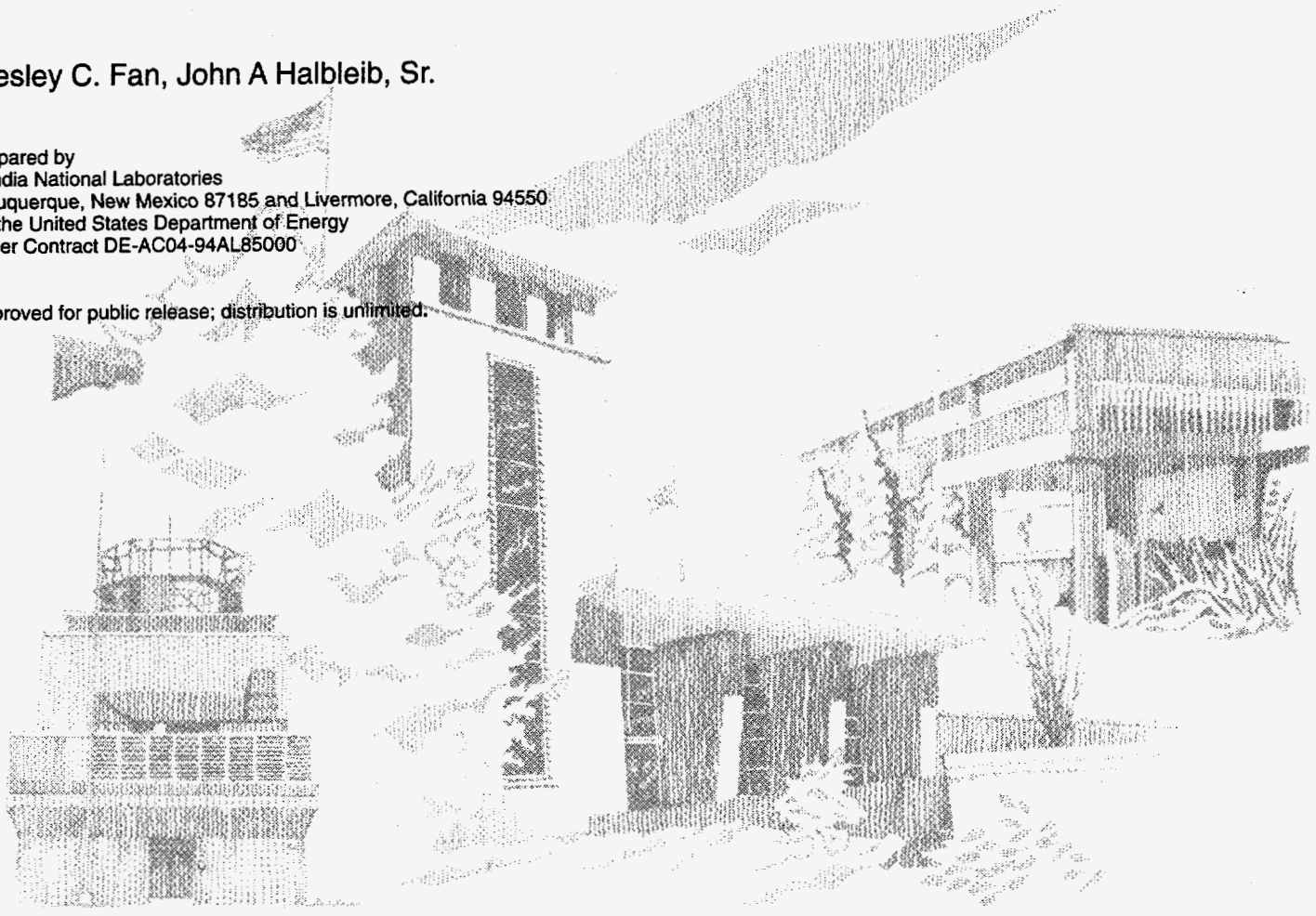
SANDIA REPORT

SAND96–2147 • UC–405
Unlimited Release
Printed September 1996

Parallel Processing ITS

Wesley C. Fan, John A Halbleib, Sr.

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-94AL85000



SF2900Q(8-81)

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

SAND96-2147
Unlimited Release
Printed September 1996

Distribution
Category UC-405

PARALLEL PROCESSING ITS

Wesley C. Fan
Radiation and Electromagnetic Analysis Department
and
John A. Halbleib, Sr.
Simulation Technology Research Department
Sandia National Laboratories
Albuquerque, NM 87185

ABSTRACT

This report provides a users' guide for parallel processing ITS on a UNIX workstation network, a shared-memory multiprocessor or a massively-parallel processor. The parallelized version of ITS is based on a master/slave model with message passing. Parallel issues such as random number generation, load balancing, and communication software are briefly discussed. Timing results for example problems are presented for demonstration purposes.

ACKNOWLEDGEMENTS

This work was made possible with the continuous support from Leonard J. Lorence and Gary J. Scrivner. We thank James H. Renken for his encouragement of this project. We also thank Clifton R. Drumm and Patrick J. Griffin for their comments and suggestions.

DISCLAIMER

Contents

List of Figures

List of Tables

PARALLEL PROCESSING ITS

I. INTRODUCTION

Advances in computer hardware and communication software have made it possible to perform parallel computing for many scientific/engineering applications. Monte Carlo calculations are inherently parallelizable because the individual particle trajectories can be generated independently with minimum need for interprocessor communication. Furthermore, the number of particle histories that can be generated in a given amount of wall-clock time is nearly proportional to the number of processors. This is an important fact because the inherent statistical uncertainty in any Monte Carlo result decreases as the square root of the number of histories. For these reasons, researchers have expended considerable effort to take advantage of different parallel architectures for a variety of Monte Carlo radiation transport codes, often with excellent results [1]. Generally speaking, parallel processing can reduce the notoriously high computational expense often associated with the Monte Carlo method and allows users to solve extremely complex problems with fast turnaround time.

The Integrated Tiger Series (ITS) [2] provides state-of-the-art Monte Carlo solutions of linear, time-independent, coupled electron/photon radiation transport problems with or without the presence of external electric and magnetic fields. It has been widely used in weapon-effect simulator design and analysis, radiation dosimetry, radiation effects studies and medical physics research. Since its inception, the goal of the ITS developers has been to simultaneously maximize physics accuracy and operational efficiency. This is accomplished by employing the most complete physics models describing the production and transport of the electron/photon cascade, the best available cross-section data and sampling distributions, and variance reduction techniques for various difficult applications [3]. In this work, we focus on a major software development for the ITS code system so that ITS calculations can now be performed in parallel. This is accomplished by developing an update for ITS Version 3 which can be adapted by users to construct and tailor the ITS codes for specific applications and for various parallel-processing platforms. These platforms can be a UNIX workstation network, a symmetrical multiprocessor, or a massively parallel machine.

In the following sections we first discuss the parallel algorithm appropriate for the ITS code system and its implementation. Two techniques that proved effective for load balancing across multiple processors and machines are briefly discussed. Timing results and performance evaluation for selected problems are described.

II. PARALLEL PROCESSING ISSUES

A. Master/Slave Paradigm

In ITS, the particle histories are divided into "batches" of equal size and the evaluation of the estimated quantities are performed using batch-averaged sample statistics. Since the

batchwise evaluation can be performed independently, it provides a natural partition for parallel processing. At present, the parallelized version of ITS is based upon a master/ slave paradigm in conjunction with message-passing. The basic operations of the parallel code can be summarized as follows: (1) the master process performs the input functions, starts up the slave processes, processes the problem-dependent parameters and sends a copy of parameters to all slaves, (2) the slaves perform the Monte Carlo calculations, i.e., generating particle trajectories and scoring, and (3) the master receives and combines the data tallied by the slaves, and finally outputs the results. These operations are shown schematically in Figure 1. All the message-passing tasks, including process control and data transfer, are handled by the communication software.

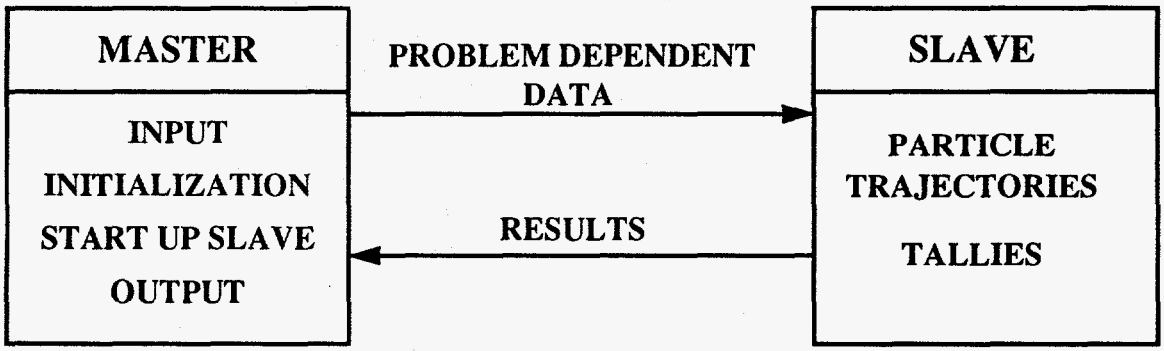


Figure 1. A schematic diagram of the master/slave model used in ITS. The master and slave processes may execute different statements by branching within a single program. The arrowheads indicate the flow of data using message passing.

One can make two observations about this master/slave model. First, there is no interprocessor communication required between the slaves. The problem-dependent data and the tallied results are transferred between master and slaves, but no data is shared between the slaves. Thus, message passing is needed only at the start and end of Monte Carlo calculation. Second, with efficient network communication, it is obvious that step (2) will require the majority of the computation time. Since each slave process can carry out these tasks concurrently, this time requirement can be reduced almost linearly with the number of processors.

B. Random Number Generation

The generation of random-number sequences for large-scale Monte Carlo simulations in a parallel-processing environment poses a non-trivial problem. The random number sequences for each processor must be independent, with good "randomness" properties, and with sufficiently long period. Here, we adopt the pseudorandom number generator, RANMAR, proposed by Marsagalia and Zaman. Detailed information and implementation can be found in the review article by James [4]. The basic algorithm of RANMAR is a combination of two different random number sequences, $\{X_i\}$ and $\{Y_i\}$. The first is a lagged Fibonacci generator,

$$X_i = \begin{cases} X_{i-97} - X_{i-33}, & if \;\; X_{i-97} \geq X_{i-33} \\ X_{i-97} - X_{i-33} + 1, & otherwise, \end{cases} \tag{1}$$

where a starting table of 97 values is initialized using a combination of lagged Fibonacci method using three lags, and a multiplicative congruential generator. The second one is a simple arithmetic sequence for the prime modulus $2^{24} - 3 = 16777213$. This sequence is defined as

$$Y_i = \begin{cases} Y_{i-1} - c, & if \;\; Y_i \geq c \\ Y_{i-1} - c + d, & otherwise, \end{cases} \tag{2}$$

where $c = 7654321/16777216$ and $d = 16777213/16777216$. The final random number is then obtained by a subtraction operation

$$Z_i = \begin{cases} X_i - Y_i, & if \;\; X_i \geq Y_i \\ X_i - Y_i + 1, & otherwise \end{cases} \tag{3}$$

The most exceptional property of this generator is the extreme ease of generating independent disjoint sequences, which can be accomplished by initializing the lagged Fibonacci generator with different integers. Furthermore, RANMAR has been tested for randomness, and has been demonstrated to have very long period [4-5]. It is noted that this generator is more expensive to compute than the simple multiplicative linear congruential generators since all the operations are carried out in floating-point. However, the effect of this may be insignificant since the computing time spent in the random number generation is almost negligible in comparison to that for particle tracking.

To employ this random number generator for parallel ITS calculations, an integer seed will be used to initialize the lagged Fibonacci sequence for each batch of histories. This seed itself is produced by a simple linear congruential generator

$$S_{n+1} = 1366 \cdot S_n + 150889 (mod \;\; 714025). \tag{4}$$

With this, an integer ranging from 1 to 714025 will be selected to start RANMAR and produce a sequence of length 10^{34}, and is guaranteed to be independent of any other sequences. Although this implementation limits the maximum number of batches to 714025, we believe it is sufficient for realistic simulation. More importantly, since each batch uses an independent random number generator, the computed results will be independent of the number of processors and are reproducible.

C. Load Balancing

The goal of load balancing is to enhance performance and achieve the greatest possible speedup of a parallel program. A balanced program can usually keep all processors busy and have them finish roughly at the same time. Otherwise, valuable processor cycles are wasted if some processors have to wait on others to finish. Load balancing is essential in parallel processing ITS since the computing time for each batch may vary, and thus adversely affect the performance. This variation in computing time may result for the following reasons: (1) the stochastic nature of particle histories; a few anomalous batches may involve long trajectories, (2) the difference in computational power on each machine in a heterogeneous configuration, and (3) the change in computational performance in a multiuser, time-sharing environment. These considerations must be taken care of by adjusting the way the problem is distributed in a parallel system.

The current version of ITS provides two load distribution schemes, namely, the static and dynamic methods. The static method is simple and easy to implement. In this method the required tasks (or batches) are divided up and assigned to the available machines or processors. The assigned number of batches can vary from machine to machine to account for different computation power for different machines. These assignments are set at the start and will not be adjusted to the actual loading and performance. As one may expect, this scheme can be quite effective on a dedicated or lightly loaded system, but it does not provide any mechanisms to address issue (1) which can be the main impediment to efficiently running ITS in parallel.

Dynamic load-balancing is accomplished by the "pool-of-tasks" paradigm. Initially, each slave process is given a batch just as in the static scheme. As a slave process finishes its task it will receive another one. With this scheme all the slave processes are kept busy as long as there are batches remaining in the pool. The work load for each processor (machine) is adjusted according to the "realistic" computational performance which may be problem-dependent and can be changing dynamically as other users share the resources.

The static and dynamic load-distribution schemes also have different impact on the dump/restart operation. For the static scheme, the batches of histories are generated in cycles, with a cycle being defined as the time period that each process starts and finishes a batch of histories. A synchronization or a barrier is set up at the end of a cycle so that the output can be updated and the dump file can be written. On the other hand, there is no clear way to define such a cycle for the dynamic scheme since the batches are started and

finished in a random fashion. Consequently, the dump file will only be written at the end of the run.

III. IMPLEMENTATION

A. Communication Software Packages: MPI, PVM, and NX

There are numerous communication software packages to support message passing on different classes of parallel machines. Although there are many variations, these packages all provide the basic functionalities such as point-to-point communication, collective communication, and process management. Recently, several systems have demonstrated that a message-passing system can be efficiently and portably implemented. In the parallelized version of ITS, we incorporated three of such systems, Message Passing Interface (MPI) [6], Parallel Virtual Machine (PVM) [7], and Intel NX library [8].

The NX library is the native message-passing library used on the Intel PARAGON and will be supported on the coming Teraflops supercomputer at Sandia. Both MPI and PVM are freely available software, and have been widely used in many scientific/engineering applications. The most attractive feature of these software packages is that they allow users to construct their own parallel machine by linking many UNIX workstations together. In addition, they also support shared-memory multiprocessors (SMP) and massively-parallel processors (MPP).

At present, we prefer MPI over PVM for the following reasons. First, MPI is becoming a standard and its features are formally specified. In contrast, PVM is an ongoing research project with no obligation to provide compatibility. Therefore, its functionality may change, and lack of support may be a concern in the future. Second, PVM has a fundamental deficiency in communication performance. This deficiency is mainly due to excessive buffering (pack and unpack) and is most apparent on multiprocessors with fast interconnection networks, such as the Intel PARAGON. On this machine, PVM communication performance can be orders of magnitude worse than that of the native message-passing library, while MPI performance is more comparable.

B. Correction Update

We have developed an "update" (a correction set for UPEML [9]) for implementation of the parallel processing capability in ITS Version 3. This update consists of two major components: *corrections for the common block structure, and codes for process control and message passing*.

It is necessary to modify the common blocks in ITS to minimize both the number and the size of messages. With the current generation of machines and communication hardware, sending a message is still an expensive operation. As a rule of thumb, the fewer messages sent, the better the performance of a program. To achieve this, we have arranged the data structure so that the variables (arrays) with the same data type are grouped together and are stored in contiguous memory locations. Moreover, variables of the same data type are separated into two groups, the global and local variables. The global

variables are those provide information on problem input and tallied results and that have to be transferred between the master and the slave processes. With this arrangement, we can send the global variables for a given data type in one package by specifying the beginning address and the appropriate length, thus minimizing the number of messages. In the rearranged common blocks, the floating-point variables are placed first, followed by the integer, logical and character variables. The global variables are placed before the local variables.

As an example, we present the old and new structures of common block /calc/ in Figures 2 and 3. This common block contains the variables mainly used in the input and monte-carlo routines. With the old structure, one needs to either pack the variables into a message buffer and send it or send a large number of small messages. Both approaches can be time consuming. The new common block allows us to transfer data with two messages, one for floating-point variables (from ASTEP to W2Z) and the other for integer variables (from NTKAY to NPLOTS).

The correction update is designed in such a way that users can incorporate their own modifications to construct and tailor the codes for their specific applications. However, one must pay attention to how and where these modifications are made. If any new global variables are introduced, they need to be included in message passing between the master and slave processes. If any changes are made to the existing common blocks, the corresponding message passing routines must be modified to assure that the messages contain appropriate memory address and length.

C. New *DEFINE and Input Keywords

To select a desired communication software in the correction update, one has to define the following keyword

*DEFINE [MPI, PVM, NX]

where only one option listed in the brackets has to be activated. Similarly, the following keyword is used to select the load-balancing scheme,

*DEFINE [STATIC, DYNAMIC].

An additional keyword is also needed in the ITS input file.

Syntax: TASKS [parameter(1)], [parameter(2)]

Example: TASKS 8,2

Default: 1,1

The first parameter associated with the keyword TASKS is NTASKS, which specifies the number of tasks that will perform the Monte Carlo calculation in this run. The second parameter, NPRINT, specifies the frequency of printed output to the scratch file. Its value should be between one and the total number of batches.

The input parameter NTASKS has different implications to the program for different communication packages and load balancing schemes. This information is given in Table 1. For the static scheme, a positive value of NTASKS implies that the master process will perform the same Monte Carlo calculation as the slave processes. The number of

```
      COMMON /CALC/
     1 ACON(INMT),          ASTEP(INMAX,INMT),      AT(NSURV,INMT),
     $ PIBL(INMT),     CORBL(INMAX,INMT),
     2 BDIS(IKTOP,IMTOP,INMT),      BETA(INMAX1), BREN(IMTOP),
     3 CALPH(IMMAX),        CCH(NANGS), CHANG(INPANG,INRANG,INTANG,INMT),
     4 CEL(INPEL,INEEL),    COSAV(INMAX,INMT),      CTH(2),
     5 CTHB,                DBREN(IMTOP1),          DCALPH(IMMAX),
     6 DEEL(INEEL1),        DEPS(INEPS1),           DGAS(INGAS),
     7 DLAM(INLAN),         DRANG(INRNG1),          DRG(INMAX1,INMT),
     8 DRGS(INMAX,INMT),    DPANG,                  DPEL,
     9 DPPS,                DPS(INPPS,INEPS),       DTANG(INTNG1),
     $ DZ,                  E(INMAX1),              EARL(INMAX,INMT),
     1 ED(INMAX),           EDGK(INMT),
     2 EEL(INEEL),          NTBX,                   MTXX,
     3 EM(INMAX),           EPS(INEPS),             ESP(IJSPEC),
     4 FAN,                 G(IMMAX,INMAX,INMT),    GAS,
     5 GAUSS(INGAS1),       PANG(INPANG),           PAIR1(JATPR,INMT),
     6 PBREM(INMAX1,INMT),  PEL(INPEL),
     7 PKI,                 PPS(INPPS),             PRUTH(INMAX1),
     8 PSEC(INMAX1,INMT),   QAV(INMAX,INMT)
      COMMON /CALC/
     1 QCON(INMAX,INMT),    QCONS(INMT),            QPHOT,
     2 QS,                  RANG(INRANG),           RANGE(INMAX1,INMT),
     $ RAV(INMAX,INMT),     NTKAY(INMT),
     4 RKT(IKTOP),          RMAX(INMT),             SHD,
     $ COHSCT(NSURV,INMT),
     5 SPECIN(IJSPEC),      SUBFAC(JAHSUB),         SURV1(NSURV,INMT),
     6 T,                   TANG(INTANG),           TB,
     $  TD, TL(INLAN1), TP, FLAMC(INMAX,INMT),
     8 W,                   WB,                     TPMIN,
     $ WX,                  Z,                      ASIGN,      JAZSCT,
     $ BCUT(INMAX1,INMT),   PBREC(INMAX1,INMT),     NSUB(INMAX,INMT),
     1 ZSR,                 CTSR
      COMMON /CALC/
     1 IANN,          ICROSS,          IFUP,              IFUPA,
     2 IFUPL,         IFUPS,           IPR,
     3 JATIN,          JF,                ICTH,
     4 LAST,          LB,              NLAN,              NGAS,
     5 ND,            NDIF,            NDIFA,
     6 NDIFL,         NDIFS,           NSCALE,            NSKALE,
     7 NT,            NTANN,           IMPI(INMT),        NEMAX,
     8 NMT,           NMAX,            MTOP,
     9 NRANG,         NEPS,            KTOP,              NTANG,
     $ JSPEC,         MMAX,            NEEL,              NPANG,
     1 NPEL,          NPPS,            NMAX1,             CWCF,
     2 LD,            LSZ,             NGMAX
*IF -DEF,TIGER
      COMMON /CALC/
     1 X,             Y,               DX,                DY,
     2 CPH(2),        SPH(2),          STH(2),            SCH(NANGS),
     3 STHB,          CPHB,            SPHB,
     2 STSR,          CPSR,            SPSR,              XSR,
     3 YSR,           W1X,             W1Y,
     4 W1Z,           W2X,             W2Y,               W2Z
*IF DEF,ACCEPT
      COMMON /CALC/
     1 LPCZ,          LBCZK,           LBCZ
*ENDIF
*IF -DEF,PCODES
      COMMON /CALC/
     1 EAUG(3,INMT),  EK(4,INMT),
     2 PKEG(INMT),     PXRAY(INMAX1,INMT),
     3 RAUG(3,INMT),  RK(4,INMT),      WK(INMT),
     4 JATKA(4,INMT)
*ENDIF
*IF DEF,PCODES
     1 RWT(INMAX1,INSH,INMT),   EPART(INMAX1,INSH,INEM1,INMT),
     1 SHEL(INMAX1,INSH,INMT),  PPART(IMTAX,INTAB,INEM1,INMT)
*ENDIF
*IF DEF,MCODES
      COMMON /CALC/
     1 CTHT, STHT, CPHT, SPHT, CTHF, STHF, CPHF, SPHF,
     2 SP,    LBN,   IBETA, IMOD
*ENDIF
*IF DEF,CYLTRAN
      COMMON /CALC/
     1 NPLOTS, RMNPLT, RMXPLT, ZMNPLT, ZMXPLT
*ENDIF
      COMMON /CALC/
     1 ECEB(IMTAX,INTAB,INMT), ECAVE(KPTMAX), ECSIG(KPTMAX)
```

Figure 2. The existing data structure of common block /calc/ in ITS Version 3. The global variables are shown in italic font.

```
      COMMON /CALC/ ASTEP(INMAX,INMT), AT(NSURV,INMT), PIBL(INMT),
     &  CORBL(INMAX,INMT), BDIS(IKTOP,IMTOP,INMT), BETA(INMAX1),
     &  BREN(IMTOP), CALPH(IMMAX), CCH(NANGS),
     &  CHANG(INPANG,INRANG,INTANG,INMT), CEL(INPEL,INEEL),
     &  COSAV(INMAX,INMT), DBREN(IMTOP1), DCALPH(IMMAX), DEEL(INEEL1),
     &  DEPS(INEPS1), DGAS(INGAS), DLAM(INLAN), DRANG(INRNG1),
     &  DRG(INMAX1,INMT), DRGS(INMAX,INMT), DPANG, DPEL, DPPS,
     &  DPS(INPPS,INEPS), DTANG(INTNG1), E(INMAX1), EARL(INMAX,INMT),
     &  ED(INMAX), EDGK(INMT), EEL(INEEL), EM(INMAX), EPS(INEPS),
     &  ESP(IJSPEC), FAN, G(IMMAX,INMAX,INMT), GAS, GAUSS(INGAS1),
     &  PANG(INPANG), PAIR1(JATPR,INMT), PBREM(INMAX1,INMT), PEL(INPEL),
     &  PPS(INPPS), PRUTH(INMAX1), PSEC(INMAX1,INMT), QAV(INMAX,INMT),
     &  QCON(INMAX,INMT), RANG(INRANG), RANGE(INMAX1,INMT),
     &  RAV(INMAX,INMT), RKT(IKTOP), COHSCT(NSURV,INMT), SPECIN(IJSPEC),
     &  SUBFAC(JAHSUB), SURV1(NSURV,INMT), TANG(INTANG), TL(INLAN1),
     &  FLAMC(INMAX,INMT), BCUT(INMAX1,INMT), PBREC(INMAX1,INMT),
     &  ZSR, CTSR
       COMMON /CALC/ ECEB(IMTAX,INTAB,INMT),
*IF -DEF,PCODES
     &  EAUG(3,INMT), EK(4,INMT), PKEG(INMT), PXRAY(INMAX1,INMT),
     &  RAUG(3,INMT), RK(4,INMT), WK(INMT),
*EI
*IF DEF,PCODES
     &  RWT(INMAX1,INSH,INMT), EPART(INMAX1,INSH,INEM1,INMT),
     &  SHEL(INMAX1,INSH,INMT), PPART(IMTAX,INTAB,INEM1,INMT),
*EI
*IF DEF,MCODES
     &  CTHT, STHT, CPHT, SPHT, CTHF, STHF, CPHF, SPHF, SP,
*EI
*IF DEF,CYLTRAN
     &  RMNPLT, RMXPLT, ZMNPLT, ZMXPLT,
*EI
*IF -DEF,TIGER
     &  SCH(NANGS), STSR, CPSR, SPSR, XSR, YSR,
     &  W1X, W1Y, W1Z, W2X, W2Y, W2Z, X, Y, DX, DY, CPH(2), SPH(2),
     &  STH(2), STHB, CPHB, SPHB,
*EI
     &  ECAVE(KPTMAX), ECSIG(KPTMAX),
     &  ACON(INMT), CTH(2), CTHB, DZ, PKI, QCONS(INMT),
     &  QPHOT, QS, RMAX(INMT), SHD, T, TB, TD, TP, W, WB, TPMIN,
     &  WX, Z, ASIGN, CWCF
       COMMON /CALC/ NTKAY(INMT), NSUB(INMAX,INMT), IANN, IFUP, IFUPA,
     &  JATIN, JF, ICTH, NLAN, NGAS, NDIF, NDIFA, NTANN, IMPI(INMT),
     &  NRANG, NEPS, KTOP, NTANG, JSPEC, MMAX, NEEL, NMAX1, LD, NMT,
*IF -DEF,PCODES
     &  JATKA(4,INMT),
*EI
*IF DEF,MCODES
     &  LBN, IBETA, IMOD,
*EI
*IF DEF,CYLTRAN
     &  NPLOTS,
*EI
*IF -DEF,TIGER
*IF DEF,ACCEPT
     &  LPCZ, LBCZK, LBCZ,
*EI
     &  NTBX, MTXX, JAZSCT, ICROSS, IPR, LAST, LB, ND, NDIFL, NDIFS,
     &  NSCALE, NSKALE, NT, NEMAX, NMAX, MTOP, NPANG, NPEL, NPPS,
     &  LSZ, NGMAX
```

Figure 3. The rearranged data structure of common block /calc/ used in the parallelized ITS. The global variables, shown in italic font, are grouped together.

	Static Load Distribution		
Communication	NTASKS	MASTER[a]	Remarks[b]
MPI	> 0	1	$NP = NTASKS$
	< 0	0	$NP = NTASKS + 1$
PVM	> 0	1	$NP = NTASKS$
	< 0	0	use PVMTASKS[c]
NX	> 0	1	$NP = NTASKS$
	< 0	0	$NP = NTASKS + 1$
	Dynamic Load Distribution		
Communication	NTASKS	MASTER	Remarks
MPI	> 0	0	$NP = NTASKS + 1$
PVM	> 0	0	$NP = NTASKS + 1$
	< 0	0	use PVMTASKS
NX	> 0	0	$NP = NTASKS + 1$

a. If MASTER =1, the master process will perform Monte Carlo calculation as well as the slave process.

b. The column shows the relationship between the parameters NTASKS and NP, which is the number of processes started for the current run. For PARAGON, it is the number of nodes (size) requested at the run time. For MPI, it is the parameter supplied with the option "-np". For PVM, this is the number of host machines under the current PVM configuration.

c. PVMTASKS is an ASCII file used to control how processes get started on a UNIX workstation cluster. Detailed information on this file is given in the appendix.

processes started is identical to the number of tasks required. Conversely, a negative value of NTASKS means that the master process will not perform the Monte Carlo calculation. Thus the number of processes started should be one more than the number of tasks specified.

For the dynamic scheme, the master process will not perform the Monte Carlo calculation. Its mission is to perform input/output, monitor and control the other processes. Therefore the number of processes started should be one more than the number of tasks specified. For most UNIX computers, this arrangement does not waste any processing power since multiple processes can often be run on a processor simultaneously.

In the appendix, we provide further information on how to obtain the communication software MPI and PVM, and how to build the executable program on various parallel platforms. It is intended to assist ITS users, with limited experience in parallel processing, to get a quick start. More comprehensive guides on MPI, PVM, and NX can be found in References [6-8].

IV. PERFORMANCE EVALUATION

A. UNIX Workstation Network

The goal of parallel processing is to make the program run faster (shorter turn-around time) than it would in the corresponding serial run. A speedup ratio is often used to evaluate the performance of a parallized program. On a dedicated system, the speedup ratio can be calculated in the following manner:

$$S_N = \frac{T_1}{T_N}, \tag{5}$$

where S_N is the parallelization speedup, T_1 is the elapsed wall-clock time for a single processor, and T_N is the elapsed wall-clock time if N processors are used in the calculation. Furthermore, one can also define the parallel efficiency as the ratio of the speedup factor to the number of processors,

$$\varepsilon = \frac{S_N}{N}, \tag{6}$$

which provides a measure of efficiency of a parallel program and takes into account of effects such as synchronization and communication overhead. A parallel efficiency of one implies that a program executed with N processors will be N times faster than that with a single processor.

Table 2 summarizes the measured speedup ratios and parallel efficiencies for seven test problems on a cluster of SUN workstations. These test problems include the three standard codes (TIGER, CYLTRAN, and ACCEPT), two P-codes, and two M-codes of the ITS system, and utilize many tally and biasing options of the system. Sufficient particle histories were required so that the input and output times were negligible in

Table 2. Measured Speedup Factors[a] and Parallel Efficiencies of ITS/PVM on a UNIX Workstation Network for Selected Test Problems.

Code	Number of Processors									
	2[b]		4		8		12		16	
TIGER	1.99	0.99	3.81	0.95	7.39	0.92	10.88	0.91	14.19	0.89
CYLTRAN	1.97	0.99	3.92	0.98	7.32	0.92	10.93	0.91	14.31	0.89
ACCEPT	1.99	0.99	3.93	0.98	7.42	0.93	10.64	0.89	14.2	0.89
TIGER-P	1.97	0.99	3.93	0.98	7.77	0.97	11.5	0.96	14.39	0.90
ACCEPT-P	1.96	0.98	3.88	0.97	7.64	0.96	11.35	0.95	14.03	0.88
CYLTRAN-M	1.96	0.98	3.87	0.97	7.42	0.93	11.06	0.92	14.56	0.91
ACCEPT-M[c]	1.80	0.90	3.49	0.87	5.83	0.73	7.75	0.65	10.78	0.67

a. The speedup factor is measured against a single SUN SPARC 2/40 workstation.

b. For each number of processors, the speedup factors are given in the left column and the efficiencies are given in the right column.

c. Due to an anomalous batch which consumed 50% more computing time, the parallel efficiencies of the ACCEPT-M problem are much lower than the other codes.

comparison to the overall CPU times. It is observed that the speedup ratios increase almost linearly with the number of processors. The parallel efficiencies are about 90% or better except for the ACCEPT-M code, where it drops to as low as 65%. Further studies indicated that the relatively poor performance of the ACCEPT-M code was caused by an anomalous batch which consumed ~ 50% more CPU time than the other batches. It is believed that one or more electrons entered a vacuum region with a uniform magnetic field with velocities almost perpendicular to the field so that they drifted very slowly through this region. Consequently, extra computing time was needed to calculate these orbits, thus prolonging the CPU time for that batch.

B. Intel PARAGON

To further demonstrate the benefit of parallel processing and the dynamics between load balancing and parallel efficiency, we consider the following problem related to the EG&G linear accelerator. In this problem, it is desired to determine the energy and angular distribution of the bremsstrahlung spectrum generated from an electron beam incident on a tantalum converter. A three-dimensional configuration of the converter and collimator is shown in Figure 4. In order to obtain accurate results with a modest number of primary electrons, one variance-reduction technique available in ITS is to artificially increase the bremsstrahlung production as electrons slow down in the converter while particle weights are adjusted accordingly so that the results are unbiased. This technique can considerably reduce the computing time by reducing the primary electron histories, since electron tracking is more time consuming than that of photons. More information on this technique can be found in Reference 2.

Timing results, using different numbers of processors, on the PARAGON machine are given in Table 3. These results are based on 512 batches with 25 electron histories per batch and with the SCALE-BREMS factor set to 10000. The load distribution for each processor is fixed; that is, each processor will perform 512/N batches, where N is the number of processors. The run time ranges from 4.5 hours for 32 processors to 21 minutes for 512 processors. As a comparison, the same run will take about 28 hours on an IBM RS6000 Model 560 workstation.

We have also estimated the speedup factors and parallel efficiencies for these calculations. These results are estimates since we did not perform the calculation on a single processor, and the run time for a single processor is approximated by summing the batch CPU times from the multiple-processor run. As shown in Table 4, at 25 histories per batch the speedup is about one half regardless of the number of processors; hence the parallel efficiencies are about 50%. A closer examination indicates that this relatively poor performance is mainly due to the fluctuation in the computing time required to generate the electron trajectories. Since there are only 25 electron histories per batch, batches involving a few, long-running electron histories will run longer than the average. Consequently, the overall computing time is dominated by the longest computing time for a batch. Combining Eqs. (5) and (6), we can estimate the parallel efficiency for N processors by

$$\varepsilon = \frac{S_N}{N} = \frac{T_1}{NT_N} = \frac{N\bar{T}}{NT_{max}} = \frac{\bar{T}}{T_{max}}, \tag{7}$$

where $\bar{T}$ is the average computing time per batch and T_{max} is the longest computing time among all batches. As shown in Figure 5, the ratio between T_{max} and $\bar{T}$ is about two, which validates the parallel efficiency observed.

There are two ways to improve the parallel efficiency. The first one is the dynamic load distribution. As discussed in Section II.C, this technique dynamically adjusts the load, the number of batches executed per processor, so that all processors will finish roughly at the same time. The processors involving long-running batches will execute less batches while the other processors will pick up additional batches. Timing results with the dynamic load balancing are also given in Table 4. The parallel efficiencies are about 90% for small numbers of processors, and decrease monotonically as the number of processors increases. It is noted that the performance of static and dynamic load distribution are about the same as the number of processors increase beyond 256. This is expected since the worst-case efficiency of the static scheme is about 50% and the number of batches executed per processor is two or less.

The second method is to reduce the difference between the average and maximum computing time per batch, which can be accomplished by increasing the number of electron histories. For this example problem, one can obtain similar results with 200 electron histories per batch and a SCALE-BREMS factor of 1000. This approach reduces the disparity in computing time as observed in the case of 25 histories per batch (see Figure 5), hence greatly enhances the efficiency. As shown in Table 4, the parallel efficiencies increase to 89% for 32 processors and 80% for 512 processors. However, since it is more time-consuming to generate electron trajectories, the overall computing time also increases slightly from the previous cases.

V. CONCLUSIONS

We have implemented a parallel-processing capability to the ITS code system. A generic update to ITS 3.0 has been developed which provides users a basic yet flexible platform for their applications. We have performed validation and timing tests in various parallel computing environments. For selected problems, this parallelized version of ITS performs very well. This capability is anticipated to become a standard feature in the future releases.

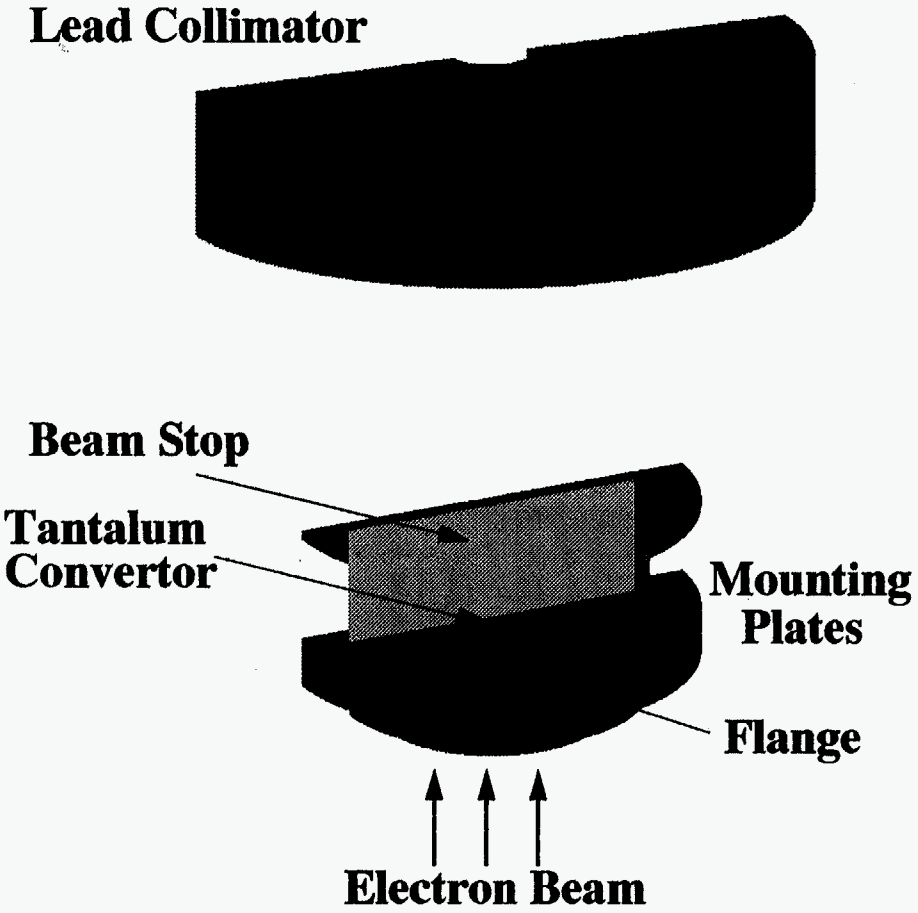


Figure 4. A cross-sectional view of the three-dimensional configuration of the EG&G LINAC bremsstrahlung convertor and collimator.

Table 3. Timing Results of ITS Calculations on PARAGON for the EG&G LINAC Problem.

Number of Nodes	Wall-Clock Time (Hours)
32	4.44
64	2.43
128	1.27
256	0.64
512	0.35
IBM/RS6000 Model 560	28.3

Table 4. Comparison of Speedup Factors and Parallel Efficiencies on PARAGON with Different Number of Particle Histories per Batch and Scale Bremsstrahlung Factor for the EG&G LINAC Problem.

Number of Nodes	Case 1			Case 2			Case 3		
	Wall Clock (Hours)	Speedup[a]	Efficiency	Wall Clock (Hours)	Speedup	Efficiency	Wall Clock (Hours)	Speedup	Efficiency
32	4.44	18	0.56	2.57	30	0.93	5.51	29	0.89
64	2.43	33	0.51	1.36	57	0.89	2.80	56	0.88
128	1.27	63	0.49	0.77	102	0.79	1.43	110	0.86
256	0.64	124	0.48	0.50	159	0.62	0.73	215	0.84
512	0.35	229	0.45	0.35	230	0.45	0.39	405	0.80

Case 1. 25 electron histories per batch with SCALE-BREMS = 10000.

Case 2. Same parameters as in Case1 but with dynamic load distribution.

Case 3. 200 electron histories per batch with SCALE-BREMS = 1000.

a. Speedup is calculated with respect to the elapsed wall-clock time of a single processor for the same number of batches and the SCALE-BREMS factor.

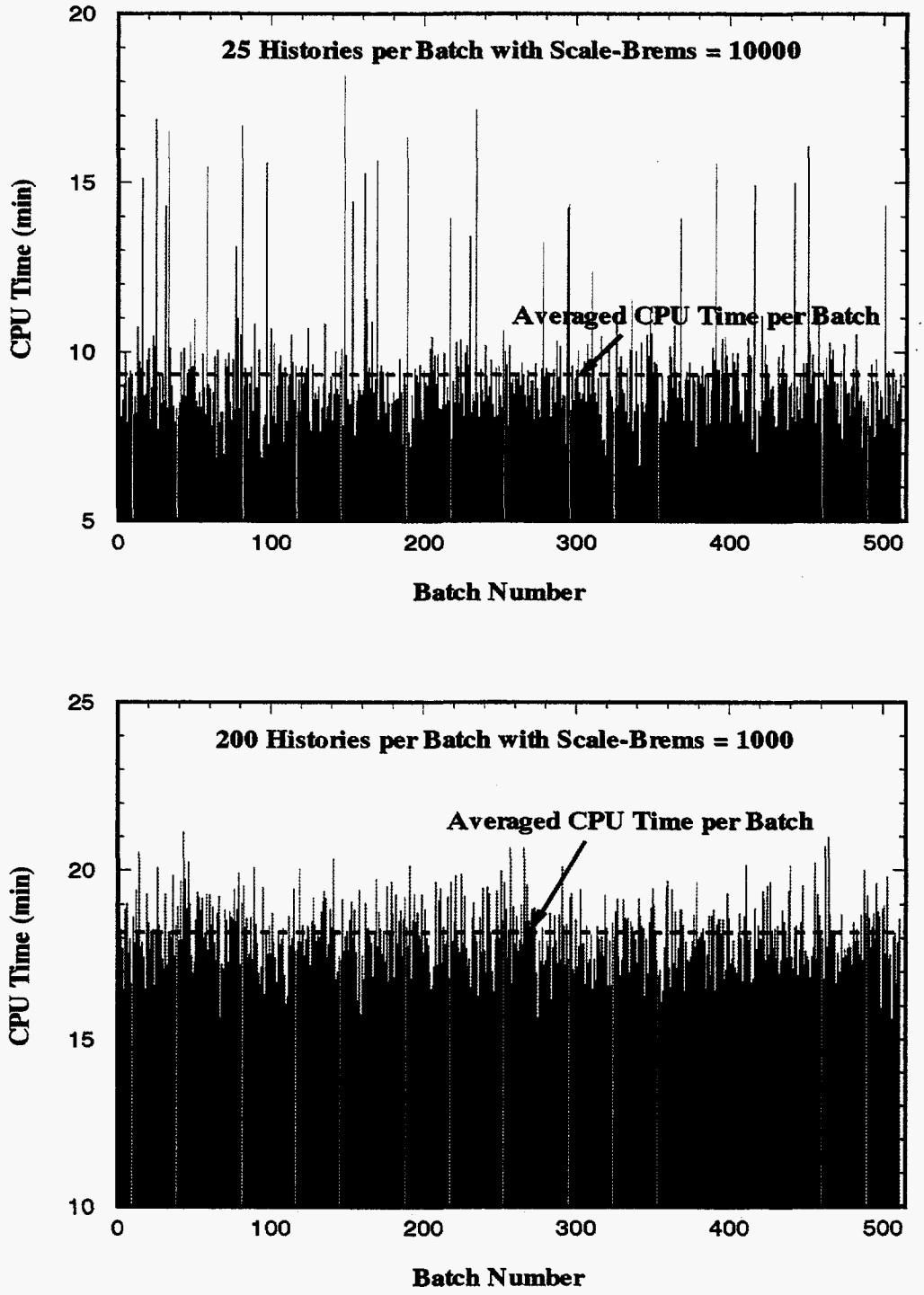


Figure 5. Comparison of computing time per batch for two ITS calculations of the EG&G LINAC problem on PARAGON. This problem was run with 512 batches and with 512 computing nodes. These timing results show that a small number of histories per batch can result in disparity in batch CPU time and thus affect load balancing and parallel efficiency.

I. REFERENCES

1. W. R. Martin, "Monte Carlo Methods on Advanced Computer Architectures," *Advances in Nuclear Science and Technology*, Vol. 22, pp. 105-164, 1991.
2. J. A. Halbleib, R. P. Kensek, T. A. Mehlhorn, G. D. Valdez, S. M. Seltzer, and M. J. Berger, "ITS Version 3.0: The Integrated TIGER Series of Coupled Electron/Photon Monte Carlo Transport Codes," Technical Report SAND91-1634, Sandia National Laboratories, 1992.
3. J. A. Halbleib, R. P. Kensek, and S. M. Seltzer, "Version 4.0 of ITS Electron/Photon Monte Carlo Transport Codes," *Trans. Am. Nucl. Soc.*, Vol. 75, pp. 329-330, 1995.
4. F. James, "A Review of Pseudorandom Number Generators," *Computer Physics Communications*, 60, pp. 329-344, 1990.
5. I. Vattulainen, K. Kankaala, J. Saarinen, and T. Ala-Nissila, "A Comparative Study of Some Pseudorandom Number Generators," *Computer Physics Communications*, 85, pp. 209-226, 1995.
6. Message Passing Interface Forum. "MPI: A Message-Passing Interface Standard," Computer Science Department Technical Report CS94-230, University of Tennessee, 1994.
7. G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 User's Guide and Reference Manual," Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, 1994.
8. K. S. McCurley, "Intel NX Compatibility Under SUNMOS," Technical Report SAND 93-2618, Sandia National Laboratories, 1994.
9. T. A. Mehlhorn and T. A. Hail, "UPEML Version 3.0: A Machine-Portable CDC Update Emulator," Technical Report SAND92-0073, Sandia National Laboratories, 1992.

I. APPENDIX

This appendix provides information on how to obtain the MPI and PVM software, and how to build and execute ITS on various parallel platforms.

Message Passing Interface

There are several freely available, quality implementations of MPI, which support a variety of platforms and communication networks. More information on these implementations can be found at the Mississippi State MPI web page *http://www.erc.msstate.edu/mpi*. Although MPI is a standard, its implementations may vary and are still evolving. Here we employ a portable implementation MPICH (Version 1.0.13) developed jointly by Argonne National Laboratory and Mississippi State University. It can be obtained by anonymous ftp to the address *ftp.mcs.anl.gov*, and the directory *pub/mpi*.

Assuming that the MPICH package is installed on a system and the related files are stored as the following:

- Header files: /opt0/mpich/include,
- Libraries: /opt0/mpich/lib, and
- Executables: /opt0/mpich/bin,

the executable program its.mpi can be built by issuing the shell command

f77 -O -o its.mpi its.f -I/opt0/mpich/include -L/opt0/mpich/lib -lmpi,

where f77 is the FORTRAN compiler on the system.

To run its.mpi on N processors, one can execute the command

mpirun -np N its.mpi < inp > out

where it is assumed that there is a machine configuration file (for example, /opt0/mpich/util/machines) which contains a list of machines where the MPI program can be run.

Parallel Virtual Machine

PVM is developed and maintained by researchers at the Oak Ridge National Laboratory and University of Tennessee, Knoxville. Information on PVM can be found at the web page *http://www.epm.ornl.gov/pvm*. At present, the parallel ITS program is designed to work with PVM3.3 or later versions.

Assuming that the PVM files are stored on a system as the following:

- Header files: /opt0/pvm3/include/X,
- Libraries: /opt0/pvm3/lib/X,

the executable program its.pvm can be built by issuing the shell command

f77 -O -o its.pvm its.f -I/opt0/pvm3/include/X -L/opt0/pvm3/lib/X -lfpvm3 -lpvm3

where X indicates the type of machines and/or operating system.

Before running any PVM programs, one has to start a PVM daemon on the system pool. This can be done using the PVM console or the hostfile option. Reference 6 provides detailed information on this subject and will not be repeated here. To execute its.pvm, simply type

its.pvm < inp > out.

Furthermore, one can use the file PVMTASKS for more control over how processes get started on a UNIX workstation cluster. This option is activated by setting the input parameter NTASKS to a negative number. The format of PVMTASKS is a set of lines of the form

<NHOSTS> # number of host machines available under PVM, and

<NPH> < HOSTNAME>

where NPH is the number of processes to be started on the machine HOSTNAME. The last line should be repeated NHOST times to use all the available machines. An example of such a file, where we want to perform 6 tasks on 3 available machines might be

3

1 sun1

1 sun2

4 sun3

It is important to note that the master process may be running on a machine listed above. Furthermore, the last machine may be a shared-memory multiprocessor which can start multiple processes.

Intel PARAGON

At Sandia, an executable program for the PARAGON must be built on the front-end machine using the cross-compiler. The SUNMOS operating system contains shell scripts that can be used to compile programs written in C and FORTRAN. This is typically done by the following command

sif77 -O -o its.pgon its.f.

The utility yod is then used to load and execute the program:

yod -size N -comm M its.pgon < inp > out,

where N is the number of nodes for the application and M is the memory (in bytes) allocated for the communication buffer on each node.

DISTRIBUTION

1	MS 1165	J. E. Powell, 9300
1	MS 1166	J. H. Renken, 5200
1	MS 1155	W. Beezhold, 9303
1	MS 1159	M. Hedemann, 9311 Attn: Staff
1	MS 1179	J. R. Lee, 9341 Attn: Staff
5	MS 1179	J. A. Halbleib, 9341
1	MS 1167	E. F. Hartman, 9351 Attn: Staff
1	MS 1166	G. J. Scrivner, 9352
1	MS 1166	C. R. Drumm, 9352
5	MS 1166	W. C. Fan, 9352
1	MS 1141	J. W. Bryson, 9361 Attn: Staff
1	MS 1146	T. F. Luera, 9363 Attn: Staff
1	MS 0899	Technical Library, 4414
1	MS 9018	Central Technical Files, 8523-2
2	MS 0619	Review and Approval Desk, 12630