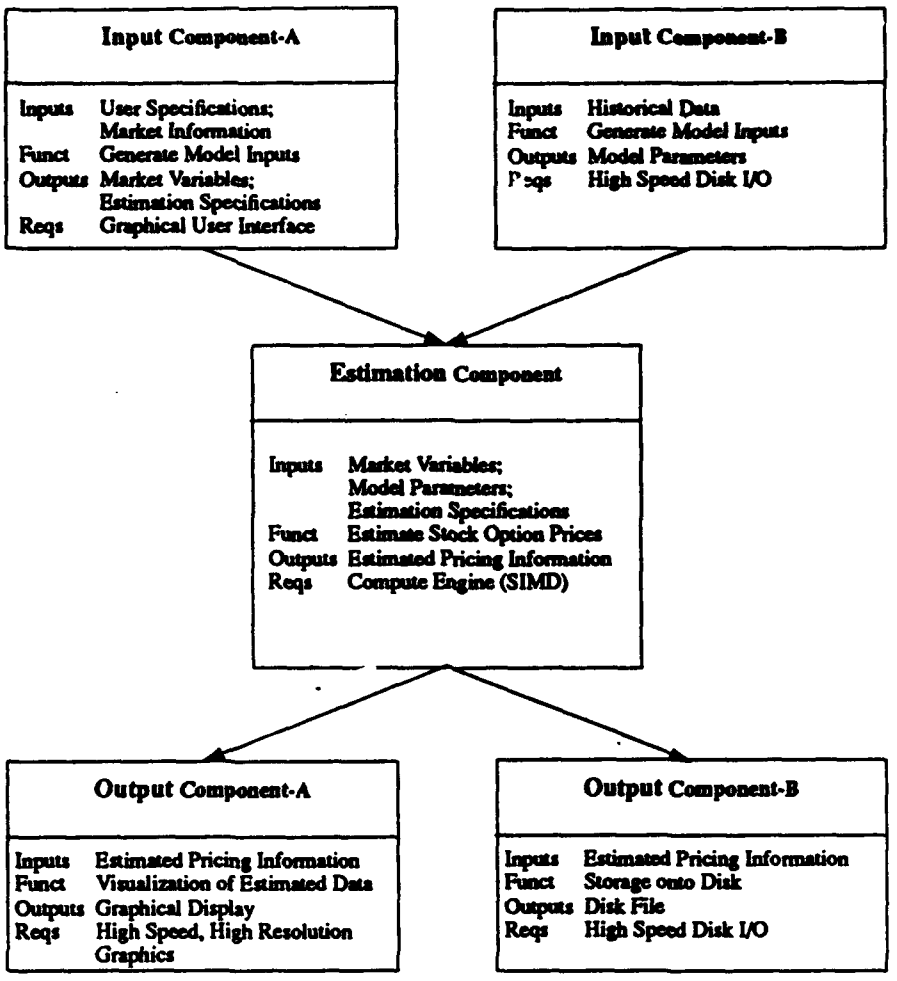


Figure 1.4: Stock Option Pricing Model: Parallelization Specifications

The parallelization specification for the running example is shown in Figure 1.4. The Input functional module is subdivided into two functional components: (1) analyzing historical data and generating model parameters; and (2) accepting market information and user inputs to generate market variables and estimation specifications. The two components can be executed concurrently. The Estimation module is identified as a standard computational module and is retained as a single functional component (to avoid getting into the details of financial modeling in this paper). The Output functional module consists of two independent functional components: (1) rendering the estimated information onto a graphical display; and (2) writing it onto disk for subsequent analysis.

1.6 Application Development Stage

The application development stage receives as its input the Parallelization Specifications and produces the Parallelized Structure which can then be compiled and executed. This stage is made up of 5 modules: (1) Algorithm Development Module; (2) System Level Mapping Module; (3) Machine Level Mapping Module; (4) Implementation/Coding Module; and (5) Design Evaluator Module. It should be noted, however, that these modules are not executed in any fixed sequence or a fixed number of times. There exists instead, a feedback system from each module to the other modules through the design evaluator module. This allows the development as well as the tuning to proceed in an iterative manner using step-wise refinement. A typical sequence of events in the application development stage can be outlined as follows:

- The Algorithm Development Module uses an initial system level mapping (possibly specified via user directives) to select appropriate algorithms for the functional components.
- The Algorithm Development Module then uses the services of the Design Evaluator Module to evaluate applicable algorithms and to tune the selection of the algorithmic implementations.
- The System Level Mapping Module uses feedback provided by the Design Evaluator Module and the Algorithm Development Module to tune the initial mapping.
- The Machine Level Mapping Module selects an appropriate machine level distribution and mapping for the particular algorithmic implementation and system level mapping. Once again, feedback from the Design Evaluator Module is used to select between alternate mappings.
- This process of step-wise refinement and tuning is continued until some termination criterion is met (e.g. until some acceptable performance is achieved or up to a maximum time limit).
- The selected algorithm, system level mapping and machine level mapping are realized by the Implementation/Coding Module which generates the parallelized structure.

1.6.1 Algorithm Development Module

The function of the algorithm development module is to assist the developer in identifying functional components in the parallelization specification and selecting appropriate algorithmic implementations. The input information to this module includes: (1) the classification and requirements of the components specified in the parallelization specification; (2) hardware configuration information; and (3) mapping information generated by the system level mapping module. It then uses this information to select the best algorithmic implementation and the corresponding implementation template from its database. It also analyzes the requirements of the selected algorithm (e.g. communication requirements, synchronization requirements, storage requirements, etc.). The algorithm development module uses the services of the design evaluator module to select between possible algorithmic implementations. Tools needed during this phase include an intelligent algorithm development environment (ADE) equipped with a database of optimized templates for different algorithmic implementations, an evaluation of the requirements of these templates and an estimation of their performance on different platforms.

The algorithm chosen to implement the Estimation Component of the stock option pricing model (shown in Figure 1.4), depends on the nature of the estimation (constant/stochastic volatility, American/European calls/puts, dividend payoff time, etc) to be performed and the accuracy/time constraints. For example, models based on Monte Carlo simulation provide high accuracy. However, these models are computationally intensive and slow and thereby cannot be used in real-time systems. Further they are not suitable for American calls/puts when early dividend payoff is possible. Binomial models are less accurate than Monte Carlo models but are more tractable and can handle early exercise. Models using constant volatility (as opposed to treating volatility as a stochastic process) lack accuracy but are simplistic and easy to compute. Modeling American calls where in the option can be exercised anytime during the life of the contract (as opposed to European calls which can only be exercised at maturity) is more involved and requires sophisticated and computationally efficient model (e.g. binomial approximation with forced recombination).

The algorithmic implementations of the input and output functional components must be capable of handling terminal and disk I/O at rates specified by the time constraint parameters. Further, the output display must provide all information required by the user.

For an illustration of the operation of this module for a predefined mapping, consider a functional component which requires the solution of a system of linear equations. If it is mapped onto an SIMD architecture, a direct parallelization of the Gauss-Jordan algorithm is applicable. However, if the target machine has a MIMD architecture, the blocked Gauss-Seidel algorithm will be more efficient.

1.6.2 System Level Mapping Module

The function of the system level mapping module is to use the information provided by the algorithm development module to appropriately map the functional components of the application to the appropriate computing elements of the HHPC environment. The objective is to map each functional component to the computing element that maximizes the performance of the application. Some data and load distribution issues may have to be resolved in this module. In addition, this module may also cluster functional component nodes specified in the parallelization specifications to obtain a better mapping. The system level mapping module uses feedback from the evaluation module to select between different mapping candidates.

System level mapping can be accomplished in an interactive mapping environment equipped with intelligent tools for analyzing the requirements of the functional components, and a knowledge base consisting of analytic benchmarks for the different computing elements and interconnection media in the HHPC environment. The tools use this information to interactively select an appropriate mapping of algorithmic implementations to the computing elements.

The algorithms for stock option pricing have been efficiently implemented on architectures like the CM2 and the DECmpp-12000 [39]. Thus, an appropriate mapping for the estimation functional component in the parallelization specification in Figure 1.4 is an SIMD architecture. The input and output interfaces (Input/Output Component-A) require graphics capability with support for high speed rendering (output display) and must be mapped to an appropriate graphics stations. Finally, Input/Output Component-B

requires high speed disk I/O and must be mapped to an I/O server with such capabilities.

1.6.3 Machine Level Mapping Module

The machine level mapping module performs the mapping of the functional component(s) onto the processor(s) of the computing elements. This stage resolves issues like data partitioning, load distribution, control distribution, etc. and makes transformations specific to that computing element. It uses the feedback from the design evaluator module to select between possible alternatives. Machine level mapping can be accomplished in an interactive mapping environment similar to that described for the system level mapping module, but equipped with information pertaining individual computing elements of a specific computer architecture.

The performance of the stock option pricing models is very sensitive to the layout of data onto the processing elements. The optimal layout is dictated by the input parameters (e.g. time of dividend payoff, terminal time, etc.) and by the specification of the architecture onto which the component is mapped. For example, in the binomial model, the continuous time processes for stock price and volatility are represented as discrete up/down movements forming a binary lattice. Such lattice is generally implemented as asymmetric arrays which are distributed onto the processing elements. It has been found that the default mapping of these arrays (i.e. in two dimensions) on architectures like the DECmpp-12000, lead to poor load balancing and performance, specially for extreme values of the dividend payoff time [40]. Further the performance in case of such a mapping, is very sensitive to this value and has to be modified for each set of inputs. Hence, in this case it is favorable to explicitly map them as one dimensional arrays. This is done by the machine level mapping module. As another example of the mapping performed by this module, consider the case of a functional component performing linear algebra which is allocated to a hypercube architecture. The function of the machine level mapping module would be to decide whether to distribute the matrix in a block or cyclic fashion and whether to perform this distribution in a column or row major fashion.

1.6.4 Implementation/Coding Module

The function of the implementation/coding module is to handle all code generation and perform the code filling of selected templates so as to produce parallel code which can then be compiled and executed on the target computer architecture. This module incorporates all machine specific codes, handles the introduction of calls to communication and synchronization routines and takes care of the distribution of data among the processing elements. It also handles any input/output redirection that may be required. Machine specific transformations and calls to optimized machine specific libraries are inserted by this module.

With regard to the pricing model application, the implementation/coding module is responsible for introducing the machine specific communication routines. For example, the binary estimation model makes use of the "end-of-shift" function for its nearest-neighbor communication. The corresponding function call in C^* (CM2) or MPL (DECmpp-12000) are introduced by this module. A possible machine specific optimization that can be introduced by this module is to reduce communication by making use of in-processor arrays. This optimization can improve performance by about two orders of magnitude [39].

1.6.5 Design Evaluator Module

The design evaluator module is a critical component of the application development stage. Its function is to assist the developer in evaluating different options (e.g. algorithms, implementations, system level mappings, machine level mappings including data partitionings, etc.), available to each of the other modules, and identifying the option that provides the best performance. It receives information about the hardware configuration, the application structure, the requirements of the selected algorithms and the mappings. This input information is then used to estimate the performance of the application on the target computer. Further, it provides insight into the computation and communication costs, the existing idle times and the overheads. This information can be used by the other modules to identify regions where further refinement or tuning is required. The module can evaluate the correctness of the implementation using performance debugging as a criterion and can detect synchronization error like deadlocks. Finally, many runtime scenarios can be evaluated (e.g. system load, network contention). The keys features of this module are: (1) the ability to provide evaluations with the desired accuracy, with minimum resource requirements and within a reasonable amount of time; (2) the ability to automate the evaluation process; and (3) the ability to perform the evaluation within an integrated workstation environment without running the application on the target computers. Support applicable to this module consists primarily of performance prediction and estimation tools. Simulation approaches can also be used to achieve some of the required functionality.

1.7 Compile-Time & Run-Time Stage

The compile-time/run-time stage handles the task of executing the parallelized application generated by the development stage to produce the required output. The input to this stage is the parallelized source code (parallelized structure). The compile-time portion of this stage consists of set of cross compilers for the computing elements and tools for scheduling and allocation. These compilers have appropriate optimization capabilities and can introduce machine-specific optimizing transformation into the parallelized structure. The compile-time software also handles the loading of the executables onto appropriate computing elements. The run-time portion of this stage handles run-time functions like debugging, scheduling, dynamic load balancing, migration, irregular communications, etc. It also enables the user to (non-intrusively) instrument the code for profiling and debugging and allows checkpointing for fault-tolerance. During the execution of the application, it accepts outputs from the different computing elements and directs them for proper visualization. It intercepts error messages generated and provides proper interpretation.

1.8 Evaluation Stage

In the evaluation stage, the developer, retrospectively evaluates the design choices made during the design process and looks for ways to improve the performance. The evaluation stage performs a thorough evaluation of the execution of the entire application, detailing communication and computation times, communication and synchronization overheads and existing idle times at every execution level (application level, node level, process level, procedure level, etc.). It uses this evaluation to identify regions in the implementation where performance improvement is possible. The evaluation procedure is accurate,

non-intrusive and does not alter the execution order of the application. Further, it allows a cost-effective evaluation (in terms of time and resources) of the application for a representative inputs set as well as the effect of various run-time parameters like system load, network contention, on performance. The scalability of the application with machine and problem size is also evaluated. The key features of this stage are: (1) the ability to provide desired accuracy and granularity of evaluation while maintaining tractability and non-intrusiveness; and (2) the ability to perform the evaluation within a friendly workstation environment without requiring the actual hardware. Support applicable to the evaluation stage include different analytic tools, monitoring tools, simulation tools and prediction/estimation tools.

1.9 Maintenance/Evolution Stage

In addition to the above described stages encountered during the development and execution of HHPC applications, there is an additional stage in the life-cycle of this software which involves its maintenance and evolution. Software maintenance is an important part of the software life-cycle and is known to span around 70% of this cycle. Maintenance includes monitoring the operation of the software and ensuring that it continues to meet its specifications. It involves detecting and correcting bugs as they surface. The maintenance stage also handles modifications needed to incorporate changes in the system configuration. Software evolution deals with improving the software, adding additional functionality, incorporating new optimizations, etc. Another aspect of evolution is the development of more efficient algorithms and corresponding algorithmic templates and the incorporation of new hardware architectures. To support such a development, the maintenance/evolution stage provides tools for the rapid prototyping of hardware and software and for evaluating the new configuration and designs without having to implement them. Other support required during this stage includes tools for monitoring the performance and execution of the software, fault detection and recovery tools, system configuration and configuration evaluation tools and prototyping tools.

Chapter 2

Tracker Description

2.1 Introduction

In this section we present an overview of the Multi-Target Tracker. The concurrent multi-target tracker [3, ?] provides a non-trivial CPU and memory-intensive application (34,000 lines of code). It is one of the modules that fits into the Battle Management Command Control and Communication system. This system consists of a set of components which interact with each other by exchanging information for data processing. The important components of the overall system include an Environment Generator and Synthesizer, the Tracker System, Decision Control System and an External Graphics Communication Module, as shown in Figure 2.1.a.

The Multi-Target Tracker gets information from the Environment Generator and Synthesizer. It processes this data and generates the parameters required by the Decision Control System. The Decision Control System uses this information along with data from the Environment Generator and Synthesizer to make decisions on how to manage the existing battle management command and control scenario. This information is fed to an External Graphics Communication module. This module acts as an extension or front-end to the Decision Control Module. For example in a Missile Tracking system the front-end can serve as a visualization tool or data interpreter which shows the trajectory and position of launched missiles. The Fire Control Module performs the actions directed by the Decision Control System.

The Multi-Target Tracker shown in Figure 2.1.b is designed to provide estimation of launch vehicle parameters for individual targets/missiles in multi-target scenarios. The system deals with a mass raid scenario and is designed to process situations with varying number of targets and launch sites. The tracker receives input from the Environment Generator and Synthesizer module (see Figure 2.1.a) in terms of sensor scans and target information. The Multiple Target Tracking system has two geostationary sensors which scar specific launch sites for missiles or targets launched from the surface of earth . The launch sites are specified in terms of latitudes and longitudes. The data from these two geostationary sensors are fed to two Focal Plane Tracking modules at 5 second intervals. The Focal Plane Tracking modules process this data using

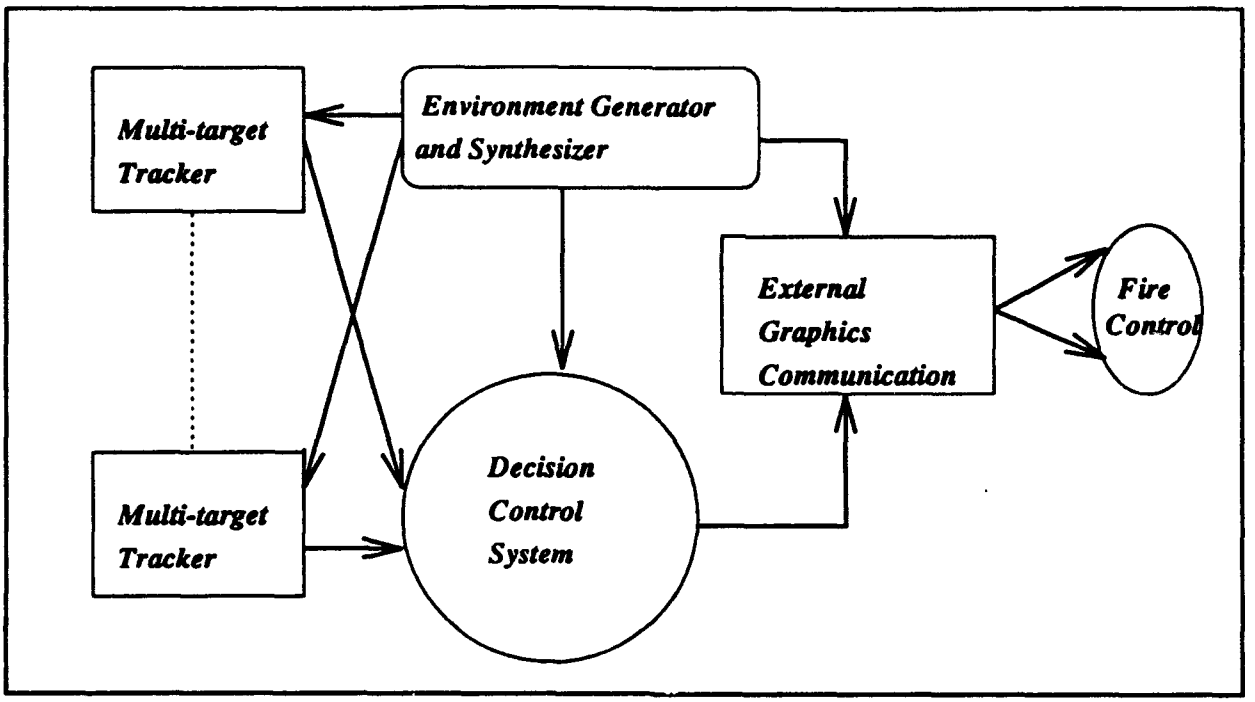


Fig 2.1.a Battle Management Command Control Communication Scenario

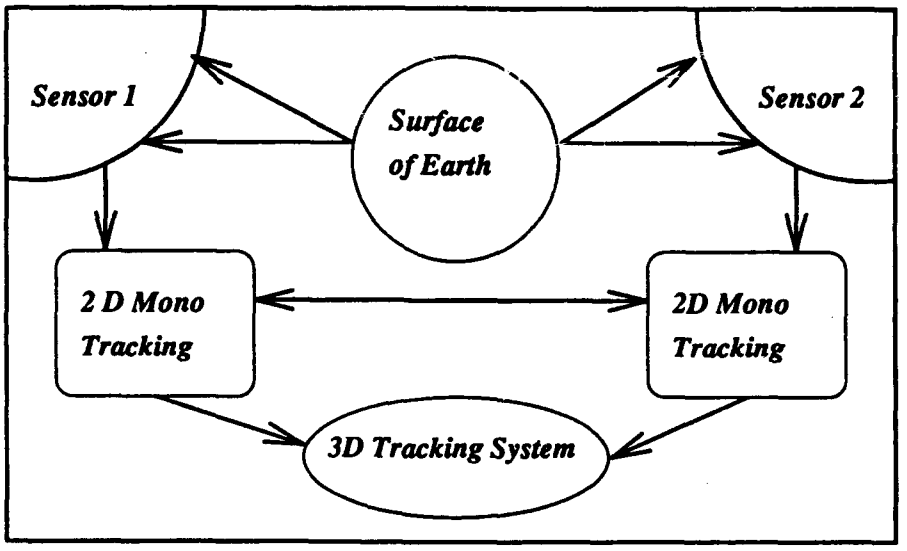


Fig 2.1.b Concurrent Multi-target Tracker Scenario

Figure 2.1: Battle Management Command Control and Communication

kinematic filtering algorithms, track pruning and prediction algorithms. The output of this module is an initial prediction of trajectories of launched missiles. This data is then fed to a 3D Tracking System which uses the data from the two Focal Plane Tracking modules to prune duplicate tracks (if any), extend existing tracks, prune bad tracks and initiate new tracks. The output of the system is a list of target trajectories.

While processing the input data, the tracker builds a database which is used by different algorithms implemented in the Multiple Target Tracking system. The system has various data pruning features to prevent explosion of the track file database. Information processing in the tracker is a complex issue. This issue has been handled by concurrently decomposing the database which is distributed for concurrent processing to various nodes. The processed data is then collected to update the database for further processing. This in effect can be viewed as a distributed database information processing environment.

There are a number of algorithm implementations which impact the performance of this code on different architectures. A Newton-Raphson iteration involving small matrices is done for every track; straightforward vectorization of this task is ineffective. Track splitting and merging give low efficiencies on vector processor architectures. The Hypercube code requires two steps which are unnecessary in the sequential code: *aggregation* of a global table of track data and *redistribution* of the tracks to ensure load balancing and minimal data transfer. Global table construction and track redistribution are accomplished with general routing software built on communication primitives. Additionally, on message-passing machines, a time-consuming global communication phase is necessary in order to sort the tracks. This coupled with the unusual memory requirement of the code, leads to a rapid loss in performance when the grain size (number of tracks per processor) becomes small.

The tracker system has been developed in a parallel/distributed environment to tackle the information processing problem. The system is a scalable solution and has been implemented on various architectures which include iPSC/860, nCUBE, and a cluster of Sun and IBM workstations. The software we used include Express [16], PVM [14], PICL [15] and p4 [17].

The biggest single problem in running the tracker program on the iPSC/860, nCUBE and cluster of workstations was found to be the rather limited node memory on these machines. The track extension formalism used in the code is such that intermediate track files can have perhaps four times as many entries as there are real targets. This is particularly true at early times of the track exercise. The operating system requirement on the nodes reduces the availabilty of memory. As a result, the number of targets that are processed has been reduced substantially.

The tracking module will help in the development of specific benchmarks for different platforms like nCUBE, iPSC/860 and cluster of Sun and IBM worstations. It demonstrates the utility of benchmarks in parallel software development methodology. Additionally, the tracker will be used to examine the behavior and performance of various tools that have been used to implement the tracker program on different architectures mentioned above.

2.2 Description of Concurrent Multi-Target Tracking Scenario

In the following sections we describe Single Target Tracking, Multiple Target Tracking and the concurrent aspects of the Tracker. The tracker deals with the boost, post-boost and midcourse phases of a "mass raid scenario" and is designed to process a few thousand targets. The nominal task for the tracking program is to provide state information on individual targets given two dimensional line-of-sight data from the sensors at regular time intervals. A Single Target Tracking system is formed from two elementary Focal Plane Tracking subsystems. Each Focal Plane Tracking sub-system processes individual data from its associated sensor, forming lists of mono tracks. These mono tracks are shared between the two Focal Plane Tracking sub-systems, and a single set of 3D tracks is formed.

The targets of interest for the tracking program are thrusting rocket boosters. In an appropriate inertial time frame, the time dependence of the boost acceleration vector, $\alpha\bar{b}t$, is assumed to be known. An individual target is then completely specified by a four component launch parameter vector

$$p = (\theta_l, \Phi_l, \Psi_l, t_l) \tag{2.1}$$

where the components θ_1 and Φ_1 specify the latitude and longitude of the launch site, Ψ_1 specifies the initial launch azimuth relative to due north, and t_1 specifies the time of launch. For arbitrary threat scenarios, the objective of the tracking program is to determine, in real time, parameter vectors for each target. The measurements available to the tracking program are projections of booster position vectors onto a two-dimensional orthogonal grid. The sensor for the tracking model provides two dimensional measurements for all targets at fixed time intervals (Δ T) (typically 5 seconds). For each such scan of data, the activities of the tracking program can be divided into two major components: [3, ?, ?].

1. Single Target Tracking
2. Multiple Target Tracking.

2.3 Single Target Tracking

The tasks accomplished by Single Target Tracking tasks are

- Track System Initialization and
- Focal Plane Tracking.

2.3.1 Track System Initialization

The Track System Initialization [3, ?, ?] manages the initialization of various tracking modules which are as follows:

```
Number of Sites
Site 1 Description
.
.
.
Site N Description

Site Description ->

Number of Objects, Latitude, Longitude
Parameters, First Target
.
.
Parameters, Last Target

Parameter Description ->

Target type ( Primary or Secondary)
Latitude, Longitude
Aim, theta1, theta2
```

Figure 2.2: Format of Input Data for Internal Generation of Threats

- Focal Plane Tracking Filter,
- Global Focal Plane Tracking Parameter Module,
- Track Extension Module,
- Maneuver Processing Parameter Module,
- Focal Plane Report Module, and
- 3D ECI Filter

2.3.2 Focal Plane Tracking

Focal Plane Tracking consists of the Generation of Focal Plane Data Set and a 3-stage filter which consists of Focal Plane kinematic filtering, Estimation of Booster Launch Parameters from Focal Plane State Vectors and a Combination of Individual Parameter Estimates using a Second Filter.

- **Generation of Focal Plane Data sets**
  The tasks involved in generating the focal plane data set aims at generating threats internally. The

inputs for generating threats include (a) Number of Sites and (b) Description of each site. The Description of each site specifies parameters of all targets in that site. The parameters for each target include the Target type, Latitude, Longitude, Aim, θ_1 and θ_2 (see Figure 2.2). The tasks involved in the generation of focal plane data sets can be itemized as follows:

1. Set sensor specification list for a given scan. Fill the x and y entries of sensor definition for the requested time.
2. Initialize a structure for all the objects that have been launched. All these objects are visible objects. The size of the object list is known globally.
3. Generate data for visible objects.
4. Decompose data blocks and distribute them among nodes such that each node is assigned to different parts of the object list.
5. Calculate true focal plane track projections of the target vector of each object onto the sensor focal plane. The true focal plane projections are the focal projections, velocities and acceleration at time 't'.
6. Sort data items and store data count.

- **Focal Plane Kinematic Filtering**
  This filter processes the 2-dimensional measurements from the sensor to form estimates of projected kinematic quantities as seen in a sensor focal plane. Positions, velocities, acceleration and jerks (j = da/dt) along with each of two orthogonal axes are used for the state variables in the filter, with stochastic contributions to the jerk introduced to allow the filter to respond to targets traveling along largely unconstrained trajectories. The size of this dynamic uncertainty is the only free parameter of the kinematic filter, and the determination of an appropriate value is discussed in detail in Ref [9]. Since the focal plane filter is linear, all gain and covariance matrices can be computed and tabulated during initialization of the tracking program. The primary output of the focal plane kinematic filter is an estimate for the reduced state vector,

$$X_{FP}[k] = (x_s, z_s, dx_s/dt, dz_s/dt)^T \tag{2.2}$$

  consisting of projected positions and velocities at time step t_k.

- **Estimation of Booster Launch Parameters from Focal Plane State Vectors**
  In this step the equation stated above is used to provide an estimate of the launch parameters of the target. For the specific trajectory model an individual trajectory is specified by a 4 component parameter vector

$$p = (\theta_l, \Phi_l, \Psi_l, t_l) \tag{2.3}$$

  The parameters of this equation have been discussed in the previous section. For the given threat scenario the four components of Eq. 2.2 are sufficient for determining the the four launch parameters in Eq. 2.3.

- **Combination of the Individual Parameters using a Second Filter**
  This step of the 3 stage filter simply inverts the relation

$$x_{OBS}[k] = f(p). \tag{2.4}$$

As justified in Ref [9], the launch parameter and covariance estimates at each scan are (essentially) independent, and the last step of the 3-stage filter simply combines these estimates to form a cumulative parameter/covariance estimate, using completely standard techniques.

Focal Plane Tracking has the following concurrent tasks

- Focal Plane Track Extensions,
- Track Redistribution
- Focal Plane Report Construction
- Focal Plane Track Initiations.

Refer to [3, ?, ?] for details. The timings for these concurrent tasks on 2,4,8 and 16 nodes are used to estimate the performance of the Multi-Target Tracker.

2.4 Multiple Target Tracking/3D Tracking

The essential problem of Multiple Target Tracking is that of associating individual sensor reports with distinct underlying tracks. If two tracks in the system are found to be equivalent then one of the tracks is simply deleted. The task of identifying equivalent tracks in the Focal Plane track file dictates the manner in which the Focal Plane tracking problem is decomposed for concurrent execution. The Multiple Target Tracker also often referred to as the 3D tracker, maintains one track per sensor data point, representing the best global interpretation of tracks through the data. In place of the multiple hypothesis model used for Focal Plane tracking, the Multiple Target Tracker is based on optimal associations. The optimal associations are of two distinct forms [8]:

1. A track-split extension mechanism which extends tracks already in a global track file with sensor reports for new data scans.
2. A track initiator which generates new entries in the global track file from previously unused data points.

The adoption of an optimal association [6] formalism essentially trivializes the concurrent decomposition of the 3D tracker. The 3D tracks are distributed among the nodes in such a way that the number of tracks per node is constant. The challenge of concurrent 3D tracking comes entirely in performing the two types of optimal associations. A general concurrent algorithm for optimal association (Munkres Algorithm) is used [4]. The resource requirement for the general optimal association problem is such that a straightforward use of the general association formalism is completely inappropriate. Instead the concurrent algorithm proceeds as follows:

1. Each node computes a list of association keys (i.e., projections onto an appropriate reference axis), for all items in its local track list.
2. The distributed lists of association are globally sorted.

3. The sorted lists are divided into a number of sub-blocks determined by appropriately large gaps in the list of keys.
4. The sub-blocks are assigned to individual nodes and the assignment problems for sub-blocks are solved using a modified formalism of the general assignment problem.

The number of separate sub-blocks found should be large as compared to the number of nodes in the tracking system. This is always the case for the concurrent tracker problem. The 3D tracker also evaluates the trajectory fit for all 3D tracks in the system. All tracking is done using kinematic system models. The 3D track file structures are huge (more than 100 floating point numbers per track). Once the tracks are distributed to each node, the estimation of track parameters is performed concurrently with each node independently performing this task for its own subset of the global track file. Multiple Target Tracking tracking is accomplished through the following tasks:

- Track File Redistribution
- Data Prediction and Associations
- Identification and Deletion of Poor Tracks
- Focal Plane Report Associations and
- 3D Initializations

2.5 Concurrent Aspects of Multi-Target Tracker

Due to the large number of track candidates, the main tracking task is the processing of a global track file. The global track file contains all the details of processed data obtained from the two geostationary sensors and threat description file. The Focal Plane Tracking module uses this track file for coarse track predictions. After each scan of data, this track file is updated by the Focal Plane Tracking modules. The 3D tracking system uses this track file to do all the necessary processing. The size of this global track file depends on the number of sweeps/scans of sensor observations and the total number of targets that are launched. The size of data to be processed here offers opportunities for concurrent processing. Each node has access to the global file. Concurrency is obtained by assigning parts of the global track file to each node. Each node performs the sequential Multiple Target Tracking algorithm for its subset of data in the global track file. This file is then redistributed to all the nodes in preparation for the next scan of data. Redistribution is done such that all tracks ending at a given data point must be assigned to the same node in the next scan. The underlying assumption is that tracks ending at a given datum, if grouped together, will give the maximum number of track-hit associations.

The concurrent tasks that are done by the Focal Plane Tracking Module are as follows:

- Focal Plane Track Extension
  In this task the track extension algorithm uses the global track file to extend already existing or

identified tracks. It is assumed that if a track appears in three consecutive scans then this is an authentic track.

- Track file Redistribution
  Data processed after each scan is redistributed as discussed above.
- Focal Plane Report Construction
  The sensor reports of processed data is created concurrently by each node.
- Focal Plane Track Initiation
  The track initiation algorithm uses the global track file to identify any new tracks.

The various concurrent tasks associated with the 3D Tracking Module are as follows:

- 3D track extension
  In this task the track extension algorithm uses the tracks generated by the Focal Plane Tracker and extends existing tracks using the same techniques used by the Focal Plane Tracker. For further details refer [6, ?].
- 3D report association
  The Focal Plane reports generated by the Focal Plane Trackers are combined to produce a single report. Additionally the report file for the present scan is correlated with the previous two scans so that consistent track identifiers are maintained throughout the lifetime of the threat.
- 3D track initiation
  The 3D track initiator investigates all three-hit candidate segments from the present and previous scans of data attempting to find potential tracks which satisfy simple kinematic cuts for the estimated velocities and accelerations of the 3-hit segment. Any new tracks produced as a result of report associations are initiated.
- Trajectory estimation.
  This tasks generates a global report file containing parameter estimates, covariance matrices and uniqueness tags with precisely one entry for each sensor report association. This file is then sorted according to the estimated launch longitudes of the individual threats.

For more details regarding the tasks mentioned above, refer to [3, ?, ?]. The next section describes the initial implementation of the tracker which will be used to describe our implementation of the tracker on different tools.

2.6 Initial Implementation of the Tracker

The initial version of the tracker was implemented using the C programming language and CUBIX programming model (see Appendix I; Introduction to Express). All communication in the tracker was implemented using CrOS III primitives. Since CrOS III primitives [7] are supported by Express, the first step was to

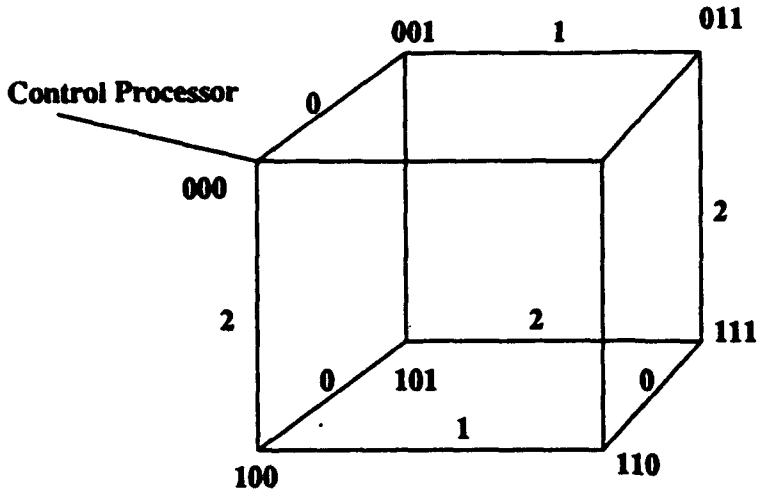
```
1. Communication call: Implements a read and write simultaneously
*  cshift( inbuf, sizeof(inbuf), source, outbuf, sizeof(outbuf), destination)
   where: inbuf is the buffer for incoming message and
    : outbuf is the buffer for outgoing message

2. Global Communication calls: These calls implement global sum,
   global maximum and global minimum.

*   glob_sum(value),
*   glob_max(value),
*   glob_min(value).

3. Topology Mapping calls:
* gridinit(dimension, no_of_processors),
* gridcoord(processor_number, &position_in_grid),
* prev_node = grid_chan(processor_number, dimension, -1),
* next_node = grid_chan(processor_number, dimension, 1).
```

Figure 2.3: Cros III Primitives

port it to Express which was known to be stable and had some history of support and maintenance. This task involved porting the tracker to Express by developing a CUBIX version using universal primitives. Universal primitives refer to those primitives that are supported by most message passing tools (eg., send and receive primitives). While these primitives may not have the same syntax, their semantics remain the same for all the tools. On the other hand primitives like *exparam* in Express [16] are specifically aimed at grabbing the cube environment. Such primtivies are not provided by all tools. In what follows we discuss the features of the original implementation of the concurrent tracker and our approach to implement it using universal primitives.

2.6.1 Communication Primitives in the Concurrent tracker

The communication primitive *cshift* [7] is used for all communications in the original implementation of the concurrent tracker. The purpose of *cshift* is to allow a group of nodes to both write and read messages among themselves without the application program specifying which nodes write first and which nodes read first. For instance, a pair of nodes can exchange messages and a group of nodes can pass messages in one step around a ring using *cshift*. The essential feature is that *cshift* is complemented on the other side of a channel by itself, rather than by a different routine. It must do both a write and read, but the order in which it calls them must be different for different nodes.



Labelling of processor numbers and channels in a three dimensional hypercube

End to End ring topology used in the tracker

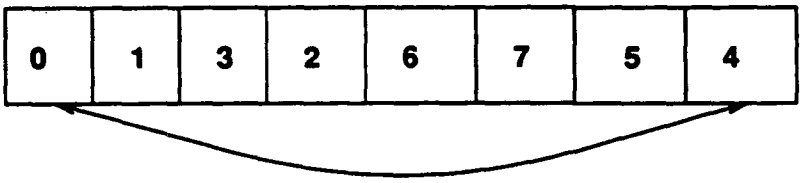


Figure 2.4: Communication and Topology Mapping calls

The tracker was originally implemented on a Mark III Hypercube. Since the only available means of distinguishing the nodes is their processor numbers, the order of reading and writing must be based on only a node's processor number and the hypercube topology. The original implementation of the tracker utilizes the following communication primitives *cshift()*, *gridinit()*, *gridcoord()*, *gridchan()*, *glob_sum()*, *glob_max()*, *glob_min* and *glob_and()*. Syntax of some of these primitives is shown in figure 2.3. Ref [7] discusses in detail all Cros III primitives.

Nodes in a hypercube that are connected by a communication channel differ by a single bit in their processor numbers. Figure 2.4.a shows the placement of nodes in a three dimensional architecture. It can be seen that all nodes with an even number of one bits in their processor numbers are adjacent only to nodes with an odd number of one bits in their processor numbers, and vice versa. Thus, the number of ones in a node's processor number can be used to determine whether the processor/node reads or writes first in *cshift*. In this implementation of *cshift* the buffers provided by the application for the incoming and outgoing messages must not overlap.

The tracker code uses *cshift* to implement global communications. The different global communication routines are:

1. glob_sum: Return cube-wide *sum* of integer arguments.
2. glob_max: Return cube-wide *maximum* of integer arguments.
3. glob_and: Return cube-wide logical *and* of integer arguments.
4. glob_rmax: Return cube-wide *maximum* of float arguments.
5. glob_rsum: Return cube-wide *sum* of float arguments.

Implementation of some of these routines in the concurrent tracker is shown in figure 2.5

2.6.2 Standard I/O in the Concurrent Tracker

The CUBIX programming model has two I/O modes. These two modes are termed as singular and multiple I/O modes. The use of singular mode is subject to a few important restrictions. First it should be clear that each node of the ensemble must concurrently execute identical I/O operations. When an input or output stream is operating in multiple mode, the individual processes may invoke CUBIX library routines with different arguments or data. The distinct data in this case is routed sequentially to or from the I/O device in an increasing order. CUBIX handles the switching of a stream between singular and multiple modes through a pair of loosely synchronous subroutines calls. An I/O stream can be switched between multiple and singular modes with the functions *fmulti* and *fsingl* [7]. The concurrent tracker has numerous *printf* statements in the code which are used for printing tracker statistics on standard output. Since the tracker works on a scan by scan basis these statistics are printed for every scan. The main timings of interest are the cumulative timings of concurrent tasks in the tracker. These timings can only be obtained at the end of the last scan. Therefore it was a good idea to inhibit the output to *stdout* for all scans except the last scan. The tracker code was modified in order to achieve this functionality. We have restricted our discussion only to screen I/O in the concurrent tracker. We have not addressed the problem of disk I/O in this project.

2.6.3 Node Environment Information

The environment grabbing function (*cparam()* in CrOS, *exparam()* in Express [7]) supplies important information about the node environment, some of which is specific to the hypercube topology. This function initializes the structure shown in Figure 2.6

The information needed by the tracker is:

1. The processor number of the node,
2. The total number of processors/nodes, and

```
/* Calculate the global sum using cshift  */
/* val_glob :-> value in my node          */
/* val_comm :-> value received from node  */
/* on other side of the channel           */

glob_sum(myval)
int myval;
{
int chan,lchan;
static int only_once = 0;

  val_glob = myval;
  for(lchan=0;lchan<dosc;++lchan)
  {
    chan = 1<<lchan;
    cshift(&val_comm,chan,INTSYZ,&val_glob,chan,INTSYZ);
    val_glob += val_comm;
  }

  return(val_glob);
}

glob_max(myval)
int myval;
{
int chan,lchan;

  val_glob = myval;
  for(lchan=0;lchan<dosc;++lchan)
  {
    chan = 1<<lchan;
    cshift(&val_comm,chan,INTSYZ,&val_glob,chan,INTSYZ);
    if( val_comm>val_glob ) val_glob = val_comm;
  }
  return(val_glob);
}
```

Figure 2.5: Implementation of Global Communication Routines in the Tracker

```
/* User level structure defining the cube environment in CrOS III */

struct cubenv {
    int doc;            /* Dimension of subcube */
    int procnum;        /* Processor number */
    int nproc;          /* Number of processors: 1<<doc */
    int cpmask;         /* Mask for host communication, NULL if not node 0
*/
    int cubemask;       /* Mask for node communication, NULL if not host */
};

A pointer to this structure is passed to the function cparam(). This
function initializes all the elements of the structure. This call is
specific to CrOS III. This facility is not available on all
the tools.

Declaration : struct cubenv CU_env;
Invocation  : cparam(&CU_env);

                    *****xxxx******

/* User level structure defining the cube environment in Express */

struct nodenv {
    int procnum;        /* Processor number of calling program */
    int nprocs;         /* Number of nodes in this processor group */
    int host;           /* Processor number for "HOST" processor */
    int taskid;         /* Identifies particular task on a node */
};

Declaration : struct nodenv NU_env;
Invocation  : exparam(&NU_env);
```

Figure 2.6: Cube environment Structure Templates

3. The dimension of the cube.

These details are specific to a hypercube environment and are used by the topology decomposition routines discussed below. This information is not needed when the tracker runs on a cluster of computers. The tracker has been modified so that it can obtain this information independent of the environment on which the tracker runs.

2.6.4 Mapping a Decomposition Topology to the Concurrent Tracker

The communication routines presented in section 2.6.1 require either a source or a destination processor except when the source or destination is implied (e.g., routines that implement collective communications between the nodes and the control process). However, when an actual application is programmed on a concurrent machine, the problem is decomposed into sub-problems that are connected by a communication topology without explicitly assigning a particular node to each region of that topology.

For example, in a problem involving a large matrix, the matrix might be decomposed into square submatrices, so that each submatrix could be assigned to a node. Such a decomposition results in a decomposition topology that is a two-dimensional grid of submatrices. While the algorithm can easily specify the necessary communications in terms of the decomposition topology, the communication routines require that processor numbers be used to refer to the source and destination nodes. In order for a concurrent algorithm to specify the source and destination processor numbers conveniently, it needs a mapping of the processing nodes onto the regions of the decomposition topology. Although the mapping routines are not communication routines, they provide a convenient utility for effectively using the communication routines.

While many types of mappings are possible, we focus on decomposition topologies that are based on Cartesian grids. The topology mapping routines used in the concurrent tracker are *gridinit()*, *gridchan()* and *gridcoord()* (see Figure 2.4.b). The *gridinit()* function takes the dimensionality of the grid and the number of nodes in each dimension as arguments and maps the nodes of the ensemble onto a *dim*-dimensional Cartesian grid. It basically performs the necessary initializations for the other two routines. The *gridcoord()* stores the cartesian coordinates of the nodes whose processor number is *proc* in the array. Figure 2.4.b shows the topology mapping of 8 nodes on a cartesian grid and the co-ordinates of the processor number in the grid. The routine *gridchan()* takes a processor number, a grid dimension and a direction, and returns the channel mask of the channel connecting the specified node to its neighbor in the indicated direction. Thus an application can use *gridchan()* to determine the mask of the channel on which to send the message so that it arrives at the neighboring nodes in any of the particular decomposition grid directions. This important is information because *cshift()* uses the channel number to communicate with any other node.

Chapter 3

Tracker Implementation

3.1 Porting the tracker on different tools

One objective of the tracker implementation was to have versions of Express, PVM, PICL and p4 with minimal changes to the existing tracker code. The main problem that is normally encountered in the process of porting any application is the presence of environment or tool-specific implementations in the source code. An application which has minimal functions specific to a tool will be relatively easy to port. At the same time having an application totally independent of any tool or environment is very difficult. The only solution is to write an interface library and implement this interface using the primitives of each tool.

In the Multi-Target Tracker implementation, communication primitives are tool or environment dependent. The original implementation assumes a *grid decomposition topology* which is not supported by all tools. See Section 2.6 for more details on this topology. This makes the porting process more difficult. We solved this problem by identifying all the communication primitives that are commonly available on the tools we considered (Express [16], PVM [14], PICL [15] and p4 [17]). We also made sure that these primitives were general enough to be supported by other software tools. It can be seen that the standard message passing primitives that are supported by most parallel/distributed software tool are facilities for *send*, *receive*, and *global* communication.

We selected a set of few message passing primitives from this set which were necessary to implement all the communication primitives used in the original implementation of the tracker. We then implemented the same functionality of the communication primitives in the original implementation of the tracker using this set. This ensured that the overall functionality of the tracker was not affected. This process had to be repeated for each message passing tool we used in this project. For example the standard send primitive in Express is *exwrite()* and the standard receive primitive in Express is *exread()*. These were the standard send and receive primitives we used to implement all communication facilities in Express. An implementation of the communication primitive *cshift* for Express is shown in Figure 3.3. It can be seen that the the Express primitive *exwrite* is used to send a message and *exread* is used to receive a message.

Implementations of cshift on other tools vary only in these send and receive primitives.

The host-node programming model was chosen to develop a universal version of the tracker because this model is supported by all tools we have used in this project. By a universal version we mean a version of the tracker which has the same structure and functionality on all the hardware architectures and tools. The only difference between them are the communication primitives that is specific to the tool under consideration. We now discuss the overall approach to develop this universal version of the tracker. The details below aim at giving the reader an overview of our implementation approach.

1 **Tool Installation (Express, PVM, PICL, p4)**

The first step in the process of porting an application to a tool is to install the tool if it does not already exist. We have discussed this process because in the course of this project we had to install all the tools except Express. All the tools that have been mentioned above, except Express, are public domain tools. The installation procedure for each tool is different and the documentation was not sufficient. However, the process of installing the tool helped us to understand the general organizational structure of the tool, the requirements of the tool and also the procedure to build the tool environment. Each tool also has specific configuration/set up procedures . These include creating and editing configuration files and starting background processes or daemons (*exinit* in Express and *pvmd* in PVM). For example, in Express three files *confile, netfile* and *express.cst* together specify the complete configuration set up. PVM has a *hostfile* which specifies all the machines that can be used by *pvmd* for starting node daemons.

2. **Understand the tool under consideration (Express, PVM, PICL and p4)**
The following issues normally arise in the process of working with new software tools:

   (a) **The structure of a normal host-node program** in the tool. We were specifically interested in this structure because it is supported by all the tools.

   (b) **The standard build/make procedure** for the tool. This is an important aspect because it helps the user/programmer understand the type of compiler and compiler options that have to be used for compiling application programs.

   (c) **The function calls and communication primitives** that are provided by the tool. We identified a subset of these calls for our use and tested their functionality by writing test stubs.

   (d) **Sample programs** that exhibit various features of the tool. This is a very helpful feature because it gives the user/programmer the capability to comprehend the tool. In this project we modified the sample programs to include the communication primitives we wanted to use in the tracker to verify our implementation.

   At the end of this process we had developed a sample program which used all the function calls that we wished to use in the tracker implementation.

3. **Develop a standard methodology**
We used the sample program that we developed [10], to test the various features and primitives that we planned to use in the tracker. Since we preferred minimal changes to the existing tracker code, our

approach was to front-end all the specific CrOS III primitives. The specific CrOS III primitives that we implemented using standard primitives are *cshift()* , *gridinit()* , *gridcoord()* , *gridchan()* , and all global communication routines. The standard *testbench* that we developed used all the primitives we had decided to front-end. We also had a testbench version which used the original primitives. The procedure we developed was *fool-proof* because the standard testbench version had to tally with the version we developed using the original primitives. This ensured that the front-ended primitives had exactly the same functionality as the original primitives.

4. **A host-node structure template**
   The process of building a standard testbench helped in defining a host-node structure for the tracker. The host-node structure that has been developed remains the same for all the tools. A template of a typical host program and a node program is shown in Figure 3.1 and Figure 3.2 respectively.

It can be seen in Figure 3.1 that the host program prompts the user to enter the number of nodes that will be used by the application. It checks if the number of nodes is a power of two and then starts the host process. On successful instantiation of the host process (statement (1) in the host template the program initiates node processes commensurate with the number of nodes specified by the user (statement (2) in the host template). The host program then sends the number of nodes that the application will use to each node (statement (3) in the host template.) This information is required by the nodes for communicating with each other. This is done by using a standard *send()* call. The procedure of sending a message is different in each tool. We discuss the specific implementations of each tool in a later section. The host program then waits for all the node processes to end (statement (5) in the host template). This has been implemented by using a dummy *receive* call in the host program and a dummy *send* call in the node program.

From Figure 3.2 it can be seen that each node gets an identification number for itself by using the appropriate tool-specific call (for example, *enroll* in PVM) (statement (1) in node template). It then receives the number of nodes that will be used by the host program (statement (2) in node template). This information is sent to each node by the host program. It then calculates the dimension of the cube which is used by the tracker for specific implementation details. The node program then has a set of topology decomposition routines (statement (3) in node template). These routines are function calls supported by Express and CrOS III. For the sake of uniformity we have re-written these routines so that they can be used by all the tools. The different routines that have been used in the tracker implementation are *gridinit()*, *gridcoord()* and *gridchan()* (discussed in Section 2.6.4).

The remaining part of the code in the node program is the implementation of the tracker. This remains the same for all the tools and is totally independent of the tool except for communication calls. At the end the node program sends a dummy message to the host (statement (4) in node template) to signal the termination of node processes.

The main communication routine used in the initial implementation of the tracker is *cshift*. Since this was specific to CrOS III we have implemented our own *cshift* function call using standard communication primitives. Figure 3.3 shows a generic implementation of *cshift* which implements a *read* and *write* functionality together. This is equivalent to the *exchange()* function call in Express. The original *cshift()* function uses the concept of channels to establish communication between any two nodes. A channel number between any two nodes is the "exclusive or" of the source and destination node. From Figure

```
main(argc, argv)
int argc;
char *argv[];
{
    int my_process_number, number_of_processors;
    int return_log_value;

    /* Prompt the user for number of nodes/processors */
    printf("Enter the number of nodes: \n");
    scanf( "%d", \&number_of_processors);

    /* Include check to ensure that the  number of processors / instances */
    /* is a power of 2. This is not required if the program is being run */
    /* on a network of workstations. However, for the sake of uniformity */
    /* we have continued to use this checking. */

    return_log_value = log2(number_of_processors);


    /* Exit if condition not satisfied */
    .
    .
    /* Start the host process */ ---- (1)
    .
    .
    /* Initiate node processes for ''number_of_processors'' */ ---- (2)
    .
    .
    /* Send total number of processors/instances to node processes */   ---- (3)
    /* In environments like PVM there is a specific set up procedure */
    /* before sending a message. Check section on PVM implementation for */
    /* more details.  */
    .
    .
    /* Wait for all the node processes to end.  */                ---- (4)
    /* This has been implemented by  using a dummy receive in the host. */
    /*  This is complemented by a  send in the node process */


    .
    /* Exit / Leave */                    ---- (5)
}
```

Figure 3.1: Host Program Structure Template

```
main()
{
    int number_of_processors, my_node_no, my_position, previous_node,next_node;

    /*  Enroll Node process / instance */            ---------------- (1)
    This returns my node number (my_node_no);

    /*  Initialize the environment variables */

    number_of_processors = receive this value from the host process;
                                                     --------------- (2)
    Dimension of cube = log2(number_of_processors);

    /* The statement above is required because the dimension of the cube */
    /* is used by the tracker in various places. Therefore even if the */
    /* tracker is running in a network environment this variable is */
    /* initialized although this is specific to hypercubes */

    /* Determine if Concurrent processing is required. This is a feature of */
    /* the tracker because it is can operate as a sequential code as well as */
    /* a concurrent piece of code. This is determined by the dimension */
    /* of the cube ( the variable initialized above ).  */

    /* Topology Decomposition routines */

    /* Initialize a grid */
    grid_init( 1, \&number_of_processors);       --------------- (3)

    /* Get my co-ordinate position in the grid */
    grid_coord( my_node_no, \&my_position);

    /* What is my left neighbors node number in the grid */
    previous_node = grid_chan(my_node_no, 0 -1);

    /* What is my right neighbors node number in the grid */
    next_node = grid_chan( my_node, 0, 1);

    Sequential / Concurrent Tracking code
    .
    /* Send dummy message to the host because it is waiting */
                                                     --------------- (4)
    /* Exit / Leave */
}
```

Figure 3.2: Node Program Structure Template

3.3 it can be seen that the destination node number is first calculated by using the source node number (*pc_corn*) and the out channel *outchan*. The type number is then set which identifies the type of message being sent. The flag bit is used to identify if the node has an even number or odd number of ones in its binary representation. For example, the binary representation of node number three is '011'.

The number of one bits in this case is two. The flag bit is set if the number of ones is even in a node number and reset otherwise. This is the key feature in communication between any two nodes, because in a hypercube architecture any two adjacent nodes differ by one bit. We have used the same concept for communication when we used a network of workstations. Depending on the value of the bit flag a node first writes to a destination and then reads from a destination or viceversa. Figure 3.3 shows the Express implementation of cshift where we have used *exwrite* and *exread* for writing and reading a message. The only things that will change for different tool implementations are the communication primitives. The function call *cshift()* is also used to implement all global communications in the tracker as discussed in Section 2.6.

3.2 Performance of Concurrent Tracker

In this section we discuss the performance results of the Multi-Target Tracker [11, ?, ?]. We present the performance results of the Multi-Target Tracker using four tools (Express, PVM, PICL and p4). The memory requirement of the tracker increases with an increase in the number of *objects/threats*. The tracker reads in threat descriptions from files and generates data using kinematic algorithms. However, each node has a copy of this generated data which is stored in an array of structures. Work load distribution for concurrent processing is achieved by indexing where each node works on a section of the array of structures present in each node. This mechanism of data distribution degrades the performance of the tracker because of excessive memory usage.

The parallel version of the tracker has been implemented by extracting parallelism from an existing sequential version. As a result the tracker is not a purely parallel code. The difference between concurrent and sequential versions is the way the global track file is processed. In the sequential algorithm each track is extended by all possible data associations. Duplicate tracks are merged to form initial tracks for processing during the next scan of data. In the concurrent version tracks are assigned to nodes such that tracks ending in a given sensor datum are assigned to the same node. Each node performs *extend/merge* tasks independently for each time step and then these tracks are redistributed.

In this project we have identified the main concurrent tasks of the tracker and measured their performance. The timings for these concurrent tasks are obtained from timing statements embedded in the tracker program. The parameter varied in the benchmarking process is *number of processors*. Another parameter that can be varied is the number of threats. This can be done by increasing the number of launch sites in the file "*threat.dat*". The number of sites is the first field in this file. See the threat file description discussed in Chapter 2 for more detail. The only point to note here is that each site should have an associated *object list/threat* description in the file "*threat.dat*".

```
int cshift ( inbuf, inchan, inbytes, outbuf, outchan, outbytes)
char *inbuf;
unsigned int inchan;
int inbytes;
char *outbuf;
unsigned int outchan;
int outbytes;
{
    int rstatus, wstatus;

        int dest;
        int type;

        dest = pc_corn  ^ outchan;

        type = 5;

        if ( bit_flag) {
       /* Send a message to destination node ''dest'' */
           exwrite(outbuf,outbytes, \&dest, \&type); /* Express Implementation */
   /* Receive a message from destination ''dest '' */
           exread(inbuf,inbytes, \&dest, \&type); /* Express Implementation */
        }
        else {
   /* Receive a message from destination ''dest'' */
           exread(inbuf,inbytes, \&dest, \&type); /* Express Implementation */
   /* Send a message to destination ''dest'' */
           exwrite(outbuf,outbytes, \&dest, \&type); /* Express Implementation */
        }
        if (wstatus ==-1) return(wstatus);
        else return(rstatus);

}
```

Figure 3.3: Cshift Implementation

The two main concurrent tasks in the Multi-Target Tracker are Focal Plane Tracking and 3D Tracking. These tasks are further divided into concurrent sub-tasks as listed below (previously discussed in sections 2.3 and 2.4):

1. **Focal Plane Tracking**
   The concurrent tasks in this module are
   (a) Focal Plane Track Extension
   (b) Track Redistribution
   (c) Focal Plane Report Construction
   (d) Focal Plane Track Initiation
2. **3D Tracking**
   The concurrent tasks in this module are
   (a) 3D Track extensions
   (b) Report Associations
   (c) 3D Track Initiations
   (d) Trajectory Estimation

The tracker program has been tested for 1, 2, 4 8 and 16 nodes on the iPSC/860, nCUBE and on a cluster of workstations. The data sets that we used for performance measurement are as follows:

1. Thirty sweeps/scans of data at 5 second intervals.
2. 8 launch sites with 80 targets per site (i.e., the number of targets is 640).

The number of targets has been limited to 640 because it is directly dependent on the memory available in the nodes of these machines. The cluster of workstations could not handle more than 640 targets, although we have tested the tracker for 1100 targets on the iPSC and nCUBE. The tracker has been implemented using different parallel/distributed software engineering tools. The hardware environment used for porting and evaluating the tracker environment is shown in Figure 3.4. In this report we have included the performance benchmarks of the tracker for all the tools we used. However we present only one set of benchmarks for each tool which represents the best *machine-tool* combination.

Speedup and efficiency measures are used to analyze the performance of each of the tasks mentioned above. The timings for these tasks on different number of nodes is compared with the timings when they run sequentially on one node.

Speedup ($S(n)$) is given by the following equation

$$S(n) = \frac{Time\ taken\ by\ a\ task\ on\ one\ node}{Time\ taken\ by\ a\ task\ on\ n\ nodes} \tag{3.1}$$

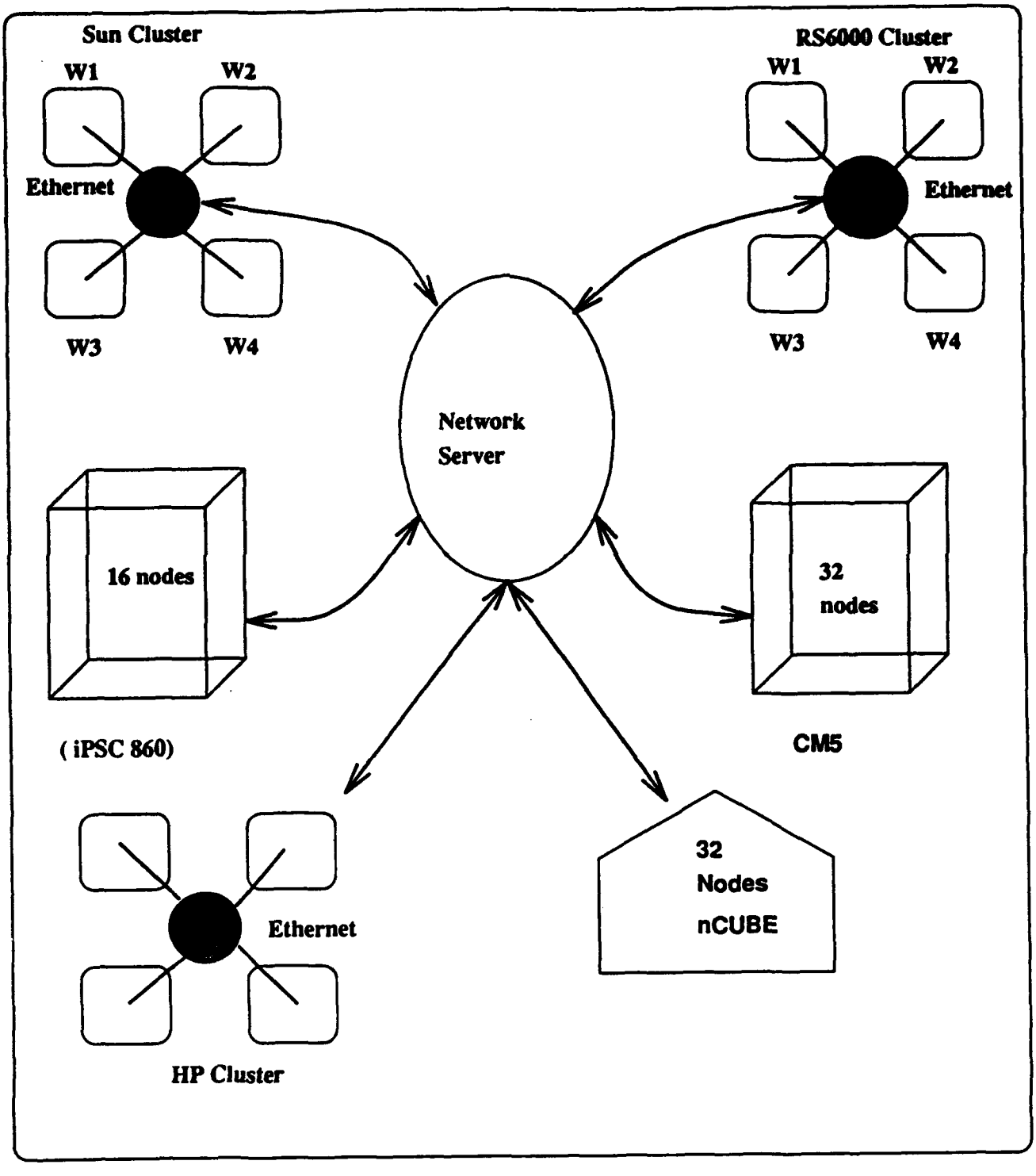


Figure 3.4: Hardware Environment for Multi-Target Concurrent Tracker

where $n \rightarrow 2, 4, 8, 16, \ldots$ nodes.

Efficiency of the concurrent tasks is given by

$$Efficiency = \frac{Speedup}{Number\ of\ Processors} \tag{3.2}$$

This measures the efficiency of running the tracker on n processors.

We will now discuss the salient features of each tool we used to implement the tracker. Each tool description will be followed by a brief explanation of the modifications we did to port the tracker on that tool. We will then present the performance of the tracker using that tool on different parallel/distributed architectures.

3.3 Express : Parallel Processing Toolkit

Express [16] is a software package developed by ParaSoft Corporation. It is designed to meet the needs of parallel/distributed applications. It is conceptually a multi-layered system. At the lowest level it provides support for allocating processors, loading programs and asynchronous message passing. At a higher level it provides the utilities designed to automatically decompose problems with regular structure. Each of the levels are logically distinct, building only on those below it. As a result it is possible to port the system to a wide variety of hardware/software systems. The Express communication environment offers a wide range of implementation strategies to both system and application designers. In particular it has been motivated by application requirements rather than any intrinsic operating system concepts. It provides a reasonable set of tools and utilities designed for parallel processing.

The salient features of Express include:

- Low level communication primitives for sending messages between processors, peripherals and other system components.
- High level message passing routines which perform a variety of parallel processing and communication tasks such as broadcasts, global averaging, global minimum/maximum computation and data redistribution.
- A *domain decomposition* library which can map problems from the physical domain in which they are naturally expressed to the underlying topology of the parallel computer hardware.
- A transparent I/O system for node processors.
- A parallel graphics system.
- PM - a graphical system for evaluating and enhancing the performance of parallel programs.

Changes to the Host Template
Statement 1 : pgind=exopen("dev/intel",nprocs,DONTCARE);
Statement 2 : exload(pgind, "node");
Statement 3 : This objective of this statement is to pass the total number of nodes that the application is using to each nodes. However Express uses the call *exparam* in the node program to do this. The function of this call is to provide each node information about the node environment.
Statement 4 : for (dest=0; dest<nprocs; dest++) exread(&mesg_no, sizeof(int), &dest, &type);
Statement 5 : exit(1);

Changes to Node Template
Statement 1 : This statement is not required in Express.
Statement 2 : The number of processors in the node environment is obtained by the call *exparam()*
Statement 3 : Remains the same as in node template.
Statement 4 : exwrite (&mesg_no, sizeof(int), &env.host, &type);

Figure 3.5: Changes to Host Program and Node Program Templates for Express Implementation

3.3.1 Express Implementation of Multi-Target Tracker

We have explained the generic host and node templates in section 3.1. To port the tracker to any tool we have to change statements 1 through 5 of the host template and statements 1 through 4 of the node template. The changes needed to implement the tracker on the Intel iPSC/860 in the Express environment are shown in Figure 3.5. Statement 1 in the Host Template allocates nodes for the program where *nprocs* specifies the total number of nodes. Statement 2 in the Host Template loads the node program on all the allocated nodes. This statement is complemented by Statement 1 in the Node Template which enrolls the node program and returns the instance of the node to the host program. The objective of Statement 3 in the Host Template is to broadcast the total number of nodes being used by the program to all the nodes. This statement is not used in Express because the function call *exparam()* can be used in the node program to get this information. Statement 2 in the Node template gets information of the node environment for the program. The key information required by the node program is the number of node instances and its own node identification number. Statement 3 in the Node Template initializes a grid for topology mapping. This grid information is used for communicating between nodes. Statement 4 in the Host Template is a dummy read (*exrread()*)implemented in the host program. The host program essentially waits at this point. This call is complemented by a dummy write(*exwrite()*) implemented in the node program. This is represented by Statement 4 in the Node Template. This statement is the last statement in the node program. Once the host program receives a message from all the nodes it exits which is represent by statement 5 in the Host Template.

3.3.2 Performance of the Multi-Target Tracker implemented using Express

Table 3.1: Performance of Main Tracking Tasks on iPSC/860 using Express (8 Sites, 640 targets)

Tracking Task	1 Node	2 Node	4 Node	8 Node
FP Tracking Summary				
FP Track Extension	181.98	130.58	63.78	40.74
Track Redistribution		6.31	17.44	30.17
FP Report Construction	10.57	6.95	8.35	9.95
FP Track Initiation	2.59	1.84	3.90	2.75
FP Tracking Total	195.14	145.68	93.47	83.61
3D Tracking Summary				
3D Track Extension	0.61	0.39	0.27	0.22
Report Association	21.53	31.23	26.05	21.92
3D track Initiations	2.02	1.11	0.59	0.38
Trajectory Estimation	5.16	3.05	1.78	1.09
Total 3D Tracking Task	29.32	35.78	28.69	24.27
Grand Total	224.46	181.46	122.16	107.88

Table 3.1 shows the timing for all concurrent tasks of the Multi-Target Tracker implemented using Express on iPSC/860. Table 3.2 shows the speedup and efficiency of running these tasks. While there are some tasks that speed up as we increase the number of nodes, there are others which do not speed up or fluctuate with the increase in the number of nodes. For example, *Track Extension* in Focal Plane Tracking as well as 3D Tracking show speedups (see Table 3.2) with the increase in number of nodes. However, *Track File Redistribution* and *Report Association* do not show increase in speedup. We observe that the tasks that do not show speedup do not consume a large percentage of the total tracking execution time for smaller numbers of nodes. Consequently, the current implementation of the tracker can show significant speedup for larger number of nodes only when the number of tasks to be processed is very large.

Table 3.2: Speedup and Efficiency of Main Tracking Task on iPSC/860 using Express (8 Sites, 640 targets)

Tracking Task	2 Node		4 Node		8 Node	
	speedup	efficiency	speedup	efficiency	speedup	efficiency
FP Tracking Summary						
FP Tracking Extension	1.39	69.50%	2.85	71.25%	4.47	55.84%
FP Report Construction	1.52	76.00%	1.27	23.91%	1.06	13.28%
FP Report Initiation	1.41	70.50%	0.66	16.50%	0.94	11.77%
FP Tracking Total	1.34	67.00%	2.09	52.25%	2.33	29.17%
3D Tracking Summary						
3D Track Extension	1.56	78.00%	2.26	56.50%	2.77	34.66%
Report Association	0.69	34.50%	0.83	20.75%	0.98	12.28%
3D track Initiations	1.82	91.00%	3.42	85.50%	5.32	66.45%
Trajectory Estimation	1.69	84.50%	2.89	72.25%	4.73	59.17%
Total 3D Tracking Task	0.82	41.00%	1.02	25.50%	1.21	15.10%
Grand Total	1.24	62.00%	1.84	46.00%	2.08	26.00%

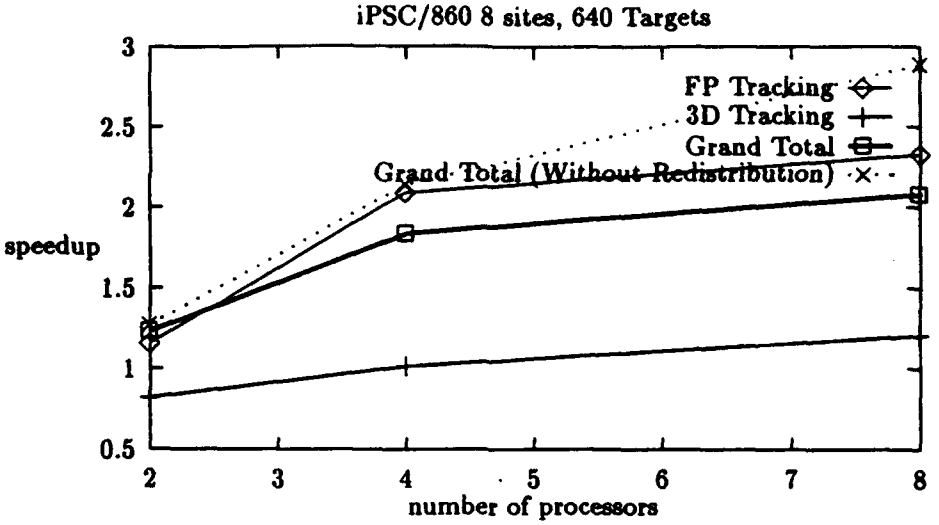


Figure 3.6: Plot of Main Tracking Tasks in Express

However with an increase in the number of nodes these tasks make up an appreciable percentage of the total time because the execution time for other tasks reduces (see Table 3.1).

This can be easily observed in the speedup curve shown in Figure 3.3.2. It can be seen that the tracker shows a speedup of more than '1' for upto 4 nodes. However after 4 nodes the performance of the tracker degrades. The main factors which affect the performance of the tracker are *Track File Redistribution* and *Report Association*. As discussed above we observe that speedup of concurrent tasks reduces with an increase in number of nodes because these two tasks consume an appreciable percentage in the total timing. Referring back to Table 3.1 we see that for 8 nodes the total execution time for *Track File Redistribution* and *Report Association* is 52.09 millisecs which is 48.2% of the total time. This is due to the fact that percentage of concurrent processing in each node reduces with the increase in the number of nodes. We also show a plot for the Grand Total time taken for concurrent tasks without *Track file Redistribution*. In this case there is an increase in speedup even after 4 nodes. This implies that this section of the code (*Track File Redistribution*) is not very efficient and should be improved for overall performance enhancement.

3.4 PVM (Parallel Virtual Machine)

PVM [14] has been developed at Oak Ridge National Laboratory and the Department of Math and Computer Science at Emory University. It is a software package that enables concurrent computing on loosely coupled networks of processing elements. The PVM computing model is based on the notion of a *virtual machine*. A virtual machine is a collection of networked computers, abstracted into a concurrent computing environment by the PVM system.

The salient features of PVM are multilanguage and heterogeneity support, scalability, provisions for fault tolerance, the use of multiprocessors and scalar machines, an interactive graphical front end, and support for profiling, tracing and visual analysis. PVM may be implemented on a hardware base consisting of different machine architectures, including single CPU systems, vector machines and multiprocessors. These computing elements may be interconnected by one or more networks, which may themselves be different (e.g., one implementation of PVM operates on Ethernet, the Internet, and a fiber-optic network). These computing elements are accessed by applications via a standard interface library that supports common concurrent processing paradigms in the form of well-defined primitives that are embedded in procedural host languages.

PVM is composed of two parts: a *daemon* that runs on each computer in a virtual machine and the *user interface library* mentioned above. When the user starts up the PVM daemon on a machine, he/she specifies an input file. This file contains a list of machines that will make up his/her virtual machine. The daemon starts up similar daemons on each of these computers. Sockets are set up between each of the daemons. All inter-daemon control and data traffic is conveyed over these sockets, and reliable delivery is guaranteed by the daemon software. Application programs are composed of *components* that are subtasks at a moderately large level of granularity. During execution, multiple *instances* of each component may be initiated. To become a part of the virtual machine each component must *enroll*. This interface routine establishes a socket between the component and the local daemon.

Application programs view the PVM system as a general and specific parallel computing resource. This resource may be accessed at three different levels: the *Transparent mode* in which component instances are automatically located at the most appropriate sites, the *Architecture-dependent mode* in which the user may indicate specific architectures to execute particular components, and the *Low-level mode* in which a particular machine may be specified.

Such layering permits flexibility while retaining the ability to exploit particular strengths of individual machines on the network. The PVM user interface is strongly typed; support for operating in a heterogeneous environment is provided in the form of special constructs that selectively perform machine-dependent data conversions where necessary. Inter-instance communication constructs include those for the exchange of data structures as well as high-level primitives providing the functionality of broadcast, barrier synchronization, mutual exclusion and rendezvous.

Application programs under PVM may possess arbitrary control and dependency structures. Any specific control and dependency structure may be implemented under the PVM system by appropriate use of PVM constructs and host language control-flow statements.
PVM supports C and Fortran by providing a separate library of communication primitives for each language. PVM also provides an X-window based software environment for parallel programs intended for unsophisticated programmers called HeNCE [14] (for Heterogeneous Networking Computing Environment). It is based on a parallel programming paradigm where an application program can be described by

a directed acyclic graph (DAG). HeNCE is composed of graphical tools for creating, compiling, executing, and analyzing HeNCE programs. HeNCE relies on PVM for process initialization and communication, but the HeNCE programmer will never explicitly write PVM code, thus providing the user with a high level of abstraction for exploiting the parallelism of a collection of machines without delving into the details of parallel programming.

3.4.1 PVM Implementation of the Multi-Target Tracker

For this implementation of the tracker we list the calls corresponding to statements for the host program structure and node program structure. These statements are shown in Figure 3.7. The PVM code was implemented on a cluster of SUN worstations and IBM/RS6000 workstations separately. A version was also tested in a heterogeneous environment of 4 SUN and 4 IBM/RS6000 workstations.

In Figure 3.7 Statement 1 in the Host Template enrolls the host program "pvm_host". Statement 2 in the Host Template initiates all the node processes "pvm_node". The number of node processes initiated is equal to *nprocs*. Statement 1 in the Node Template enrolls the node processes in each node. Statement 3 in the Host Template broadcasts a message to all nodes about the total number of node instances being used by the program. The corresponding Statement 2 in the Node Template receives this message from the host program. Statement 3 in the Node Template initializes a grid for topology mapping. This is used by the nodes for communicating between themselves. Statement 4 in the node template is at the end of the node program. Each node sends (*snd()*) a dummy message to the host program. This complemented by a receive (*rcv()*) in Statement 4 of the Host Template. Once the Host program receives this message from all the nodes the host program exits which is illustrated in Statement 5.

3.4.2 Performance of the Concurrent Tracker implemented using PVM

Changes to the Host Template

Statement 1 : me = enroll("pvm_host");

Statement 2 : for (i = 0;i< nprocs;i++) initiate("pvm_node", (char*)0);

Statement 3 :

```
initsend();
putnint( &nprocs, 1);
if (snd("pvm_node",-1,3) < 0) {
printf("HOST: Broadcast of 3 failed");
leave();
exit(1);
}
```

Statement 4 :

```
for ( i = 0; i < nprocs; i++)
tmp rcv(i);
getstring(&mesg_no,1);
```

Statement 5 : leave();

Changes to Node Template

Statement 1 : procnum = enroll("pvm_node");

Statement 2 :

```
nsproc = rcv(3);
getnint(&nsproc, 1);
```

Statement 3 : Remains the same as in the node template.

Statement 4 :

```
initsend();
putstring(&mesg_no, 1);
snd("pvm_host", -1, pc_corn);
leave(); exit(1);
```

Figure 3.7: Changes to Host Program and Node Program Templates for PVM Implementation

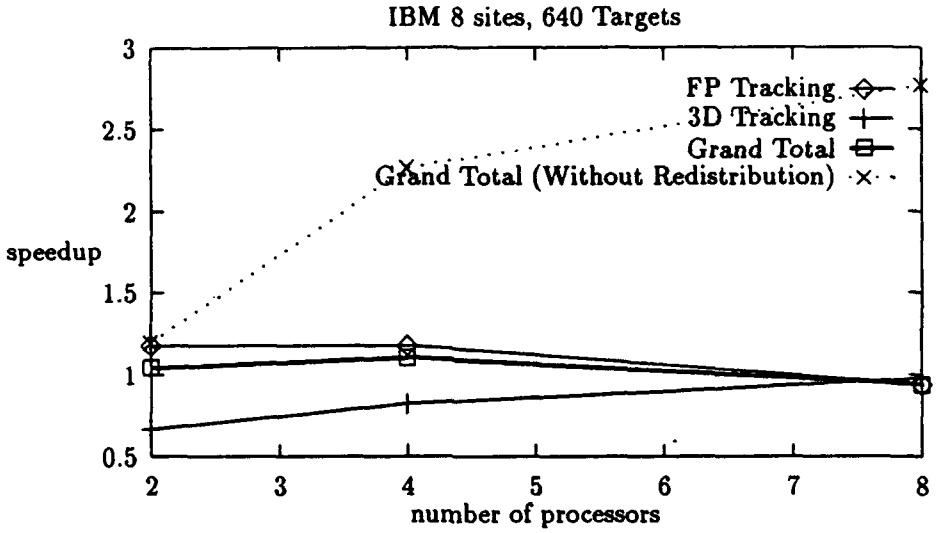


Figure 3.8: Plot of Main Tracking Task in PVM

Table 3.3: Performance of Main Tracking Tasks on IBM RS/6000 using PVM (8 Sites, 640 targets)

Tracking Task	1 Node	2 Node	4 Node	8 Node
FP Tracking Summary				
FP Track Extension	137.80	100.64	36.62	25.58
Track Redistribution		19.96	83.1	125.56
FP Report Construction	10.3	5.46	5.30	7.74
FP Track Initiation	3.62	2.28	3.72	3.58
FP Tracking Total	151.72	128.34	128.74	162.16
3D Tracking Summary				
3D Track Extension	0.28	0.24	0.25	0.33
Report Association	24.04	38.44	32.04	26.89
3D track Initiations	1.18	1.00	0.35	0.23
Trajectory Estimation	2.23	1.47	0.87	0.73
Total 3D Tracking Task	27.73	41.15	33.51	28.18
Grand Total	179.45	169.49	162.25	190.34

Table 3.3 shows the timing for the concurrent tasks of the Multi-Target Tracker implemented using PVM on a cluster of IBM RS/6000 workstations. Table 3.4 shows the speedup and efficiency of these tasks on this cluster. Figure 3.4.2 shows the plot for *Speedup* versus *number of processors* for these tasks. Referring to Table 3.3 we can see that the total time taken for *Track File Redistribution* and *Report Association* for 8 nodes is 152.45 millisecs. This contributes to 80% of the total time for FP Tracking. We attribute this increase from 48.2% in the previous case to 80% (see Tables 3.3 and 3.4) to the network latency

associated with Ethernet.

Table 3.4: Speedup and Efficency of Main Tracking Task on IBM RS/6000 (8 Sites, 640 targets) using PVM

Tracking Task	2 Node		4 Node		8 Node	
	speedup	efficiency	speedup	efficiency	speedup	efficiency
FP Tracking Summary						
FP Tracking Extension	1.36	68.00%	3.76	94.00%	5.38	67.25%
FP Report Construction	1.88	89.00%	1.94	48.50%	1.38	17.25%
FP Track Initiation	1.58	79.00%	0.97	24.25%	1.01	22.38%
FP Tracking Total	1.18	59.50%	1.18	29.50%	0.94	11.75%
3D Tracking Summary						
3D Track Extension	1.16	58.00%	1.12	28.00%	0.85	10.63%
Report Association	0.63	31.50%	0.75	18.75%	0.89	11.13%
3D track Initiations	1.00	50.00%	3.37	84.25%	5.13	64.13%
Trajectory Estimation	1.52	76.00%	2.56	64.00%	3.05	38.13%
Total 3D Tracking Task	0.67	33.50%	0.83	20.75%	0.98	12.25%
Grand Total	1.05	52.50%	1.11	27.65%	0.94	11.78%

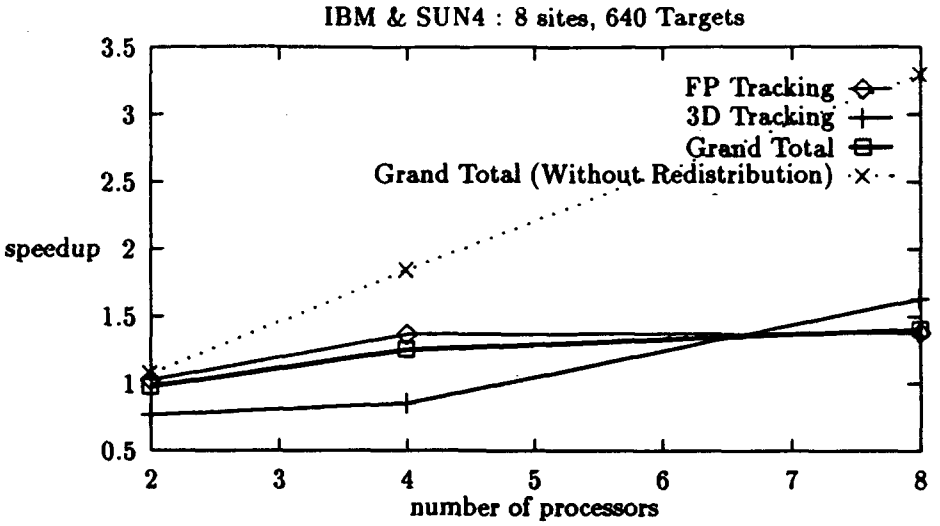


Figure 3.9: Plot of Main Tracking Task in PVM in a Heterogeneous Environment

Table 3.5: Performance of Main Tracking Task using PVM on Heterogeneous Nodes: SUN4 and IBM RS/6000 (8 Sites, 640 targets)

Tracking Task	1 Node	2 Node	4 Node	8 Node
FP Tracking Summary				
FP Track Extension	281.70	242.38	105.44	61.62
Track Redistribution		33.80	88.64	142.60
FP Report Construction	14.44	13.54	11.04	10.18
FP Track Initiation	3.48	2.00	4.04	3.32
FP Tracking Total	299.62	291.72	219.16	217.72
3D Tracking Summary				
3D Track Extension	1.27	1.48	0.82	0.63
Report Association	39.53	55.41	48.51	27.66
3D track Initiations	3.56	3.23	1.50	0.82
Trajectory Estimation	6.23	5.93	2.97	1.75
Total 3D Tracking Task	50.59	66.05	58.80	30.86
Grand Total	350.21	357.77	277.96	248.58

We used SUN4 and IBM RS/6000 workstations to test the tracker on a heterogeneous environment. Table 3.5 shows the timing for all concurrent tasks of the Multi-Target Tracker implemented using PVM on a cluster of 4 SUN4 and 4 IBM RS/6000 workstations. Table 3.6 shows the speedup and efficiency of main tracking tasks on this cluster. Figure 3.4.2 shows the plot for *Speedup* versus *Number of Processors* for

these tasks.

Table 3.6: Multiple Tracking Task on Heterogeneous Nodes: SUN4 and IBM RS/6000 (8 Sites, 640 targets) using PVM

Tracking Task	2 Node		4 Node		8 Node	
	speedup	efficiency	*speedup*	efficiency	speedup	efficiency
FP Tracking Summary						
FP Tracking Extension	1.16	58%	2.67	66.75%	4.57	57.13%
FP Report Construction	1.07	53.50%	1.31	32.75%	1.42	17.75%
FP Report Initiation	1.74	87.00%	0.86	21.50%	1.05	13.10%
FP Tracking Total	1.03	51.50%	1.37	34.25%	1.38	17.20%
3D Tracking Summary						
3D Track Extension	0.86	43.00%	1.55	38.75%	2.02	25.20%
Report Association	0.70	35.50%	0.81	20.25%	1.43	17.86%
3D track Initiations	1.10	55.00%	2.37	59.25%	4.34	54.27%
Trajectory Estimation	1.05	52.50%	2.10	52.50%	3.56	44.50%
Total 3D Tracking Task	0.77	38.50%	0.86	21.50%	1.64	20.50%
Grand Total	0.98	49.00%	1.26	26.00 %	1.41	17.63 %

It can be seen from Figure 3.4.2 that *3D Tracking Task* and *Grand Total (without Redistribution)* show appreciable speedups. We have ignored *Track Redistribution* because of network latency problems in the ethernet environment.

3.5 PICL : Portable Instrumented Communication Library

PICL [15] is a portable instrumented communication library developed by Oak Ridge National Laboratory. It is designed to provide portability, ease of programming, and execution tracing in parallel programs. It provides portability between many machines and multiprocessors environments. It is fully implemented on the Intel iPSC/860, in the nCUBE/3200 families of hypercube multiprocessors and on the Cogent multiprocessor workstations.

In addition to supplying low-level communication primitives such as *send* and *receive*, PICL simplifies parallel programming by providing a set of high-level communication routines for global broadcast, global maximum, and barrier synchronization. These routines can help the novice user avoid common synchronization and programming errors and save programming time even for the veteran user. These high-level routines also facilitate experimentation and performance optimization by supporting a variety of interconnection topologies. Execution tracing has been built into the PICL routines, and routines are provided to control the type and amount of tracing. A separate package called ParaGraph [15] is available to display the tracing output graphically. The tracing facility is useful for performance modeling, performance tuning and debugging. The PICL library is made up of three distinct sets of routines: a set of low level communication and system primitives, a set of high-level global communication routines and a set of routines for

invoking and controlling the execution tracing facility.

PICL assumes a host-node programming model where the user has to use the processor designated as the host to access the other processors (nodes). The high-level routines, which are built on top of the low-level routines, are global communication functions that are useful in the development of parallel algorithms and application programs. The high-level routines are designed to run on various network topologies so the user can take advantage of the physical interconnection network and algorithm characteristics.

When the user requests execution tracing, code is activated within PICL routines in order to produce time-stamped records detailing the course of the computation on each processor. There are three distinct types of trace records generated: event, computation statistics, and trace message. With this data the user can evaluate the performance of the code and locate possible performance bottlenecks.

3.5.1 PICL Implementation of Multi-Target Tracker

For this implementation of the tracker we list the calls corresponding to these statements for the host and node templates. The changes needed for the PICL implementation are shown in Figure 3.10. We tested the PICL implementation of the Tracker on cluster of workstations(SUN4) and the Intel iPSC/860. Referring to Figure 3.10, Statement 1 in the Host Template initiates the host program. Statement 2 in the Host Template loads all the node programs. The number of node programs loaded is specified by *nprocs*. This is complemented by Statement 1 in the Node Template. Statement 2 in the Node Template gets the node environment information with the help of the call *setarc0()*. Statement 3 in the Host Template is not required because this statement is used to broadcast node information to the nodes. However in this case Statement 2 of the Node Template can access this information directly. Statement 3 in the Node Template is at the end of the node program. This is a dummy send message (*send0()* and is complemented by a receive message *recv0()* in the Host Template. After the host program receives this message from all the nodes, it terminates as shown in Statement 5 in the Host Template. Similarly the node program also terminates after sending this message as shown in Statement 4 in the Node Template.

3.5.2 Performance of the Concurrent Tracker implemented using PICL

Changes to the Host Template

Statement 1 : open0(&nprocs, &me, &host);

Statement 2 :

```
load0("picl_node", -1);
setarc0( &nprocs, &top, &ord, &dir);
```

Statement 3 : This statement is not required in PICL. The node program gets the node environment using the setarc0() call in the node program.

Statement 4 :

```
for (i=0; i<nprocs; i++)
recv0(mesg_no, sizeof(int), i);
```

Statement 5 : close0(1);

Changes to Node Template

Statement 1 : open0(&nsproc, &procnum, &host);

Statement 2 : setarc0(&nsproc, &top, &ord, &dir);

Statement 3 : send0(&mesg_no, sizeof(int), pc_corn, host);

Statement 4 : close0();

Figure 3.10: Changes to Host Program and Node Program Templates for PICL Implementation

Table 3.7: Performance of Main Tracking Task on iPSC/860 using PICL (8 Sites, 640 targets)

Tracking Task	1 Node	2 Node	4 Node	8 Node	16 Node
FP Tracking Summary					
FP Track Extension	176.87	160.50	119.00	60.13	24.37
Track Redistribution		6.38	17.63	35.75	37.81
FP Report Construction	11.88	8.87	8.66	9.94	8.36
FP Track Initiation	7.50	5.12	4.13	7.74	3.5
FP Tracking Total	196.25	180.87	149.42	113.56	74.06
3D Tracking Summary					
3D Track Extension	0.50	0.38	0.62	0.50	0.62
Report Association	39.62	57.88	41.75	37.50	28.25
3D track Initiations	1.62	1.62	1.12	1.62	1.25
Trajectory Estimation	5.62	4.00	2.38	2.12	2.00
Total 3D Tracking Task	47.36	63.88	45.87	41.74	32.12
Grand Total	243.61	244.75	195.29	155.30	106.18

Table 3.7 shows the timing for the concurrent tasks of the Multi-Target Tracker implemented using PICL on an iPSC/860. Table 3.8 shows the speedup and efficiency of main tracking tasks.

Table 3.8: Multiple Tracking Task on iPSC/860 using PICL (8 Sites, 640 targets)

Tracking Task	2 Node		4 Node		8 Node		16 Node	
	speedup	efficiency	speedup	efficiency	speedup	efficiency	speedup	efficiency
FP Tracking Summary								
FP Tracking Extension	1.10	55.00%	1.49	37.27%	2.94	36.75%	7.26	45.36 %
FP Report Construction	1.34	67.00%	1.37	34.25%	1.19	14.94%	1.42	8.86 %
FP Report Initiation	1.46	73.00%	1.82	45.50%	0.97	12.11%	2.14	13.39 %
FP Tracking Total	1.09	54.50%	1.31	32.75%	1.73	21.60%	1.72	10.8 %
3D Tracking Summary								
3D Track Extension	1.32	66.00%	0.81	20.25%	1.00	12.50%	0.81	5.04 %
Report Association	0.69	34.50%	0.95	23.75%	1.06	13.21%	1.40	8.76 %
3D track Initiations	1.00	50.00%	1.45	36.16%	1.00	12.50%	1.30	8.1 %
Trajectory Estimation	1.41	70.50%	2.36	59.00%	2.65	33.14%	2.81	17.56 %
Total 3D Tracking Task	0.74	37.00%	1.03	26.25%	1.13	14.18%	1.47	9.21 %
Grand Total	0.99	49.50%	1.25	31.25%	1.75	19.63%	2.29	14.31 %

Figure 3.5.2 shows the *Speedup* versus *Number of Processors* plot for these tasks.

Near-linear speedup for the Multi-Target Tracker was obtained for this *machine-tool* combination. We observe in Figure 3.5.2 that the reduction in execution time or speedup is proportional to the increase in the number of nodes.

3.6 p4 : Portable Programs for Parallel Processors

p4 [17] is a library of macros and subroutines developed for programming a variety of parallel machines, networks of workstations and single shared-memory multiprocessors. It was developed at Argonne National Laboratory.

It supports the following basic computational models:

- Shared memory model (monitors).
- Distributed memory model (message passing).
- Combination of the above two models.

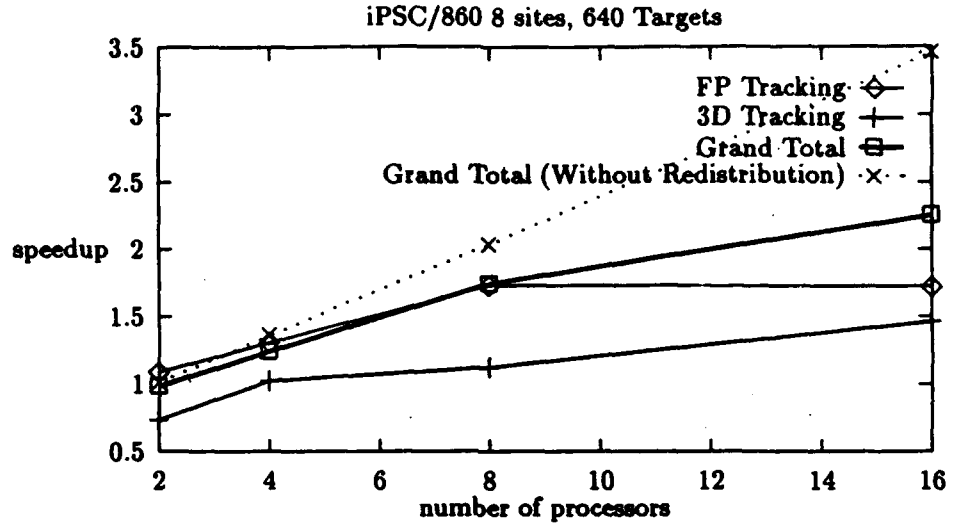


Figure 3.11: Plot of Main Tracking Task in PICL

For the shared memory model of parallel computation, p4 provides a set of primitives from which monitors can be constructed, as well as a set of useful monitors. For the distributed memory model, p4 provides *send* and *receive* operations and creation of processes according to a process group file describing files to be executed and the machines to execute them.

p4 ensures easy portability to a wide range of architectures/platforms. The p4 functions are simple and easy to comprehend. p4 does not require presence of daemons on machines before any application can be run, unlike many other tools. p4 offers features for automatic and user supervised process creation and message passing. p4 allows creation of multiple processes/nodes on a single machine.

Other features of p4 are summarized below:

- Support for heterogeneous networks.
- SYS V IPC for several architecture types.
- Debugging facility for automatic logging/tracing.
- Automatic or user-supervised process creation and message passing.
- High-resolution timing functions for several architectures.
- Error and interrupt handling.

Changes to the Host Template for p4 implementation

Statement 1 : p4_initenv(&argc,argv);
Statement 2 : p4_num_total_ids();
Statement 3 :
This statement is not required in p4. The node program gets the node environment by calling the function p4_num_total_ids(), itself.
Statement 4 : p4_wait_for_end();
Statement 5 : Not required in p4. p4_wait_for_end() function takes care of clean up.

Changes to Node Template for p4 implementation

Statement 1 : This is not required in p4. Host program get the information about nodes from Process Group file and assign them unique IDs.
Statement 2 : p4_num_total_ids();
Statement 3 : Remains the same as in the node template.
Statement 4 : p4_wait_for_end();

Figure 3.12: Changes to Host Program and Node Program Templates for p4 Implementation

p4 supports a set of *send* and *receive* procedures for communicating between nodes. These procedures depend on a lower-level set of procedures that handle local and network communication. They are transparent to the fact that they have to travel through a network or shared memory or any other such mechanism. p4 provides both blocking and non-blocking procedures for sending messages. p4 provides broadcast facility to all processes. p4 also provides routines for a variety of *global operations* (add, multiply, global maximum, global minimum). Since the order in which the nodes apply the operations is not strictly defined, the operation must be commutative.

3.6.1 Implementation of Multi-Target Tracker using p4

The changes needed for the p4 implementation are shown in Figure 3.12. Statement 1 in the Host Template initializes the p4 system and allows p4 to extract any command line arguments passed to it. Statement 2 in the Host and Node Templates will get the total number of id's started by p4 in all clusters. Statement 3 in the Host Template is not required for p4 implementation. Statement 3 in the Node Template is the Topology mapping routine which is used by the nodes for communicating between themselves. Statement 4 in the Host and Node Template wait till the end of program and take care of clean up when the program terminates.

3.6.2 Performance of the Concurrent Tracker implemented using p4

Table 3.9 shows the timing for all concurrent tasks of the Multi-Target Tracker implemented on IBM RS/6000. In Table 3.10 we have been able to test the tracker for upto 8 nodes on this platform. Some routines have shown speedup with an increase in the number of nodes while others have not. For example *Track Extension* in Focal Plane Tracking shows a speedup while execution time increases for 3D Track Extension with an increase in the number of nodes. *Track Redistribution* shows a significant increase in execution time. This clearly indicates areas for improvement in tracker code. However, we observed that other tasks which do not show significant speedup do not takeup a big percentage of total tracking task.

Table 3.9: Performance of Main Tracking Task using p4 on IBM RS/6000 (8 Sites, 640 targets)

Tracking Task	1 Node	2 Node	4 Node	8 Node
FP Tracking Summary				
FP Tracking Extension	116.33	120.39	60.60	34.94
Track Redistribution		33.73	143.06	262.89
FP Report Construction	8.59	6.10	8.87	11.67
FP Report Initiation	1.41	1.86	4.00	3.93
FP Tracking Total	126.33	162.08	216.53	313.43
3D Tracking Summary				
3D Track Extension	0.28	0.35	0.35	0.42
Report Association	20.50	30.30	26.91	22.25
3D track Initiations	1.06	0.84	0.45	0.27
Trajectory Estimation	1.64	1.56	1.05	0.95
Total 3D Tracking Task	23.48	33.05	28.76	23.89
Grand Total	139.81	195.13	255.29	337.33

Table 3.10: Performance of Main Tracking Task using p4 on IBM RS/6000 (8 Sites, 640 targets)

Tracking Task	2 Node		4 Node		8 Node	
	speedup	efficiency	speedup	efficiency	speedup	efficiency
FP Tracking Summary						
FP Track Extension	0.97	48.32%	1.92	48.00%	3.33	41.62%
FP Report Construction	1.41	70.04%	0.97	24.15%	0.74	9.21%
FP Track Initiation	0.76	37.83%	0.58	14.50%	0.36	4.45%
FP Tracking Total	0.77	38.1%	0.58	14.50%	0.40	5.00%
3D Tracking Summary						
3D Track Extension	0.80	40.00%	0.80	20.00%	0.67	8.33%
Report Association	0.68	33.82%	0.76	19.05%	0.92	11.51%
3D track Initiations	1.26	63.09%	2.35	58.85%	3.93	49.07%
Trajectory Estimation	1.05	52.88%	1.56	39.04%	1.73	21.57%
Total 3D Tracking Task	0.71	35.55%	0.82	20.43%	0.98	12.30%
Grand Total	0.72	36.00%	0.55	13.69%	0.41	5.18%

Figure 3.6.2 shows the plot for *Speedup* versus *Number of Processors*. It is observed that tracker shows speedup only for *FP Track Extension* and *3D Track Initiation* and *Trajectory Estimation* tasks. From the graph it is observed that the performance of Tracker degrades with an increase in number of nodes because *Track Redistribution* consumes an appreciable percentage of the total execution time due to communication overhead. This can be explained by the fact that concurrent processing on the nodes decreases as the number of nodes is increased. However, when *Track Redistribution* is not considered in calculating overall speedup, a speedup is seen. This is shown in Figure 3.6.2. As mentioned earlier, this decrease in speedup indicates where the tracker code is not very efficient.

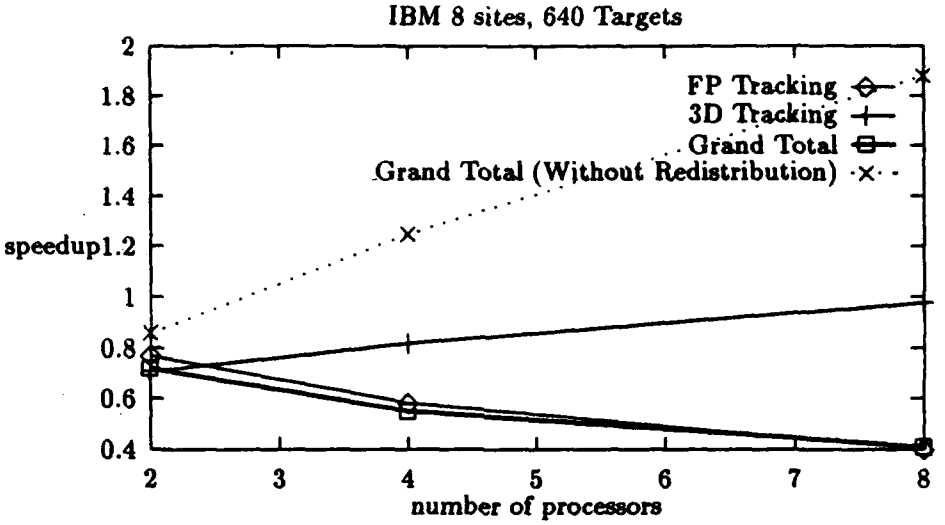


Figure 3.13: Plot of Main Tracking Tasks in p4

3.7 Performance Comparison

The current version of the Concurrent Multi-Target tracker which has been implemented by us on different architectures, with Express, PVM , PICL and p4 is not very efficient. It does not scale well in a distributed programming environment because the implementation has not been tuned for each concurrent task in this environment. Some of the tasks which show speedup with the increase in the number of nodes for all the tools are:

- Focal Plane Track Extension
- Focal Report Construction
- 3D Track Extension
- 3D Track Initiation and
- 3D Trajectory Estimation.

It can however be seen that Track Redistribution does not show any speedup. In fact as the number of nodes increase the percentage of time required by this task increases. At the same time the percentage of time required by other tasks decrease. This makes Track Redistribution a major contributor in time taken by the Tracker, which accounts for a large percentage decrease in the overall speedup. This indicates that the tracker has to be implement track redistribution more efficiently. For this reason in the plot of Main Tracking tasks of the tracker (e.g, Figure 3.6, 3.8, 3.9) we have also plotted the the Grand Total of Focal

Plane Tracking Tasks and 3D Tracking Tasks without Track Redistribution. There are also other areas such screen I/O which effect the performance of the tracker. The initial implementation of the tracker printed out timing details after each scan. The code was modified to print timing results only at the end of the last scan. We observed that this increased the performance of the tracker. Comparing the performance results of the tracker implementation on all the tools we observed that the performance of the Multi-Target Tracker was better on the nCUBE and Intel iPSC (see Express and PICL Performance results, Tables 3.1 and Table 3.7). The timings on these platforms were faster than the implementations on worstations by a factor of 2 (obtained by comparing Grand Total entries for 4 nodes on Tables 3.1, 3.5 and 3.7). It can be seen from Tables 3.3 and 3.5 that the PVM implementation of the tracker on RS/6000 cluster was faster than the heterogeneous implementation which used Rs/6000 and SUN4 workstations. It can also be seen that the plots of Grand Total timing for Main Tracking Tasks excluding Track redistribution show almost a linear speedup for all the tools.

Chapter 4

Tool Evaluation Methodology

In this chapter we develop a set of criteria to evaluate and benchmark tools used to develop parallel/distributed software systems and applications. The steps for tool evaluation are quantified and a tool evaluation methodology is presented.

4.1 Motivation

In parallel processing, a lot of work has been done in hardware development whereas software development has not been able to keep pace with it. We have different architectures like "shared memory " and "distributed memory" hardware architectures. However, we do not have a uniform environment for application development on these architectures. Often the software is tied to the architecture of the machine because the application typically uses specific communication primitives for that particular architecture. This results in a non-portable application which may have to be rewritten to be ported to any other architecture. The parallel/distributed software development process is shifting from homogeneous environments to heterogeneous environments. This calls for a uniform development environment to write applications that are completely portable. Such an environment can also provide transparency across different architectures.

All the tools we used seem to have been developed with specific applications in mind and attempts to generalize the tool were made at a later stage in the software development cycle. In this project we implemented the Multi-Target Tracker on four different tools: Express [16], PVM [14] , PICL [15] and p4 [17]. All the tools which we used to implement the tracker had their own strengths and weaknesses. While they were suitable for certain application development requirements, they were not suitable for some other cases. Therefore we had to develop our own methodology for porting the tracker to the different platforms supported by the tools studied. As an example, the tracker was originally written using a *hostless* programming model. Some of the tools we considered did not support this model. Therefore, we had to re-write the tracker using the host-node programming model. We observed, in the course of the project, that these tools work very well on some architectures and for a certain class of applications. However,

there were some necessary features missing in the tools we used. For example, the tools do not have a good debugger which is required in any software development process. We believe that a set of tool criteria listing the necessary features in any tool will help the overall software development process.

We envisage two ways in which the set of tool criteria will help the parallel/distributed software development process:

- They can be used for:
  1. **Evaluating** a tool with respect to other tools, and
  2. **Selecting** the best tool for a given class of applications.
- They can be used to **minimize or remove the deficiencies** in available versions of existing software tools, and also serve as valuable **inputs to tool developers.**

4.2 Proposed Approach for Tool Evaluation

Currently, there are no general criteria against which a given tool can be evaluated independently of the applications for which the tool is intended to be used; nor is it easy to lay down such a set. We believe that the criteria used to evaluate a tool, as well as the importance given to individual criteria, depends on the perspective of the person evaluating the tool. For instance, a programmer or actual user would lay more emphasis on the speed of execution of the communication primitives of the tool, because a good *response time* is important for the user. Consequently the user may give a grade of 4 (excellent or critically important, see Figure 4.1) for this criterion. On the other hand, it is likely that a manager would give more importance to *throughput* rather than response time in evaluating a tool, as the manager likes to have utilization close to unity (100 %). However, the response time increases exponentially towards infinity. This simple example shows that it is difficult to devise a universal set of criteria for evaluating a tool. Hence, our solution is to adopt a *layered* approach to the whole process : categorize a set of *levels*, and within each level, group related *criteria* that will be used to evaluate a tool. Each level is analogous to a perspective.

The levels are sufficiently general purpose so that any person evaluating a tool can identify the level(s) most important to him/her, and focus on the tools performance with respect to the parameters within that level(s). Furthermore, the criteria within a level, and the levels themselves, can be *prioritized* by the process of assigning *weights* to them. We will discuss more about assigning weights in the following sub-section on Methodology of Tool Evaluation. Finally, rather than having a static set of criteria for tool evaluation, the set of criteria is **extensible.** This will help the evaluator add/remove criteria depending on the application and/or his/her preferences. Using this philosophy, we have developed a tool evaluation *template* with maximum flexibility across tools, applications and evaluator perspectives.

We have identified four basic levels for tool criteria :

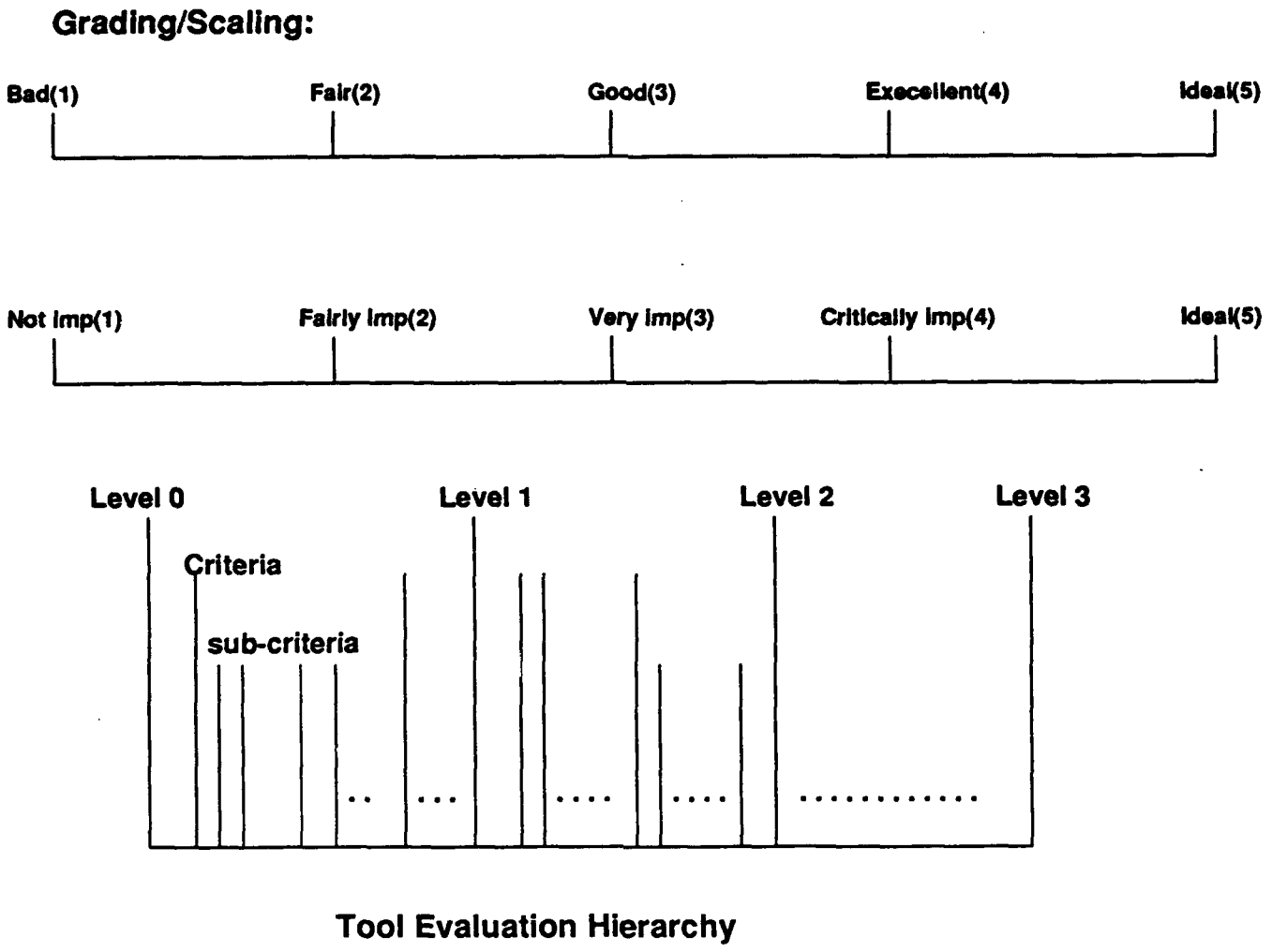


Figure 4.1: Grading Scheme and Tool Hierarchy

- **Level 0** Hardware/Software Requirements
- **Level 1** : Tool Capability
- **Level 2** : User Capability
- **Level 3** : Software Development Capability

Each level has been divided into different criteria and each criteria has been further sub-divided into sub-criteria (if any). We then define a tool evaluation process which could help in evaluating tools used for parallel/distributed computing. The tool evaluation process is explained using the Multi-Target Tracker as a running example. Having explained the overall approach, we now deal with the actual criteria that will be used for tool evaluation.

4.3 Tool Evaluation Criteria

The tool criteria presented below cover a broad spectrum of requirements. These requirements do not form an exhaustive list and can be extended by the users according to their own requirements. At the same time the user can also use a small *subset* of these requirements. The choice of criteria depends on user needs. Our intent is to present a frame work for benchmarking different tools used in parallel/distributed software development. The criteria we have listed below do not have to be in the same levels as shown below. The discussion below can be most effectively used as a guideline, rather than a hard-and-fast procedure for tool evaluation.

As we previously discussed, tool criteria can be classified into 4 levels. For each level, we list the important criteria. A criterion may or may not have sub-criteria. Each criterion and sub-criterion has to be graded on a linear scale (shown in Figure 4.1). Section 4.4 discusses in detail the tool evaluation process where different criteria are graded depending on the requirements of an application. In the following discussion we give a brief explanation alongside most criteria/sub-criteria and the concept of grading a criterion from different points of view is introduced.

4.3.1 Level 0: Hardware/Software Requirements

1. **Tool Cost**
   Many tools exist as public domain software, and so come free, while commercially available tools come at a monetary price. For a small organization planning to install a tool, tool cost may override all other considerations. If a tool does not easily fit into an organization's budget requirements, then it is likely that this criterion will get a low grade.
2. **Integration of the Tool into Existing Environment**
   Successful use of a tool requires a fit between the tool and the environment in which it will be used. The tool can be more effective if it is similar to the current environment/setup for an application.

The tool should run on the appropriate hardware and operating system. Tool installation should be a straight forward process. Data interchange, if required between the tool and other tools used by the application, should be straight-forward. If a tool is inexpensive but its use necessitates the acquisition of expensive hardware and/or software, then it is likely to get a poor grade for this criterion.

3. **Memory Requirement/Software Overhead**
The tool running on the user's hardware should be able to handle a development task of the size required by the user. For example the Multi-Target Tracker is a very memory intensive application. The tool should not add extra memory overhead like running huge background processes/daemons that will slow down the performance of the application. p4 and PICL do not have background processes, whereas Express and PVM have background processes which need to be running on each node to handle host-to-node communication and node-to-node communication. These daemons represent software overhead because they do not contribute to the useful computations required by an application. In this case p4 and PICL will receive a higher grade than the other two tools.

4. **Portability**
Portability is a very important feature that must be addressed by every tool. This helps to transport an application from one hardware/software environment to another with minimal changes to the application code. For example, topology mapping is a portability issue because a user should not be concerned about the underlying topology if the application has to be completely portable. A tool should support a wide range of topologies to transport the application to any parallel/distributed configuration. Express supports the grid decomposition topology which is not very elegantly supported by other tools. In this case Express will get a higher score as compared to other tools.

4.3.2 Level 1: Tool Capability

1. **Supported Platforms**
The tool should be able to support a wide range of computers like shared memory parallel computers, distributed memory multiprocessors and network-based computers. For example Express, PVM and PICL do not have versions that run on CM5 presently whereas p4 is supported on CM5. Hence p4 will receive a higher grade among these tools.

2. **Heterogeneous Processing Capability**
The tool should be able to support a single instance of an application across different architectures. This means that an application can be transparently executed on n nodes where all n nodes need not be of the same hardware architecture/machine. For example we used PVM to run a single instance of the Multi-Target Tracker on a cluster of SUN, IBM and HP workstations. Tools which do not support this capability will receive a low grade.

3. **Generality of the Tool Interface**
The tool should be designed to be used by more than one user at a time. The tool interface should be compatible with other tools in a tool set or other commercially available tools. There should be no extra overhead involved in terms of performance and programming when the tool interface is used. For example, a parallel application may have a user interface module which uses X-Windows or MS Windows ; at the same time it may use a parallel/distributed tool for parallel processing.

4. **Data mapping and Decomposition**
Data decomposition is done explicitly for applications implemented using existing parallel/distributed *tools*. A *tool should* be able to do this automatically and optimally for a given application. This helps to reduce the errors and load balancing problems that arise because of user controlled data decomposition. At present there are not many tools which provide this capability for the simple fact that data decomposition differs according to the application requirement and there seems to be no consensus on a standard methodology for this problem. All the tools we used will receive a very low grade for this criterion because this feature is not supported.

5. **Communication Services**
The communication primitives supported by existing libraries in a tool can be characterized as *point-to-point communication* and *group communication*.

   (a) **Point to Point Communication**
   This is the basic message passing primitive for any parallel/distributed programming tool. To provide efficient point-to-point communication most systems provide a set of function calls for *send* and *receive* primitives. These primitives can be synchronous (blocking) or asynchronous (non-blocking) primitives. All the tools we studied had this type of communication primitive and so this criteria will get the same grade for all the tools. However, if we measure the performance of running this communication primitive on each tool, the results will probably vary from tool to tool according to this measure.

   (b) **Group Communication**
   Group communication for many parallel/distributed computing environments can be further classified into three categories, *1-to-many*, *many-to-1*, and *many-to-many*, based on the number of senders and receivers. We now have a situation where we have criteria under the sub-criterion *Group Communication* , which then become sub-sub-criteria. This shows the extensibility of the tool criteria. All the three sub-sub-criteria categories do one form of global communication.

      - **1-to-Many Communication**
      Broadcasting and multicasting are the most important examples in this category. Some of the system libraries do not explicitly use a separate broadcast or multicast function call. Users should choose proper broadcast primitives according to the application.
      - **Many-to-1 Communication**
      In many-to-one communication, one process collects the data distributed across several processes. Usually such a function is referred to as a *reduction operation*. A global operation combine is an example of such a form of communication.
      - **Many-to-Many Communication**
      There are different types of many-to-many communication. The simplest example is the case where every process needs to receive the result produced by a reduction operation. From an implementation point of view, such an operation can be implemented by a many-to-one operation and a one-to-many operation. Communication patterns of many-to-many operations can be regular or irregular.

   (c) **Configuration Control and Management**
   The tasks of configuration control and management are quite different from system to system. A subset of the configuration control and management primitives supported by the tools that were used in this project are given below:

- allocate or deallocate one processor or a group of processors, e.g., *exopen/exclose, getcube/relcube*
- Load, start, terminate, or abort programs, e.g., *exload, exstart, abort.*
- Dynamic reconfiguration, e.g., *exgrid, gridinit.*
- Process concurrent or asynchronous file I/O, e.g., *fmulti/fsingl.*
- Query the status of the environment, e.g., *exparam, pstatus.*

4.3.3 Level 2: User Capability

1. **Language Support**
   The number of languages supported or support for a specific language can be important parameters in selecting and/or evaluating a tool. For example, all the tools we used in this project support C and Fortran. An application requirement may be such that different parts of an application may have to be written in different languages. For example, if a tool directly interfaces with device drivers, then some part of the code may have to be written in a lower-level language. In such a case a tool that supports a wide range of languages will receive a higher grade.

2. **Ease of Programming**
   One measure of a tools effectiveness is the ease with which the user can interact with it. If the user spends more time thinking about how to use the tool or making the tool work, then the tool is hindering and not helping with the task. To justify using a tool, *the tools benefit must offset its cost* and the time spent using it. To be more precise, we divide this criteria into the following sub-criteria.

   (a) **Tailoring the Software**
   A tool can be utilized for a wide variety of uses. If the tool can be tailored to user needs or to a particular user style, the [illegible] as the potential to be used with more dexterity and at a faster rate. Some of the issues th[illegible] commonly addressed in tailoring the software are :
   - Allowing the user facilities to define new commands and macros and to chain macros together.
   - Allowing the user to reconfigure the tool to tradeoff between different resource parameters such as response speed and memory utilization.
   - Redefining the tool input and output format.

   (b) **Predictability**
   Unpredicted responses from the tool usually result in unsatisfied users and unwanted output. Command names should suggest function, and a user should rarely be surprised by a tools response. An example of the issues that arise when we consider tool predictability is the ability to predict or expect the response from the tool in most cases. For example, if a tool is configured to run on a maximum of 8 nodes and if a user tries to run it on 16 nodes, it is an error. In this case a predictable response expected from the tool would be an error message which informs the user that the number of nodes have been exceeded. An unexpected response would be if the program simply hangs.

   (c) **Error Handling**
   The tool should be able to gracefully exit when an unretrievable error occurs. In other cases

the error message should be a pointer to the type of error that has occurred. Protection from costly errors should be provided. For example, when the application requires more memory than what is available, it is an error condition. In this case the tool should give an appropriate error message, delete all allocated memory and exit the program, without causing the terminal to hang. All the tools that we used in this project do not have a mature error/exception handling feature and hence, will receive a low grade for this criteria.

3. **Programming Models Supported**
   The different programming models supported by parallel/distributed tools are CUBIX/hostless model and host-node programming model. Express supports both models. Hence Express will get a higher grade as compared to other tools we used in this project.

4. **Interoperability**
   An important aspect of a tool that is often overlooked is the ease with which a tool can be incorporated into any development environment. The development community may have to accept changes that the tool may inflict upon the environment. We have not used this feature in our application; any feature that has not been used by the evaluator gets an average grade.

5. **Learning Curve**
   Depending on how complex it is, learning a tool can result in considerable expense, time and frustration. The tools command set should be consistent and understandable. The complexity of the tool should be proportional to the complexity of the application; i.e., the tool should be able to simplify a problem rather than complicate it. Prospective users should have sufficient background to successfully use the tool. The user should be able to do something very quickly to see what happens and evaluate the results without a long set up time. The tool should be based on a small number of easy to understand/learn concepts that are clearly explained. The number of functions provided should be small enough to do the work the tool is intended to do. Finally, the time required to understand and become proficient in using the tool should be acceptable to an average user and a project team. Grading this criteria is also related to the experience of the user.

6. **Run Time Support for Parallel I/O**
   Since I/O is often *the* bottleneck in parallel programs, a tool providing run-time support for parallel I/O would be preferable to one which serializes I/O. For example, Express allows the user to switch screen I/O modes from multiple to single modes by using *fmulti* and *fsingl* function calls. In this case, tools which support different forms of parallel I/O will get a higher grade.

7. **Debugging Capability**
   The different functionalities that can be used as a basis for evaluating the debugging capabilities of a tool are :

   (a) **Tracing** : The tool should provide a capability to include function calls in the code which will help tracing the flow of execution of the application.

   (b) **Setting break points** : A user should be able to stop execution of a program at logical points by using this facility.

   (c) **Display values of different data types** : A debugger should be able to display values of all data types. For example some debuggers cannot display values of a user defined data type like a typedef structure, but can display the value of a string variable or an integer variable.

The tools we used in the project did not have debugging facilities and hence will receive a very low grade.

8. **Application Profiling**
An important aid in evaluating the performance of a parallel program is a profile of the program execution. This includes information, preferably visual, about the distribution of the work load on the processors, the percentage of time spent doing I/O and computation, and resource utilization. Such information provides valuable insight that helps in fine tuning an application to optimize application goals (e.g. : throughput, speedup, utilization). This also helps in identifying the bottlenecks in the application.

9. **Tool Robustness**
Robustness of a tool is a combination of such factors as the reliability of the tool, the performance of the tool under failure conditions, the criticality of the consequences of tool failures, the consistency of the tool operations and the way in which a tool is integrated into an environment. While the robustness of individual tools is important, it is secondary to the robustness of the environment in which the tool operates. Many characteristics of robust operation are dependent on a more global environment where the tool must have correct interfaces to the environment. For example, most tools need not be concerned about device interface (inter-networking in a parallel/distributed environment). These issues should be handled by the environment in which they are embedded. The tool should be concerned with having correct interfaces to be inserted in the environment and to operate properly with the environment. A few tool-related robustness issues are discussed below.

   (a) **Consistency**
   The tool should have well defined syntax and semantics. The tool should be able to operate in a system with unique identification of each node in the environment. For example, each node in an application instantiation should have a unique identification number. Some tools do not incorporate this feature consistently. For example, we expect that if 8 nodes are used for an application, they should be numbered from 0 to 7. PVM does not incorporate this feature consistently after abnormal termination of an application. In such a case the master pvmd daemon has to be killed and process of starting pvmds has to be done all over again. In this case PVM will receive a low grade.

   (b) **Evolution**
   Tools normally evolve over time to accommodate changing requirements, changes to the environment, corrections to detected flaws, and performance enhancements. Some of these issues can be listed as follows :

   - A tool should be built in such a way that it can evolve and retain compatibility between versions.
   - The tool should be able to smoothly accommodate changes to the environment in which it operates.
   - New versions of the tool must be able to interface with old versions of other related tools.
   - Separate versions of a tool should be able to coexist on the system.

   In the course of our project we tried to implement the Multi-Target Tracker on a newer version of PVM (from 2.4.2 to 3.0). However PVM (3.0) was not directly downward compatible and it will need some rework to achieve portability. In this case PVM will receive a lower grade.

(c) **Fault Tolerance**
There are many ways of defining fault tolerance. We are not concerned with the general problem, but with fault tolerance that specifically relates to individual tools. The various issues in fault tolerance are :

- The tool should have well-defined atomic actions. This means that no intermediate states should be registered, and that any environmental failures during execution of an application should not cause irreparable damage once the failure has been repaired and the system has been restarted.
- The tool should have the capability of roll-back recovery.

We have not used this feature in our project and hence this criteria will receive an average grade.

10. **Support and Maintenance**
A tool should have sufficient customer support after it has been installed. Moreover, the amount of bookkeeping prior to starting, during the execution and after the termination of a program must be minimal. For example, Express has background daemons which when killed do not release the semaphores. Sometimes even an explicit *exclean* does not do the job. In contrast, although PVM also maintains background processes called *pvmd's* , it kills all the slave pvmd's if the master pvmd is killed. More importantly, Express has better customer support than PVM, probably due to the fact that Express is available commercially while PVM is a public domain software.

11. **Performance of Tool Services**
The performance of a tool can greatly affect the ease with which it can be used. Poor tool performance can create costs that negate many of the benefits realized from tool use. A tool performance should be acceptable or relative to the complexity of operations. An application must not run for unreasonable amounts of time. When the tool supports multiple users the response time must be acceptable with the maximum user load. The tool should be able to dispose of any useless byproducts it generates in the process of executing an application. As explained earlier, the specific sub-criteria used to evaluate the performance of a tool depends on the application characteristics, the goals of tool evaluation and the perspective of the evaluator. Some of the factors that are likely to be considered are :

(a) **Response time** under different load conditions.

(b) **Communication Speed** if the anticipated applications are communication-intensive.

(c) **Throughput** if optimal resource utilization is the dominant aim.

(d) **Compilation time** if large applications consisting of hundreds of programs are to be supported by the tool.

We have used the performance data of the Multi-Target Tracker for different tools on a single architecture to grade this criteria.

4.3.4 Level 3: Application Development Capability

Application development capability is divided into a set of stages which correspond to phases typically encountered in the software development process. These stages can be described as follows

1. **Application Specification Stage**
   This is the input to the software development process. It is generated from the application description and specification which is generated from the application itself (if it is a new problem) or from existing sequential code (porting of dusty decks). A tool will be evaluated with respect to its capability to assist the user in developing an application at this stage. Most of the currently available software tools do not have support for such an environment.

2. **Application Analysis Stage**
   A tool should assist the user perform the following functions at this stage :

   - Module creation problem. This means identification of tasks which can be executed in parallel.
   - Module classification problem, i.e., identification of standard modules.
   - Module synchronization problem, i.e., analysis of mutual interdependencies. This stage corresponds to the design phase of the software life-cycle models and its output corresponds to the design document.

3. **Parallelization Specification Stage**
   A tool for developing parallel/distributed software should provide powerful primitives to allow the developer to easily parallelize the functional modules of an application.

4. **Application Development Stage**
   A tool will be benchmarked against its ability to support the following functions :

   - Algorithm Development Module
     Assists the developer in identifying the functional components in the parallelization specification and selecting appropriate algorithmic implementations.
   - System Level Mapping Module
     Uses the information provided by the algorithm development module to appropriately map the functional component to a suitable computing element in the environment.
   - Machine Level Mapping Module
     Performs the mapping of the functional components onto the processor(s) of the computing element to which it has been allotted by the system level mapping module.
   - Implementation/Coding Module
     Handles all code generation and performs code filling of selected template so as to produce parallel code which can then be compiled and executed.
   - Design Evaluator Module
     Assists the developer in evaluating different options available to each of the other modules, (e.g., algorithms, implementations, system level mappings, machine level mapping)

5. **Compile-Time/Run-Time Stage**
   The tool should support the task of executing parallelized specification generated by the development stage to produce the required output.

6. **Evaluation Stage**
   A tool should assist the developer to retrospectively evaluate the design choices made during the design process and look for ways to improve the performance.

7. **Maintenance/Evolution Stage**
   A tool should provide the capability of monitoring the operation of the software and ensuring that it continues to meet its specifications.

Having presented the general tool criteria, we now propose a methodology for tool evaluation which systematically grades the tool criteria we discussed above.

4.4 Methodology for Tool Evaluation

The tool evaluation approach follows a five step process which can be categorized as follows:

1. **Assign criteria to the each level.**
   It is important to assign criteria to the appropriate levels. This process depends on the ultimate goal of tool evaluation. The tool may be evaluated for a specific application, or can be compared with other tools.

2. **Identify sub-criteria (if any) in all criteria that have been listed.**
   A criteria for a tool may be a very broad based definition for a particular requirement. By classifying it into sub-criteria it may be possible to convey the actual requirement of that criteria for tool evaluation.

3. **Define a scaling/grading mechanism for sub-criteria, criteria and levels.**
   We believe that scaling/grading is the most important part of the evaluation process. In this step we make an attempt to quantify abstraction. Figure 4.2 shows an example of this step. It can be seen that scaling/grading can be made in only a subjective manner. In order to quantify these details we assign numerical values (*grades*) to these subjective details. The values range from 1 to 5 (see Figure 4.1) depending on the degree of importance of that level/criterion/sub-criterion. A grade of 5 is rarely assigned because this is termed as an ideal situation. This helps us to identify a normalization mechanism. The grades of sub-criteria are normalized against unity (see Figure 4.2) and the result is assigned as the grade for the associated criterion.

4. **Assign weights to criteria and levels.**
   After having assigned a grade value to each sub-criteria, we get the effective grade of all criteria for all levels. However, criteria may not be divided into sub-criteria in which case we directly assign grades to such criteria. Since criteria are interdependent, their importance in that tool has to be evaluated using a *weighting mechanism* which assigns weights ranging from one to five. A higher weight tends to show the increased importance of the criteria for the tool/application. Having calculated weights for all criteria we multiply the grades of criteria with their weights and then normalize these value against one. These normalized values are assigned as grade values to the associated levels (see Figure 4.2).

5. **Calculate percentage acceptance of the tool.**
   Having calculated the grades for each level we again assign weights to different levels as described in the previous step. The same process of normalization used for criteria values is employed to get

a normalized value of all levels (see Figure 4.2). This normalized value of levels is defined as the *measure of acceptance* of the tool for a specific requirement.

The next section shows a mathematical representation of the method. Finally, we explain the algorithm using a running example.

4.4.1 Tool Evaluation Algorithm

Step 1: Determine the grade $G(C_j)$ for each criterion C_j

$$G(C_j) = \sum_{i=1}^{m} \frac{G(Sc_i)}{m \times p} \quad (4.1)$$

if $m > 0$ for C_j, *where*

$G(Sc_i) \rightarrow$ the grade of sub-criteria Sc_i of criteria C_j,
$G(C_j) \rightarrow$ the grade of criteria C_j,
$m \rightarrow$ total number of sub-criteria for criteria C_j, and
$p \rightarrow$ maximum (ideal value) for any sub-criteria Sc_i in criteria C_j

This summation is performed only if a criteria c_j has any sub-criteria Sc_i, otherwise, the criterion grade is assigned a value between 0 and 1, after which the evaluation proceeds to step 2. For example, in Figure 4.2, the criterion *Tool Robustness* does not have any sub-criteria, so it is directly assigned the grade 0.5.

Step 2: Determine the weighted grade ($WG(C_j)$) of a criteria C_J

$$WG(C_j) = G(C_j) \times w(C_j), where \quad (4.2)$$

$w(C_j) \rightarrow$ the weight associated with criteria C_j

Step 3: Determine the normalized grade of each level

$$NG(L_k) = \frac{\sum_{i=1}^{n} WG(C_i)}{\sum_{i=1}^{n} w(C_i)}, where \quad (4.3)$$

$L_k \rightarrow$ any level k ,
$NG(L_k) \rightarrow$ the normalized grade of level k ,
$C_i \rightarrow$ any criteria in level k,
$WG(C_i) \rightarrow$ the weighted grade of $C_i$in level k,
n $\rightarrow$ total number of criteria for level k,
$w(C_i) \rightarrow$ weight associated with criteria C_i,

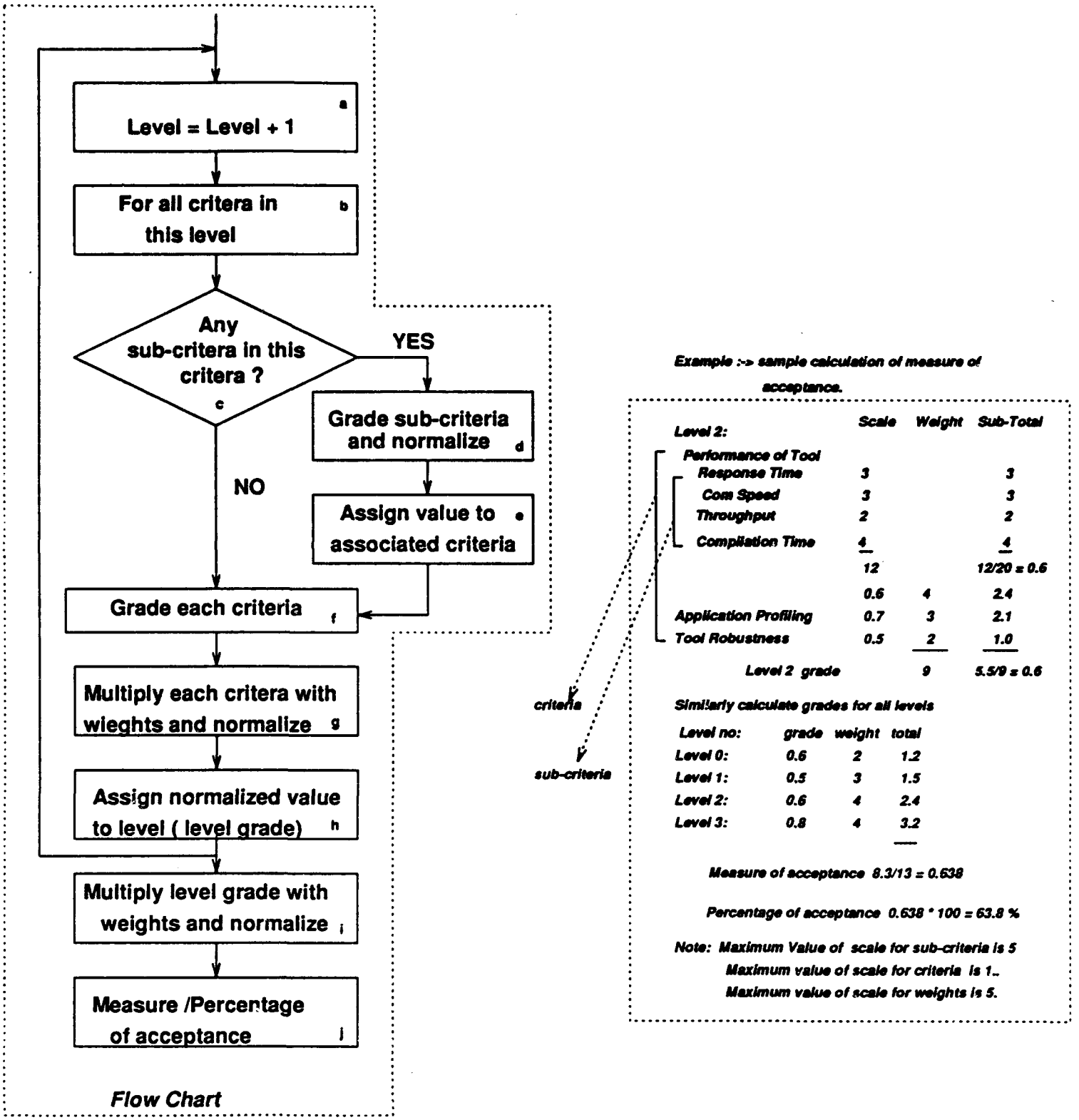


Figure 4.2: Tool Algorithm and Implementation Example

Step 4: Determine the Tool Acceptability Metric (the *Measure of Acceptance (MOA)*

$$MOA = \frac{\sum_{i=1}^{k} NG(L_i) \times w(L_i)}{\sum_{i=1}^{k} w(L_i)}, where \tag{4.4}$$

$w(L_i) \rightarrow$ weight associated with level L_i
$k \rightarrow$ total number of levels
$MOA \rightarrow$ Measure of Acceptance of the tool

The *Percentage Measure of Acceptance of the tool is* 100 times the MOA.
Figure 4.2 shows a flow chart of the tool evaluation algorithm.

4.4.2 Step-by-Step Explanation of Tool Evaluation Process

The flow chart in Figure 4.2 shows a systematic approach to the Tool evaluation methodology we described in the previous sections. For each level, (*block 'a' in the flow chart*), consider one criterion at a time. For each criterion (*block 'b' in the flow chart*), if there are any sub-criteria (*block 'c' in the flow chart*), then use equation (1) which grades each sub-criterion and assigns the normalized value to the associated criterion. For example Figure 4.2 shows the flowchart of the method and a snapshot during the calculation of the measure of acceptance.

With reference to the latter, it can be seen that level 2 has a criterion *Performance of Tool Services* which has sub-criteria (a) *Response time* (b) *Communication Speed* (c) *Throughput* and (d) *Compilation time*. Here *Response time* is assigned a grade/scale of 3, *Communication Speed* is assigned a grade/scale of 3, *Throughput* is assigned a scale of 2 and *Compilation Time* is assigned a scale of 4. The scales of all these sub-criteria have been added together which results in a cumulative grade/scale of 12. This is then normalized using the maximum attainable grade by each sub-criterion which in this case is 5. The total maximum attainable grade for all sub-criteria adds up to 20. Therefore the normalized value is 12/20 = 0.6. This value is assigned as the grade of the associated criterion (*block 'd' and 'e' in the flow chart*) which in this case is *Performance of Tool Services* .

Following the flowchart further we can see that if there are no sub-criteria associated with a criterion then each criterion is graded (*block 'f' in the flow chart*) in the same way as we graded sub-criteria above. The only difference is that each such criterion is directly graded on a scale of 0 to 1, i.e., the maximum attainable value by such a criterion is 1. In addition to this, each criterion is assigned a weighted value which reflects its relative importance in that level. The maximum attainable weighted value for any criteria is 5. For example in Figure 4.2 we can see that in Level 2, *Application Profiling* and *Tool Robustness* do not have sub-criteria. Hence, they are directly given a grade of 0.7 and 0.5, respectively. Their relative weights are 3 and 2 respectively. We then multiply each criterion grade with its associated weight (*block 'g' in flow chart*) and enter the value in a sub-total column as shown in Figure 4.2. These values are added together and are normalized against the sum of maximum attainable values by each criterion (*block 'g' in flow chart*). The maximum attainable value by any criterion is the weight of the criteria itself. For example, in Figure 4.2 the maximum attainable value by *Application Profiling* is 1 x 3 = 3, which is the

weight of the criterion itself. This is equivalent to equation (2) shown in the mathematical representation of the tool evaluation algorithm. Let us assume for the sake of explanation that Level 2 has only three criteria : *Performance of Tool Services*, *Application Profiling* and *Tool Robustness*. The normalized value adds up to 2.4 + 2.1 + 1.0 / (4 + 3 + 2) = 5.5 / 9 = 0.6 (see Figure 4.2). This step is equivalent to equation (3) in the mathematical representation.

As we follow the flow chart in Figure 4.2 we see that the value 0.6 is the level grade for Level 2 (*block 'h' in flow chart*). Each level grade is calculated in the same manner we described above and we have grades for Level 0, Level 1, Level 2 and Level 3.

The flow chart in Figure 4.2 shows the procedure for each level and the loop back to the block 'a', *level = level + 1*, is essential for the next level. The process of assigning weights to each of these of levels is done in the same manner (*block 'i' in the flow chart*) as we did it for each criteria in a level. In the example in Figure 4.2 we can see that the grades and associated weights for each level are multiplied and is assigned to a total column. These values are then normalized against the maximum attainable value by each level. The result in the example shown in Figure 4.2 is 8.3 / 13 = 0.638. This step corresponds to equation (4). This value corresponds to the normalized grade for the entire tool (*block 'j' in flow chart*) and is the *Measure of Acceptance (MOA)* of the tool. The *Percentage Measure of Acceptance* for any tool is then 100 times the MOA and for the example in Figure 4.2 it corresponds to 0.638 * 100 = 63.8 %.

It can be noticed that our approach follows a top down approach for defining and assigning criterion and sub-criterion at different levels; whereas it follows a bottom-up approach for assigning grades and weights to sub-criterion, criterion and levels. Our methodology of combining the top-down and bottoms-up approach is particularly powerful because it succeeds in quantifying abstraction.

In the absence of discrete metrics, this layered approach of progressively quantifying abstraction can be used to assess the suitability of a parallel/distributed software development environment for a users requirement. Using the system of grading and assigning weights to criteria and levels, the user/developer can tailor the methodology to reflect his/her priorities. This results in a powerful and flexible approach to tool evaluation, selection and development. In the next chapter we apply this tool evaluation methodology to develop tool evaluation benchmarks for the tools we used in this project.

Chapter 5

Tool Evaluation Benchmarks

The chapter on Tool Evaluation Methodology (chapter 4) discusses the different tool criteria that have to be taken into consideration for evaluating a tool. The HHPC software development model we presented in chapter 1 is a subset of this Tool Evaluation Methodology. As discussed in the previous chapter, tool evaluation criteria has 4 levels These levels are:

- Level 0: Hardware/Software Requirements
- Level 1: Tool Capability
- Level 2: User Capability
- Level 3: Application Development Capability

Evaluation of level 3 requirements for each tool shows how each tool is useful in implementing the HHPC software process development model presented in chapter 1.

In this chapter we present tool evaluation benchmarks for all the tools (Express, PVM, PICL, p4) used in this project and process evaluation benchmarks (level3) for the HHPC software model.

It is important to notice the benchmark results for the same set of tools can be different for a different application. The benchmarking process is dependent on the needs of the application. For example in this project our intention was to have a tracker implementation for every tool and on all architectures that are available at NPAC. We did not spend time on tuning the performance or enhancing the algorithms used in the tracker.

In section 5.1 we will apply the HHPC software development process model discussed in chapter 1 to the implementation of the tracker using the studied software tools. We will discuss the relevance of each module in this model and its relation to the tracker implementations. This discussion is useful to develop the process benchmarks or evaluation of level 3 criteria explained section 5.2. We will then develop tool evaluation benchmarks for each tool which also includes the process evaluation benchmarks(level3).

5.1 Mapping the HHPC software development model to the Multi-Target Tracker porting process

The main task in this project was to port this software to a combination of different architectures and tools. Each module in the HHPC software development model is considered in the order shown in Figure 1.2 . We explain the different factors taken into consideration while porting the Multi-Target Tracker with relation to this model. The output of each module acts as an input to the next module.

1. **Dusty Decks**
   We started the project with an existing version of the concurrent multi-target tracker written using Cros III primitives, which was developed for the Mark III Hypercube. This is represented by the Dusty Decks module shown in Figure 1.2.

2. **Application Specification Filter**
   The requirement of the project is represented by this stage (see Figure 1.2). The output of this filter for our project is as follows:

   - Port the existing Cros III version to Express, PVM, p4 and PICL.
   - Support as many hardware architectures as possible which support the tools mentioned above.
   - Benchmark Performance of the tracker for each machine-tool combination.
   - Evaluate the application development tools used in the project.

3. **Application Analysis Stage**
   This stage involves analyzing the application to satisfy the requirements of the application specification filter. The output of the application analysis stage for the project were as follows:

   - A clear picture of the structure of the existing software.
   - Physical requirements (e.g., hardware and software that will be used)
   - Conversion requirements which include
     (a) Identification of a programming model that is supported by all application development tools that we proposed to use. The host-node programming model was used in this project.
     (b) Identification of a systematic procedure to port the software to all the tools. This involves an external design phase which gives shape to the initial design approach to be used. Refer to the section on *porting the tracker on different tools* discussed in the chapter 3.
   - Specifications for timing all concurrent tasks implemented in the tracker.
   - Requirements and procedure for tool evaluation.

4. **Parallelization Specification**
   The parallelization specification stage in this project involved only identifying message passing primitives supported by all the tools. For example the primitives identified in express were (a) *exopen()* (b) *exload* (c) *exwrite* and (d) *exread* The output of this stage did not involve adding any new parallelization specification although areas for potential parallelism have been identified which can enhance the

performance of the tracker. For example, the implementation of the tracker requires each node of the system to maintain a global structure of the track database. Each node of the tracker works on sections of this database. This requirement of the tracker degrades the overall performance of the tracker due to increased memory requirements.

5. **Application Development Stage**
   This stage consists of a design evaluation process which would help us to implement the requirements produced by the application analysis stage. Referring to fig 1.2 we can see that there are 4 stages in the Design Evaluation Process. We will briefly discuss all these stages.

   - Algorithm Development
     Existing algorithms were not modified, nor any new algorithms were added in this project. This is an important stage when a new application is being developed or the existing application needs to be tuned to improve its performance. Tuning an application for performance improvement will involve modifying existing algorithms or using new efficient algorithms to replace existing algorithms when required.
   - System Level Mapping
     This stage was not used in this project.
   - Machine Level Mapping
     A gridmap interconnection topology onto a cartesian grid was used in this project. The different primitives that allow such a decomposition are (a) *gridinit()* (b) *gridcoord() and (c)* gridchan(). This topology is supported by Cros II primitives. In order to port the tracker to different tools these primitives were re-implemented using the C programming language. See section 3.1 on *Porting the tracker on different tools* in chapter 3.
   - Implementation/Coding
     This was the key stage where we developed a standard paradigm to enable a simple and general porting process. A detailed discussion of this process is given in chapter 3 (section 3.1 *Porting the tracker on different tools*). This stage was was developed for all the tools we used for porting the tracker (Express, PVM, PICL and p4).

   In addition to these stages header file definitions had to be modified for porting the tracker to different architectures. For example the Express implementation of the tracker needs the header file definitions of *express.h* and *cros.h*. The PICL implementation of the tracker needs declaration of a global integer *host*. These modifications have been incorporated by adding pre-processor statements in the header file track89.h and basic89.h. The timing/clock routine had to be changed on the iPSC/860 (changed from *clock()* to *mclock()*) for timing concurrent tasks in the tracker. The code was modified to report timing outputs only for the last scan of the tracker and cumulative concurrent timings of all previous scans. This reduced screen I/O and also helped to improve the overall performance.

6. The output of the Application Development Stage is a parallelized structure of the tracker which has been implemented using standard primitives of the studied tools.

7. **Compile-Time/Run-Time Stage**
   The compile and run time stage involves writing makefiles for each architecture and each tool. There

```
ARCH    = RIOS

PVM  =   $(HOME)/pvm
PLIB    =   $(PVM)/src/$(ARCH)/libpvm.a
EXECDIR =   $(PVM)/$(ARCH)
SDIR    =   .

CC      =   cc
CFLAGS  =   -g
LIBS    =   -lm

.c.o:
    $(CC) $(CFLAGS) -c  $(PLIB) $(LIBS) $<

OBJS = pvm_host.o

pvm_host:   $(OBJS)
            $(CC) $(CFLAGS) -o pvm_host $(OBJS) $(PLIB) $(LIBS)
            mv pvm_host $(EXECDIR)

default:    pvm_host
            @echo "Done with host make"
```

Figure 5.1: PVM Host Makefile

are two makefiles which we identify as a host makefile and a node makefile. For example the makefile for the PVM implementation of the tracker on a cluster of IBM/RS6000 workstations is as follows:

We require a separate host makefile and node makefile because on implementations of the tracker like the iPSC/860 the compilers for the host(*cc*) and node (*icc*) programs are different.

8. **The acceptance test stage**
   This stage is not explicitly shown in Figure 1.2 and it is assumed to be part of the compile and run time stage. This stage involves verifying the correctness of our implementation. We ensured correctness by cross verifying the results of the original implementation and the results of our implementation of the tracker.

9. **Evaluation Stage** The evaluation stage consists of verifying the specification requirements with the functionalities implemented by the software. One requirement of the project was to develop performance benchmarks. These benchmarks have been developed and explained in detail in Chapter 3. Another requirement was to develop different versions of the Multi-Target Tracker using Express, PVM, PICL and p4. These versions have been developed and explained in detail in chapter 3.

```
ARCH    = RIOS

PVM     =   $(HOME)/pvm
PLIB    =   $(PVM)/src/$(ARCH)/libpvm.a
EXECDIR =   $(PVM)/$(ARCH)
SDIR    =   .

CC      =   cc
CFLAGS  =   -g
LIBS    =   -lm

.SUFFIXES :
.SUFFIXES : .c .o .e .r .f .y .yr .ye .l .s

.c.o:
    $(CC) $(CFLAGS) -c -DPVM $(PLIB) $(LIBS) $<

OBJS   = *.o

SRCS   = *.c

pvm_node:   $(OBJS) $(PLIB)
            $(CC) $(CFLAGS) -o pvm_node $(OBJS) $(PLIB) $(LIBS)
            mv pvm_node $(EXECDIR)

default:    node
        @echo "Done With Node Program"

$(OBJS):    track89.h basic89.h
```

Figure 5.2: PVM Node Makefile

10. **Evaluation Stage**
    On successful completion of the evaluation stage we have different versions of the tracker implementation. This gives rise to evolution specification for future enhancements and hence a evolution stage in the software development cycle. We have not dealt with this stage in this project. As a result the end of the Evaluation stage marks the beginning of the Maintenance stage.

5.2 Tool Evaluation Benchmarks

We will use a tabular representation to explain the tool evaluation process. Here is a brief explanation of the layout of the Table. The evaluation proceeds from left to right and top to bottom (see Table 5.1- 5.3). The level number is entered in column 1. Criteria within a level are entered in column 2. If a criterion has sub-criteria, they are entered in the next column. If there are sub-criteria, each sub-criterion is assigned a grade from 1 to 5 (as explained earlier) in column 4. The grades of all sub-criteria are summed and entered in column 5. If there are no sub-criteria for a criterion, the criterion is directly assigned a grade from 0 to 1 in column 5. The values in columns 5 and 6 are used to get the normalized criterion grade in column 7. The weight for each criterion is entered in column 8. The value in column 7 is multiplied with that in column 8 to get the weighted criterion grade in column 9. The sum of all the weighted criterion grades for a level gives the level grade in column 10. The maximum level grade in column 11 is just the sum of the weight assigned to each criterion in column 8 with respect to the level under consideration. The values in columns 10 and 11 are used to calculate the normalized level grade in column 12 for each level. After assigning a weight to each level in column 13, the product of columns 12 and 13 gives the weighted level grade for each level in column 14. Finally, the overall acceptance for the tool in column 15 is calculated by dividing the sum of the weighted level grades in column 14 by the sum of the level weights in column 13.

5.2.1 Tool Evaluation Benchmarks for PVM

We now explain the evaluation process for PVM. Table 5.1 to 5.3 shows the evaluation. We used PVM 2.4.2 for this work. Also, columns 4, 5, 8 and 13 are most important from the point of view of user-interaction : the values for the other columns are dependent on these columns and can be automatically calculated by an interactive evaluation tool. Hence, we focus on columns 4, 5, 8 and 13 in the discussion below.

1. **Level 0 : Hardware/Software Requirements**
   All the criteria in this level do not have sub-criteria, hence columns 3 and 4 are empty for all of them. PVM got the ideal grade of 1 for *tool cost* because it is public domain software, developed by Oak Ridge National Laboratory. However tool cost was not important to us in our effort and hence we gave it the low weight of 1. PVM got a high grade of 0.8 for *integration of tool into existing environment* because installation was easy and it worked without too many hitches in our hardware and software environment. This feature was important to us, so we gave this criterion a high weight of 4. PVM got a low grade for *overheads* since running a program in PVM requires execution of daemon processes (*pvmd*) on all participating sites. *Portability* was one of the most important concerns in

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
0	Tool Cost	-	-	1	1	1	1	1						
	Integration of tool into existing environment	-	-	0.8	1	0.8	4	3.2						
	Overheads	-	-	0.6	1	0.6	4	2.4						
	Portability	-	-	0.8	1	0.8	5	4.0	10.6	14	0.75	4	3.0	
1	Platforms Supported	-	-	0.8	1	0.8	5	4.0						
	Heterogeneous Processing cabability	-	-	0.8	1	0.8	5	4.0						
	Generality of Tool Interface	-	-	0.8	1	0.8	5	4.0						
	Data Mapping and Decomposition	-	-	0.4	1	0.4	4	1.6						
	Communication Services	Point to Point Communication	3											
		Group Communication	3	6.0	10	0.6	4	2.4						
	Configuration Control and Management	-	-	0.8	1	0.8	4	3.2	19.2	30	0.64	5	3.20	
2	Language Support	-	-	0.8	1	0.8	3	2.4						
	Ease of Programming	Tailorability	3											
		Predictability	4											
		Error Handling	4	11	15	0.73	5	3.65						

Table 5.1: Example Evaluation of PVM

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
2 (cont)	Programming Models Supported	-	-	0.8	1	0.8	3	2.4						
	Interoperability	-	-	0.6	1	0.6	3	1.8						
	Learning Curve	-	-	0.8	1	0.8	5	4.0						
	Run Time support for Parallel I/O	-	-	0.4	1	0.4	3	1.2						
	Debugging Capability	-	-	0.4	1	0.4	3	1.2						
	Application Profiling	-	-	0.8	1	0.8	3	2.4						
	Tool Robustness	Consistency	3											
		Evolution	3											
		Fault Tolerance	3	9	15	0.60	5	3.0						
	Support and Maintenance	-	-	0.4	1	0.4	4	1.6						
	Performance of Tool Services	-	-	0.6	1	0.6	4	2.4	26.05	41	0.64	5	3.20	
3	Application Specification Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Application Analysis Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Parallelization Specification Capabilities	-	-	0.8	1	0.8	1	0.8						

Table 5.2: Example Evaluation of PVM (cont..)

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
3 (contd.)	Application Development Stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Compile - Time / Run - Time stage capabilities	-	-	0.8	1	0.8	1	0.8						
	Evaluation stage capabilities	-	-	0.8	1	0.8	1	0.8						
	Maintenance / Evolution stage capabilities.	-	-	0.6	1	0.8	1	0.6	4.8	7	0.69	1	0.69	0.6727
													Percentage Acceptance :	67.27 %

Table 5.3: Example Evaluation of PVM (cont..)

our effort, hence this criterion was given the highest weight of 5. PVM measured up well in this respect : we ported the tracker on IBM RS/6000s, SUN 4s and HP Apollo workstations using PVM. Hence, PVM was assigned a 0.8 grade for this criterion.

2. **Level 1: Tool Capability**
   PVM does support many different types of *Hardware Platforms*, (weight 5) hence the grade of 0.8. PVM also did well on *heterogeneous computers* (weight 5 again) : for example, we executed the tracker on a heterogeneous combination of SUN 4s and IBM RS/6000s. *Generality of tool interface* (weight 5) also got a high grade of 0.8 : PVM's interface is simple and easy to use and multiple users can use the tool. All tools used in our effort have little or no *data mapping and decomposition* capabilities , hence the low grade of 0.4. Both sub-criteria in *communication services* got relatively low grades of 3 for PVM because communication in PVM is a multi-instruction process. For example, to send a message, the send buffer must first be initialized (*initsend()*, then the message to be sent must be put in the buffer (*putnint()*), and only then can the message be sent by using the *snd()* call. Finally, PVM provides good *configuration control and management facilities* (weight 4), hence, a high grade of 4.

3. **Level 2 : User Capability**
   PVM supports C and Fortran, hence, it got a grade of 4 for *language support*. Extensive language support was not an issue for us hence, the weight 3. PVM supports two very general *programming models* : *tree computations* and *crowd computations*, of which most other programming models are subsets. Hence, the high grade of 0.8 for this criterion. We have not evaluated *interoperability* for any of the tools we used : in such a case, we assign an average grade of 0.6. For similar reasons, the weight assigned was also 3. Although the documentation for version 2.4.2 of PVM was sketchy, we found PVM fairly easy to learn. Hence, it got a good grade of 0.8 for *learning curve*; which was assigned a weight of 4, since we had to learn how to use 4 tools in the course of the project.

   The next three criteria were not too important for us, hence, they were given (relatively) low weights of 3. We found PVM to be a fairly *robust* tool. In version 2.4.2 of PVM, the master daemon has to be explicitly killed after termination of the master process. Moreover, we tried using PVM version 3 instead of version 2.4.2, but had a hard time using it. For instance, we could not get version 3 to work on HP Apollo workstations, although version 2.4.2 works fine on them. For these reasons, the sub-criteria in this criterion received average grades. PVM gave us reasonably good performance for the tracker, hence the average grade of 0.6 for *performance of tool services*.

4. **Level 3 : Application Development Capability or Process Evaluation**
   This level is also used for developing process benchmarks. We did not develop the original implementation of the tracker from scratch. However we have evaluated this level from a portability point of view using the Multi-Target Tracker as a running example. The benchmarks for this level are also called *process evaluation benchmarks* for PVM.

   The application specification stage and analysis stage are independent of the tool that we used in this project. Hence an average grade of 0.6 is assigned to both these criteria. The Parallelization specifications differ for each tool which essentially is the communication primitives provided by the tool. The primitives provided by PVM were easy to use and sufficient for our requirements, so we have given a grade of 0.8. The application development stage capabilities will remain the same for all

the tools because we developed a standardized procedure for porting the tracker to any tool. No extra work had to be done for any tool after this methodology was developed. Hence we have assigned an average grade of 0.6 for this criteria. The compile and run time stage capabilities have been assigned a grade of 0.8. The evaluation stage capabilities have been assigned a grade of 0.8. The maintenance and evolution stage gets an average grade of 0.6 because we have not used this stage in our project. It can be seen from column 12 that the normalized level grade is 0.69 which is equivalent to 69% acceptance for level 3 or *process evaluation grade* for PVM in this project.

The overall percentage tool acceptance which takes into account level 0-3 for PVM version 2.4.2 using our tool evaluation methodology is **67.27 %** and the process evaluation benchmark for PVM is **69%**.

5.2.2 Tool Evaluation Benchmarks for Express

We now briefly explain the evaluation process for Express. Table 5.4- 5.6 shows the evaluation. We used version 3.0 for this work.

1. **Level 0 : Hardware/Software Requirements**
   All the criteria in this level do not have sub-criteria, hence columns 3 and 4 are empty for all of them. Express got the ideal grade of 1 for *tool cost* since we got this tool free. However, since tool cost was unimportant to us in our effort, we gave it the low weight of 1. Express got an average grade of 0.6 for *integration of tool into existing environment* because express has to be configured through configurations file like express.cst, confile, netfile and plotfil. This is not a straight forward process and the inconsistent behavior of express (specially on nCUBE) made it more difficult. Express gets an average grade for *overheads* because it requires execution of daemon processes (*exinit()*) on all participating sites. *Portability* was one of the most important concerns for our effort, hence this criterion was given the highest weight of 5. We could port the express version of the tracker on SUN 4 workstations and nCUBE. Hence we assigned a grade of 0.6 for express in this criteria.

2. **Level 1: Tool Capability**
   Express did not support many different types of *Hardware Platforms*, (nCUBE and SUN workstations when we used Express, weight 5) hence the grade of 0.6. Express also did not do well on *heterogeneous computers* (weight 5 again), hence a grade of 0.6. *Generality of tool interface* (weight 5) also got an average grade of 0.6. All tools used in our effort have little or no *data mapping and decomposition* capabilities , hence the low grade of 0.4. Both sub-criteria in *communication services* got relatively grades of 4 for Express. Express does not provide *configuration control and management facilities* (weight 4), hence, an average grade of 0.6.

3. **Level 2 : User Capability**
   Express supports C and Fortran, hence, it got a grade of 0.8 for *language support*. Ease of Programming has been mostly been given an average grade except for error handling in Express which is not good. The express version of the tracker does not exit gracefully when there is an error. When the program is aborted by using the UNIX *kill* command express does not take of releasing semaphores. Express supports two very general *programming models* : *CUBIX* and *host-node* , of which most

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
0	Tool Cost	-	-	1	1	1	1	1						
	Integration of tool into existing environment	-	-	0.6	1	0.6	4	2.4						
	Overheads	-	-	0.6	1	0.6	4	2.4						
	Portability	-	-	0.6	1	0.6	5	3.0	8.8	14	0.63	4	2.52	
1	Platforms Supported	-	-	0.6	1	0.6	5	3.0						
	Heterogeneous Processing cabability	-	-	0.4	1	0.4	5	2.0						
	Generality of Tool Interface	-	-	0.6	1	0.6	5	3.0						
	Data Mapping and Decomposition	-	-	0.4	1	0.4	4	1.6						
	Communication Services	Point to Point Communication	4											
		Group Communication	4	8.0	10	0.8	4	3.2						
	Configuration Control and Management	-	-	0.6	1	0.6	4	2.4	15.2	30	0.51	5	2.55	
2	Language Support	-	-	0.8	1	0.8	3	2.4						
	Ease of Programming	Tailorability	3											
		Predictability	3											
		Error Handling	2	8	15	0.53	5	2.65						

Table 5.4: Example Evaluation of Express

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
2 (cont)	Programming Models Supported	-	-	0.8	1	0.8	3	2.4						
	Interoperability	-	-	0.6	1	0.6	3	1.8						
	Learning Curve	-	-	0.8	1	0.8	5	4.0						
	Run Time support for Parallel I/O	-	-	0.6	1	0.6	3	1.8						
	Debugging Capability	-	-	0.6	1	0.6	3	1.8						
	Application Profiling	-	-	0.8	1	0.8	3	2.4						
	Tool Robustness	Consistency	3											
		Evolution	3											
		Fault Tolerance	3	9	15	0.60	5	3.00						
	Support and Maintenance	-	-	0.4	1	0.4	4	1.6						
	Performance of Tool Services	-	-	0.6	1	0.6	4	2.4	26.25	41	0.64	5	3.20	
3	Application Specification Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Application Analysis Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Parallelization Specification Capabilities	-	-	0.6	1	0.6	1	0.6						

Table 5.5: Example Evaluation of Express (cont..)

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
3 (contd)	Application Development Stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Compile - Time / Run - Time stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Evaluation stage capabilities	-	-	0.8	1	0.8	1	0.8						
	Maintenance / Evolution stage capabilities.	-	-	0.6	1	0.8	1	0.6	4.4	7	0.63	1	0.63	0.5933
													Percentage Acceptance :	59.33 %

Table 5.6: Example Evaluation of Express (cont..)

other programming models are subsets. Hence, the high grade of 0.8 for this criterion. We have not evaluated *interoperability* for any tools we used : in such a case, we assign an average grade of 0.6 or 3 (as applicable) . For similar reasons, the weight assigned was also 3. The documentation for Express was good and we found Express fairly easy to learn. Hence, it got a good grade of 0.8 for *learning curve*; which was assigned a weight of 5.

The next three criteria were not too important for us, hence, they were given (relatively) low weights of 3. We gave a grade of 0.4 for support and maintenance in Express because we did not get any support from Parasoft Corporation when we wanted to port the Cros III version of Express.

Express did not give very good performance for the tracker, hence the average grade of 0.6 for *performance of tool services.*

4. **Level 3 : Application Development Capability or Process Evaluation**
   The application specification stage and analysis stage are independent of the tool that we used in this project. Hence an average grade of 0.6 is assigned to both these criteria. The Parallelization specifications differ for each tool which essentially is the communication primitives provided by the tool. The primitives provided by Express were not very easy to use although quite comprehensive, hence, a grade of 0.6. The application development stage capabilities will remain the same for all the tools because we developed a standardized procedure for porting the tracker to any tool. No extra work had to be done for any tool after this methodology was developed. Hence we have assigned an average grade of 0.6 for this criteria. The compile and run time stage capabilities have been assigned a grade of 0.6. The evaluation stage capabilities have been assigned a grade of 0.8. The maintenance and evolution stage gets an average grade of 0.6 because we have not used this stage in our project. It can be seen from column 12 that the normalized level grade is 0.63 which is equivalent to 63% acceptance for level 3 or *process evaluation* for PVM in this project.

The overall percentage tool acceptance which takes into account level 0-3 for Express version 3.0 using our tool evaluation methodology is **59.33 %** and the process evaluation benchmark for Express is **63%**.

5.2.3 Tool Evaluation Benchmarks for PICL

We now briefly explain the evaluation process for PICL. Table 5.7- 5.9 shows the evaluation.

1. **Level 0 : Hardware/Software Requirements**
   All the criteria in this level do not have sub-criteria, hence columns 3 and 4 are empty for all of them. PICL got the ideal grade of 1 for *tool cost* because PICL is a public domain tool developed at Oak Ridge National Laboratory. However, since tool cost was unimportant to us in our effort, we gave it the low weight of 1. PICL got of grade of 0.8 for *integration of tool into existing environment* because PICL is very easy to set up. PICL gets an good grade of 0.8 for *overheads* because it does not have any background processes to maintain. *Portability* was one of the most important concerns for our effort, hence this criterion was given the highest weight of 5. PICL measured up well in this respect : we ported the tracker on SUN4 workstations and iPSC/860 without any problems at all. Hence, we assigned a grade of 0.8 for PICL in this criteria.

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
0	Tool Cost	-	-	1	1	1	1	1						
	Integration of tool into existing environment	-	-	0.8	1	0.8	4	3.2						
	Overheads	-	-	0.8	1	0.8	4	3.2						
	Portability	-	-	0.6	1	0.6	5	3.0	10.4	14	0.74	4	2.96	
1	Platforms Supported	-	-	0.6	1	0.6	5	3.0						
	Heterogeneous Processing cabability	-	-	0.4	1	0.4	5	2.0						
	Generality of Tool Interface	-	-	0.6	1	0.6	5	3.0						
	Data Mapping and Decomposition	-	-	0.4	1	0.4	4	1.6						
	Communication Services	Point to Point Communication	3											
		Group Communication	3	6.0	10	0.6	4	2.4						
	Configuration Control and Management	-	-	0.6	1	0.6	4	2.4	14.4	30	0.48	5	2.40	
2	Language Support	-	-	0.8	1	0.8	3	2.4						
	Ease of Programming	Tailorability	3											
		Predictability	4											
		Error Handling	3	10	15	0.67	5	3.35						

Table 5.7: Example Evaluation of PICL

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
2 cont'd	Programming Models Supported	-	-	0.6	1	0.6	3	1.8						
	Interoperability	-	-	0.6	1	0.6	3	1.8						
	Learning Curve	-	-	0.8	1	0.8	5	4.0						
	Run Time support for Parallel I/O	-	-	0.4	1	0.4	3	1.2						
	Debugging Capability	-	-	0.4	1	0.4	3	1.2						
	Application Profiling	-	-	0.6	1	0.6	3	1.8						
	Tool Robustness	Consistency	4											
		Evolution	3											
		Fault Tolerance	3	10	15	0.67	5	3.35						
	Support and Maintenance	-	-	0.6	1	0.6	4	2.4						
	Performance of Tool Services	-	-	0.8	1	0.8	4	3.2	26.50	41	0.64	5	3.20	
3	Application Specification Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Application Analysis Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Parallelization Specification Capabilities	-	-	0.6	1	0.6	1	0.6						

Table 5.8: Example Evaluation of PICL (cont..)

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalised Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalised Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
3 (contd)	Application Development Stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Compile - Time / Run - Time stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Evaluation stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Maintenance / Evolution stage capabilities.	-	-	0.6	1	0.8	1	0.6	4.2	7	0.60	1	0.60	0.6107
													Percentage Acceptance :	61.07 %

Table 5.9: Example Evaluation of PICL (cont..)

2. **Level 1: Tool Capability**
   PICL does not support many different types of *Hardware Platforms*, (weight 5) hence the grade of 0.6. PICL also did not do well on *heterogeneous computers* (weight 5 again), hence a grade of 0.4. *Generality of tool interface* (weight 5) also got an average grade of 0.6. All tools used in our effort have little or no *data mapping and decomposition* capabilities , hence the low grade of 0.4. Both sub-criteria in *communication services* got relatively grades of 3 for PICL. PICL does not provide *configuration control and management facilities* (weight 4), hence, an average grade of 0.6.

3. **Level 2 : User Capability**
   PICL supports C and Fortran, hence, it got a grade of 0.8 for *language support*. Ease of Programming has been mostly been given an average grade except for Predictability in PICL which is good. PICL only supports the host-node programming model hence, the an average grade of 0.6 for this criterion. We have not evaluated *interoperability* for any of the tools we used : in such a case, we assign an average grade of 0.6. For similar reasons, the weight assigned was also 3. The documentation for PICL was fairly decent and we found PICL easy to learn. Hence, it got a good grade of 0.8 for *learning curve*; which was assigned a weight of 5.

   The next three criteria were not too important for us, hence, they were given (relatively) low weights of 3. We gave an average grade of 0.6 for support and maintenance to PICL because we did not need any support and maintenance for PICL. PICL did give good performance for the tracker, hence a high grade of 0.8 for *performance of tool services*.

4. **Level 3 : Application Development Capability or Process Evaluation**
   The application specification stage and analysis stage are independent of the tool that we used in this project. Hence an average grade of 0.6 is assigned to both these criteria. The Parallelization specifications differ for each tool which essentially is the communication primitives provided by the tool. The primitives provided by PICL were not very extensive and hence we have given a grade of 0.6. The application development stage capabilities will remain the same for all the tools because we developed a standardized procedure for porting the tracker to any tool. No extra work had to be done for any tool after this methodology was developed. Hence we have assigned an average grade of 0.6 for this criteria. The compile and run time stage capabilities have been assigned a grade of 0.6. The evaluation stage capabilities have been assigned a grade of 0.6. The maintenance and evolution stage gets an average grade of 0.6 because we have not used this stage in our project. It can be seen from column 12 that the normalized level grade is 0.60 which is equivalent to 60% acceptance for level 3 or *process evaluation* for PICL in this project.

The overall percentage tool acceptance which takes into account level 0-3 for PICL using our tool evaluation methodology is **61.07 %** and the process evaluation benchmark for PICL is **60%**.

5.2.4 Tool Evaluation Benchmarks for p4

We now briefly explain the evaluation process for p4. We used version 1.2 of p4 for this work. Table 5.10 - 5.12 shows the evaluation.

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
0	Tool Cost	-	-	1	1	1	1	1						
	Integration of tool into existing environment	-	-	0.8	1	0.8	4	3.2						
	Overheads	-	-	0.8	1	0.8	4	3.2						
	Portability	-	-	0.8	1	0.8	5	4.0	11.4	14	0.81	4	3.24	
1	Platforms Supported	-	-	0.8	1	0.8	5	4.0						
	Heterogeneous Processing capability	-	-	0.6	1	0.6	5	3.0						
	Generality of Tool Interface	-	-	0.6	1	0.6	5	3.0						
	Data Mapping and Decomposition	-	-	0.4	1	0.4	4	1.6						
	Communication Services	Point to Point Communication	4											
		Group Communication	3	7.0	10	0.7	4	2.8						
	Configuration Control and Management	-	-	0.6	1	0.6	4	2.4	16.8	30	0.56	5	2.80	
2	Language Support	-	-	0.8	1	0.8	3	2.4						
	Ease of Programming	Tailorability	2											
		Predictability	3											
		Error Handling	3	8	15	0.53	5	2.65						

Table 5.10: Example Evaluation of p4

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalised Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalised Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
2 (cont)	Programming Models Supported	-	-	0.6	1	0.6	3	1.8						
	Interoperability	-	-	0.6	1	0.6	3	1.8						
	Learning Curve	-	-	0.8	1	0.8	5	4.0						
	Run Time support for Parallel I/O	-	-	0.6	1	0.6	3	1.8						
	Debugging Capability	-	-	0.4	1	0.4	3	1.2						
	Application Profiling	-	-	0.6	1	0.6	3	1.8						
	Tool Robustness	Consistency	3											
		Evolution	3											
		Fault Tolerance	3	9	15	0.60	5	3.00						
	Support and Maintenance	-	-	0.4	1	0.4	4	1.6						
	Performance of Tool Services	-	-	0.6	1	0.6	4	2.4	25.05	41	0.61	5	3.05	
3	Application Specification Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Application Analysis Stage Capabilities	-	-	0.6	1	0.6	1	0.6						
	Parallelization Specification Capabilities	-	-	0.6	1	0.6	1	0.6						

Table 5.11: Example Evaluation of p4 (cont ..)

Level No. (1)	Criterion (2)	Sub-Criterion (3)	Sub-Criterion Grade (4)	Criterion Grade (5)	Maximum Criterion Grade (6)	Normalized Criterion Grade (7)	Criterion Weight (8)	Weighted Criterion Grade (9)	Level Grade (10)	Maximum Level Grade (11)	Normalized Level Grade (12)	Level Weight (13)	Weighted Level Grade (14)	Overall Acceptance (15)
3 (cont.)	Application Development Stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Compile - Time / Run - Time stage capabilities	-	-	0.8	1	0.8	1	0.8						
	Evaluation stage capabilities	-	-	0.6	1	0.6	1	0.6						
	Maintenance / Evolution stage capabilities.	-	-	0.6	1	0.8	1	0.6	4.4	7	0.62	1	0.62	0.6473
													Percentage Acceptance :	64.73 %

Table 5.12: Example Evaluation of p4 (cont ..)

The evaluation criteria used are the same as described for PVM. Since p4 is being evaluated for the same application Tracker, the weights for each criteria and sub-criteria remain the same for p4 also. The grades for judging the tool varies and this reflect the final acceptance percentage of this tool for the Tracker application.
The grading process for p4 is explained below briefly.

Level 0 : Hardware/Software Requirements p4 has an ideal grade of 1 for *tool cost* since it is a public domain tool. p4 got a high grade of 0.8 for *integration of tool into existing environment* because of ease of installation and absence of hitches in integrating into our hardware platforms. p4 requires a remote shell established on each of the remote nodes and hence, it gets a grade of 0.6 for *overheads*. The tracker was ported to the IBM RS/6000, SUN 4 workstations and CM-5 using p4, without any glitches; hence a grade of 0.8 for this criterion.

Level 1 : Tool Capability p4 supports different platforms and heterogeneous capability. Hence, p4 gets a grade of 0.8 for *platforms supported* and a grade of 0.6 for *heterogeneous processing capability* since it did not perform well for a combination of SUN and IBM workstations. p4 gets a grade of 0.8 for *generality of tool interface* since it was ported on heterogeneous platforms with little difficulty. p4 offers very little in the way of *data decomposition* functions and hence a grade of 0.4 for this capability. *Communication services* gets low grades of 3 in each sub-criterion because of overhead involved in execution of communication routines.

Level 2 : User Capability p4 supports C and FORTRAN and hence the grade of 4 for *language support*. p4 is easy to use and handle, and hence the grades for the sub-criteria in *ease of programming* are high. p4 was easy to learn and the *debugging facilities* offered by p4 were inadequate in aiding debugging of programs, hence a low grade of 0.4 for this criterion. Documentation was good so p4 gets a grade of 0.8 for *learning curve*. p4 was found to be a fairly robust tool. The average grades for the sub-criteria indicate this. p4 was found to be easy to support and required little house-keeping. Reasonable performance of the tracker was achieved using p4, hence the grade of 0.6 for *tools services* criterion.

Level 3 : Application Development Capability The application specification stage and analysis stage are independent of the tool that we used in this project. Hence an average grade of 0.6 is assigned to both these criteria. The Parallelization specifications differ for each tool which essentially is the communication primitives provided by the tool. The primitives provided by p4 were not very extensive and hence we have given a grade of 0.6. The application development stage capabilities will remain the same for all the tools because we developed a standardized procedure for porting the tracker to any tool. No extra work had to be done to any tool after this methodology was developed. Hence we have assigned an average grade of 0.6 for this criteria. The compile and run time stage capabilities have been assigned a grade of 0.8. The evaluation stage capabilities have been assigned a grade of 0.6. The maintenance and evolution stage gets an average grade of 0.6 because we have not used this stage in our project. It can be seen from column 12 that the normalized level grade for is 0.62 which is equivalent to 62% acceptance for level 3 or *process evaluation* for p4 in this project.

The overall percentage tool acceptance which takes into account level 0-3 for p4 version 1.2 using our tool evaluation methodology is **64.73 %** and the process evaluation benchmark for p4 is **62%**.

Table 5.13 shows the process evaluation benchmarks for the tools we used in this project. This metric takes into account only the level 3 of tool evaluation criteria. The metric shows how well or easily each tool could be applied to the HHPC software development process model developed in chapter 1. The tools are listed in the order of acceptance.

Tool	Acceptability Metric
PVM	69 %
Express	63 %
p4	62 %
PICL	60 %

Table 5.13: Process Evaluation Bencnmarks for PVM, Express, p4 and PICL

Table 5.14 shows the tool evaluation benchmarks for the tools we used in this project. This metric reflects the over all tool acceptance which takes into account level 0 to level 3 of tool evaluation criteria with the Concurrent Multi-Target Tracker as a running example.

Tool	Acceptability Metric
PVM	67.27 %
PICL	61.07 %
p4	64.73 %
Express	59.33 %

Table 5.14: Tool Evaluation Benchmarks for PVM, PICL, p4 and Express

Chapter 6

Summary and Conclusions

Current trends in parallel/distributed computing indicate that the future of parallel computing lies with the integration of existing computers into a single heterogeneous high performance computing environment that allow them to cooperate in solving complex problems. The HHPC environment will capitalize on existing architectures and on current advances in computing, networking and communication technology to provide efficient, *cost-effective*, *scalable*, *high performance* distributed computing. Software development in any parallel/distributed environment is a non-trivial process and requires a thorough understanding of the application and the architecture. In this project, we proposed a software development model for an HHPC environment. This model is defined as a set of stages which correspond to phases encountered in the software development process. We have ported a complex command and control application, the concurrent Multi-Target Tracker, to different computer platforms ranging from parallel computers to workstations using important message passing software tools. The main objective of choosing the tracker application was to allow us to experiement with its implementation using different tools. This experimentation assisted us in developing a hierarchical approach to evaluate and benchmark parallel/distributed software tools such as PVM, EXPRESS, PICL and P4. This hierarchical approach evaluates tools using four levels where each one represents one perspective. Level 0 evaluates tools from the perspective of cost and its integration to an existing computing environment. Level 1 evaluates tools from a tool capability perspective. Level 2 evaluates tools from a user capability perspective and Level 3 evaluates the capability of tools to assist in developing new HHPC applications or in parallelizing existing sequential code and porting it on some of the HHPC platforms. Furthermore, in this project, we develop a systematic approach to quantify the tool criteria so that we can eventually produce an overall tool score or tool acceptability metric, that can be used to compare tools. This algorithm can also be used to select the best set of tools suitable for a given class of applications and HHPC environment.

Future work can be outlined as follows:

1. Develop a benchmarking suite that covers most application classes needed in BM/C3IS. For each application class in this suite, identify the best high performance distributed computing environment to run this application and the tool(s) needed to implement such an application class.

2. Investigate the issues and requirements for an integrated software development environment for heterogeneous high performance computing.
3. Develop an integrated software development model for HHPC applications.
4. Develop a comprehensive evaluation methodology for parallel/distributed software tools.
5. Apply the software model developed in item 3 to implement a representative set of BM/C3IS applications using a subset of parallel/distributed software tools.
6. Validate and fine tune the evaluation methodology based on the results and experiences learned from developing and running the selected BM/C3IS applications.

Bibliography

[1] Salim Hariri, Manish Parashar, JongBaek Park, and Fang-Kuo Yu, "An Environment for High-Performance Distributed Computing", Technical Report SCCS-418, Northeast Parallel Architectures Center, Syracuse University, 111 College Place Room # 3-201, Syracuse NY 13244-4100, 1992.

[2] Salim Hariri, Manish Parashar, Jong Baek Park, and Fang-Kuo Yu, "A Case for Heterogeneous Network Computing", Technical Report SCCS-417, Northeast Parallel Architectures Center, 111 College Place, Room # 3-201, Syracuse NY 13244-4100, 1992.

[3] T.D.Gottschalk, "Concurrent multi-target Tracking", The Fifth Distributed Memory Computing Conference, Volume I, pages 52-57, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, California 90720-1264, 1990. IEEE Computer Society Press.

[4] T.D.Gottschalk, "Concurrent Implementation of Munkres Algorithm", January 1990, Fifth Distributed Memory Conference, Caltech Report C3P-921.

[5] T.D.Gottschalk, and P.Burns, "Simulation89 tracking", September 1989. Caltech Report C3P-820.

[6] T.D.Gottschalk, "A New Multi-Target Tracking Model", California Concurrent Computation Project, California Institute of Technology, Pasadena, California 91125.

[7] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker, "Solving Problems on Concurrent Processors", New Jersey : Prentice Hall, 1988.

[8] T.D.Gottschalk, "CALTRAX The Tracking Program for Simulation 87", California Institute of Technology, Pasadena, California 91125, Caltech Re port C3P-478.

[9] T.D.Gottschalk, "Precision filters for Boost Phase Tracking", California Institute of Technology, Pasadena, California 91125, Caltech Report C3P-479.

[10] Geoffrey C. Fox, and David W. Walker, "A Portable Programming Environment for Multiprocessors", California Institute of Technology, Pasadena, CA 91125, Caltech Report C3P-496.

[11] Paul Messina, Arnold Alagar, Clive Ballie, Edward Felten. , Paul Hipes, ANke Kamrath, Robert Leary, Wayne Pfeiffer, Jack Rogers, David Walk er, and Roy Williams, "Benchmarking Advanced Architecture Computers", Caltech Supercomputing Facility, San Diego Super Computing Center, Department of Mathematics, University of South Carolina, Caltech Report, C3P712.

BIBLIOGRAPHY

[12] Paul Messina, "Performance Study of Missile Tracking Algorithm on Selected Computer Architectures", California Institute of Technology, Pasadena, California 91125, Caltech Report C3P-668. 87", California Institute of Technology, Pasadena, California 91125.

[13] Hyt T. Cao, and Clive F. Ballie, "Caltech Missile Tracking Program, A Benchmark Comparison", California Institue of Technology, Pasadena, CA 91125, October 1988.

[14] Adam Beguelin, Jack Dongara, Al Geist, Robert Manchek , and Vaidy Sunderam, "U Guide to PVM", Oak Ridge National Laboratory, Oak Ridge TN 378 31-6367 and Department c ematics and Computer Science, Emory University, February 1993.

[15] G. A. Geist, M.T. Heath, B.W. Peyton, and P.H. Worley, "User Guide to PICL", Mathematical Sciences Section, P.O. Box 2009, Bldg. 9207-A, Oak Ridge National Laboratory, Oak Ridge, TN 37831-8083, August 1990.

[16] Parasoft Corporation, "Express 3.0 Documentation", Parasoft Corporation, 2500, E.Foothill Blvd. Pasadena, CA 91107.

[17] Ralph Butler, and Ewing Lusk, "User's Guide to the p4 Programming System", Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439-4801

[18] Erwin Kreyszig "Advanced Engineering Mathematics", Professor of Mathematics, Ohio State University, Columbus, Ohio.

[19] Robert Firth, Vicky Mosley, Richard Plethia, Lauren Roberts, and William Wood, " A guide to the classfication and assessment of software engineering tools", CMU/SEI-87-TR-10 Technical Report, August 1987.

[20] Goeffrey C. Fox, "Concurrency Practice and Expereince", Volume 4 Number 6, Wiley Publishers, Spetember 1992.

[21] R.Olson, "Parallel Processing in a Message Based Operating System", IEEE Software, July 1985.

[22] G. A. Geist, and V.S. Sunderam, "Network Based Concurrent Computing on the PVM System", Oak Ridge National Laboratory, Oak Ridge, TN 37831 and Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322.

[23] D.Reed and D.Grunwald, "The performance of multicomputer interconnection network", IEEE Computer, June 1987.

[24] Oliver A. McBryan, and Eric F. Van de Velde, "Hypercube Algorithms and Implementations", Courant Institue of Mathematical Sciences, New York University, New York, NY 10012.

[25] Stone, H.S, "High-Performance Computer Architecture", Addision-Wesley, 1987.

[26] Anthony Skjellum, Alvin P.Leung, and Manfred Morati, "A Portable Multicomputer Communication library atop the reactive kernel", California Institute of Technology, Pasasdena, California 91125.

[27] Glenn Zorpette, "Teraflops Galore", *IEEE Spectrum*, vol. 29, pp. 26–76, September 1992.

[28] Salim Hariri, Manish Parashar, JongBaek Park, Fang-Kuo Yu, and Geoffrey Fox, "A Message Passing Interface for Parallel and Distributed Computing", To be presented at the 2nd International Symposium on High Performance Distributed Computing, Spokane, Washington, July 1993.

[29] Richard F. Freund and D. Sunny Conwell, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing", *Supercomputing Review*, vol., pp. 47–50, October 1990.

[30] Norris Parker Smith, "The Future of High-Performance Computing: The 1990 Federal Assessment", *Supercomputing Review*, vol. , pp. 52–53, oct 1990.

[31] Gordon Bell, "Ultracomputers: A Teraflop Before Its Time", *Communications of the ACM*, vol. 35, pp. 27–47, August 1992.

[32] Ashfaq Khokar, Viktor K. Prasanna, Mohammad Shaaban, and Cho-Li Wang, "Heterogeneous Supercomputing: Problems and Issues", *Heterogeneous Processing Workshop, IPPS '92*, vol., 1992.

[33] Victor R. Basili and John D. Musa, "The Future Engineering of Software: A Management Perspective", *IEEE Computer*, vol. 24, pp. 90–96, September 1991.

[34] Gregor Von Laszewski, Manish Parashar, A. Gaber Mohamed, and Geoffrey C. Fox, "On the Parallelization of Blocked LU Factorization Algorithms for Distributed Memory Architectures", *Supercomputing '92, Minneapolis*, vol., pp. 170–179, November 1992.

[35] Manish Parashar, Salim Hariri, A. Gaber Moahamed, and Geoffrey C. Fox, "A Requirement Analysis for High Performance Distributed Computing over LAN's", *in Proceedings, First International Symposium on High Performance Distributed Computing*, pp. 142–151, September 1992.

[36] J. E. Boillat, H. Burkhart, K. M. Decker, and P. G. KROPF, "Parallel Computing in the 1990's: Attacking the Software Problem", *Physics Report (Review Section of Physics Letters)*, vol. 207, pp. 141 - 165, 1991.

[37] Lucian Russell and R. N. C. Lightfoot, "Software Development Issues for Parallel Processing", *Proceedings of the 12th Annual International Computer Software and Applications Conference*, vol., pp. 306–307, 1988.

[38] Manish Parashar, Salim Hariri, Tomasz Haupt, and Geoffrey C. Fox, "An Integrated Software Development Model for Heterogeneous High Performance Computing", Technical Report SCCS-453, Northeast Parallel Architectures Center, Syracuse University, 111 College Place Room # 3-201, Syracuse NY 13244-4100, April 1993.

[39] Kim Mills, Gang Cheng, Michael Vinson, Sanjay Ranka, and Geoffrey C. Fox, "Software Issues and Performance of a Parallel Model for Stock Option Pricing", *Proceedings of the 5th Australian Supercomputing Conference, Melbourne, Australia*, vol. , December 1992.

[40] Kim Mills, Gang Cheng, Michael Vinson, and Geoffrey C. Fox, "Expressing Dynamic, Asymmetric, Two-Dimensional Arrays for Improved Performance on the DECmpp-12000", Technical Report SCCS-261, Northeast Parallel Architectures Center, 111 College Place, Syracuse University, Syracuse, NY 13244-4100, October 1992.

[41] Manish Parashar, Salim Hariri, Tomasz Haupt, and Geoffrey C. Fox, "An Interpretive Framework for Application Prediction", Technical Report SCCS-479, Northeast Parallel Architectures Center, Syracuse University, 111 College Place Room # 3-201, Syracuse NY 13244-4100, April 1993.

[42] Thomas Bemmerl, Arndt Bode, Peter Braun, Olav Hansen, Thomas Treml, and Roland Wismüller, *The Design and Implementation of TOPSYS*, Technische Universität München, Institut Für Informatik, July 1991, Ver 1.0.

[43] D. Gannon, Y. Gaur, V. Guarna, D. Jablonowski, and A. Malony, "FAUST: An Integrated Environment for Parallel Programming", *IEEE Software*, vol. , pp. 20–27, July 1989.

[44] Constantine D. Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, and Bruce Leung, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", *Proceedings of the International Conference on Parallel Processing*, vol. 2, pp. 39–48, Aug. 1989.

[45] J. J. Dongarra and D. C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs", in L. H. Jamieson, D. B. Gannon, and R. J. Douglas, editors, *The Characteristics of Parallel Algorithms*, vol. . MIT Press, 1987.

[46] Özalp Babaoğlu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli, and Luigi Alberto Giachini, "Paralex: An Environment for Parallel Programming in Distributed Systems", Technical report, Department of Mathematics, University of Bologna, Piazza Porta S. Donato, 5, 40127 Bologna, Italy, 1991.

[47] Arthur Ieumwananonthachai, Akiko N. Aizawa, Steven R. Schwartz, Benjamin W. Wah, and Jerry C. Yan, "Intelligent Mapping of CommunicationProcesses in Distributed Computing Systems", *Supercomputing '91, Proceedings*, vol. , pp. 512–521, 1991.

[48] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer, "An Interactive Environment for Data Partitioning and Distribution", *Proceedings of the* 5^{th} *Distributed Memory Computing Conference, Charleston, South Carolina*, vol. , pp. 1160–1170, Apr. 1990.

[49] Alan Sussman, "Execution Models for Mapping Programs onto Distributed Memory Parallel Computers", Technical Report 189613, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, Virginia 23665-5225, Mar. 1992.

[50] Manish Gupta and Prithviraj Banerjee, "Compile-Time Estimation of Communication Costs in Multicomputers", Technical report, Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1101 W. Springfield Avenue, Urbana IL 61801, .

[51] Peter H. Mills, Lars S. Nyland, Jan F. Prins, John H. Reif, and R. W. Wagner, "Prototyping Parallel and Distributed System in Proteus", *Proceedings of the* 3^{rd} *IEEE Symposium on Parallel and Distributed Processing*, vol. , 1991.

[52] H. Zima, H. Bast, and M. Gerndt, "SUPERB: A Tool for Semi-Automatic SIMD/MIMD Parallelization", *Parallel Computing*, vol. , 1988.

[53] Sandeep Bhatt, Marina Chen, Cheng-Yee Lin, and Pangfeng Liu, "Abstractions for Parallel N-body Simulations", Technical Report DCS/TR-895, Yale University, 1992.

[54] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, K. McKinley, and J Subhlok, "The ParaScope Editor: An Interactive Parallel Programming Tool", *Supercomputing '89, Reno, Nevada*, vol. , Nov. 1989.

[55] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 206–217, Apr. 1990.

[56] Bernd Mohr, "SIMPLE: A Performance Evaluation Tool Environment for Parallel and Distributed Systems", *Proceedings of the 2nd European Distributed Memory Computing Conference (EDMCC2)*, vol. , pp. 80–89, Apr. 1991.

[57] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The Rice Parallel Processing Testbed", 1988 ACM 0-89791-254-3/88/0005/0004 pp 4-11, 1988.

[58] Daniel Pease, Arif Gafoor, Ishfaq Ahmad, David L. Andrews, Kamal Foudil-Bey, Thomas E. Karpinski, Mohammad A. Mikki, and Mohammad Zerrouki, "PAWS: A Performance Evaluation Tool for Parallel Computing Systems", *IEEE Computer*, vol. , pp. 18–29, Jan. 1991.

[59] F. Andre and A. Joubert, "SiGle: An Evaluation Tool for Distributed Systems", *Proceedings of the International Conference on Distributed Computing Systems*, vol. , pp. 466–472, 1987.

Appendix A

Glossary

- **Crystal Router Algorithm**
  An algorithm used to pass messages between arbitrary nodes of a hypercube. It is useful for irregular problems when the message traffic changes dynamically

- **Express**
  An product of *Parasoft* Corporation which is used for developing Parallel/Distributed applications.

- **Grading/Scaling**
  A process of assigning numerical values called grades for a Tool criterion and Tool sub-criterion in a Tool level hierarchy

- **HHPC**
  Heterogeneous High Performance Computing: An environment which capitalizes on existing architectures and on current advances in computing, networking and communication technology to provide efficient, cost-effective, scalable, high-performance distributed computing.

- **iPSC/860**
  A hypercube machine developed by Intel Corporation.

- **MIMD**
  Multiple Instruction Multiple Data. Parallel machine which can process Multiple Instructions with Multiple Data are called MIMD machines

- **Newton-Raphson Iteration**
  An iteration method for solving equations $f(x) = 0$, where f is assumed to have a continuous derivative f'. This method is commonly used because of its simplicity and great speed.

- **MOA**
  Measure Of Accpetance which indicates the acceptance of a tool for a given application. MOA is the result of applying the Tool Evaluation process.

- **Munkres Algorithm**
  An algorithm used to generate the optimal association solution for modified distance matrix.

- **nCUBE**
  A hypercube machine developed by nCUBE Corporation.
- **PICL**
  Portable Instrumented Communication Library designed to provide portability, ease of programming, and execution tracing in parallel programs
- **PVM**
  Parallel Virtual Machine: A software package that allows the utilization of heterogeneous network of parallel and serial computers as a single computational resource.
- **p4**
  A library of macros and subroutines developed at Argonne National Laboratory for programming a variety of parallel machines.
- Report Construction
  Process of generating a report file containing parameter estimates, covariance matrices and uniqueness-tags prepared after each sensor scan.
- **SIMD**
  Single Instruction Multiple Data. Parallel Machines which can process Single Instructions with Multiple Data are called SIMD machines.
- **Tool Criterion**
  A feature, facility or capability that is provided by a tool.
- **Track Extension**
  A mechanism for extending an existing track in a global track file
- **Trajectory Estimation**
  A mechanism of predicting the path taken by a target in terms of latitude, longitude, initial launch relative to due north and time of launch.
- **Tool Hierarchy**
  A hierarchy of levels which contain tool criteria and sub-criteria used for evaluating a tool.
- **Track Initiation** Identifying the existence of a new target.
- **Track Pruning**
  A mechanism of deleting non-existent tracks. Such entries are created in the global track file because of inconsistent sensor reports.
- **Track Redistribution**
  Process of distributing the global track file across the nodes in preparation for processing the next scan of data.
- **Tool sub-criterion**
  Sub-set of a Tool criterion

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.