Subsystem Renewal Proposal to SSC Laboratory for R&D of a Parallel Computing Farm

Penn-Princeton-SSCL-Intel Collaboration[1]

K. Anupindi, L. D. Gladney, P. T. Keener,
N. S. Lockyer and M. A. Rezaei
Department of Physics, David Rittenhouse Laboratories
University of Pennsylvania, Philadelphia, PA 10104

J. G. Heinrich and K. T. McDonald
Joseph Henry Laboratories, Princeton University
Princeton, NJ 08544

L. A. Roberts
Superconducting Super Collider Laboratory
Dallas, TX 75237

Justin Rattner
Intel Scientific Computers
Beaverton, OR 97006

[1]Contact Persons: L. D. Gladney and N. S. Lockyer

Executive Summary

The need for massive computing at the **SSC** is well established. A critical design feature is the ability to inject data at high rates into several-thousand-parallel compute-nodes. This is the main design characteristic of the Intel-**MESH** architecture. We have been working with Intel towards a machine that meets the needs of a high rate detector at the SSC. In the last year we have completed the following tasks:

- Major portions of the CERN Program Library including GEANT, as well as ISAJET, PYTHIA, and JETSET have been ported to the Intel iPSC/860 Hypercube. We have provided CERN with these modifications.
- A parallel processor with 4 i860-nodes and 3 I/O-nodes, configured as a hypercube, has been purchased and used extensively.
- Simulations of hypercube I/O properties have been completed using Verilog.
- Demonstration of a large class of random-number multipliers completed. We note CERN RANF fails Knuth's criteria for the spectral test.
- Software is written that implements a parallel-tracking trigger for the SSC.
- We are preparing to use the Caltech Consortium MESH machine (512 i860 nodes with 40 MBytes/sec between nodes).

We propose to demonstrate the data-injection capabilities of the Intel-MESH architecture. We request funding of **$312K** for purchase of a prototype MESH and to conduct research in FY92 with the following major goals:

- Continue a joint-project with Intel to incorporate a FAST-I/O-Board into the Intel-MESH simulation package–Hypersim.
- Demonstrate injection-rate of the MESH using the FAST-I/O-Board.
- Measure time to build events *vs.* injection-rate on the MESH.
- Measure Level-3 trigger rejection times with concurrent algorithms running on the MESH.
- Continue to develop a version of `ISAJET` and `GEANT` that executes simultaneously on multiple-nodes.

Contents

List of Figures

1 Introduction

The SSC detector CPU computing needs exceed that available from industry today[1]. In addition, the I/O requirements at the SSC may be as high as 10^{11} Bytes/sec into the computer farm. For these reasons, massively-parallel-computing architectures have become the subject of study by us and others in the High Energy Physics community[2, 3, 4, 5, 6, 7, 8]. In this proposal, we concentrate on the Intel iPSC/860 based Hypercube architecture and the subsequent machine, the Intel MESH[9]. The Hypercube is a very successful architecture and has been popular in scientific applications over the last several years. However, the number of processor nodes is limited to 128. Future applications at the SSC will require several thousand processing nodes, up to 10^6 VAX MIPS equivalent.

A computer with a new architecture, developed by Intel Scientific Computers Inc., called a 2-D MESH, has been delivered to the Caltech Consortium this last summer[10]. It contains 512 i860 processor nodes with an internode bandwidth of about 40 MBytes/sec. The MESH machine at Caltech is a prototype of a commercial product to be released in 1992 by Intel. This Intel machine will demonstrate capabilities of up to about 2000 nodes with internode bandwidths of about 200 MBytes/sec.

The general goal of this proposal has been to demonstrate the trigger capabilities of RISC processors and the MESH architecture. One of the more challenging areas is B physics. In order to study CP-violating phenomena, the study of most of the B cross section at a luminosity of $10^{32}\text{cm}^{-1}\text{sec}^{-2}$ or greater is desirable. After a simple decision by the Level 1-2 trigger which is built in custom hardware, we attempt to demonstrate that the remaining $\sim 10^5$ events/sec can be injected into the MESH network and the events built. The MESH network is suited for both the high-rate data-injection and the high bandwidth internode communication necessary to build the events. The RISC processors must be powerful enough to validate the event as interesting. We are studying concurrent trigger algorithms within the context of the Intel MESH as a further possibility of enhancing the trigger rejection efficiency.

2 Description of Hypercube at Pennsylvania

We have purchased a seven node iPSC/860 machine containing four compute nodes (RX) and three I/O nodes. Each RX node contains a 40 MHz i860 processor with 16 MB (expandable to 64 MB) of main memory. The RX nodes also contain a Direct Connect Module (DCM) daughter board. The DCM allows the node to communicate with other nodes over the backplane.

The i860 contains a RISC engine, two floating point units and a graphics accelerator. It implements superscalar parallelism and has been measured by Intel to have 80 single precision MFLOPs peak, 60 double precision MFLOPS peak, and 33 integer VAX MIPs. It has a 160 MB/sec peak DRAM access rate. The processor also features a 4 Kbyte instruction cache, an 8 Kbyte data cache and an on board

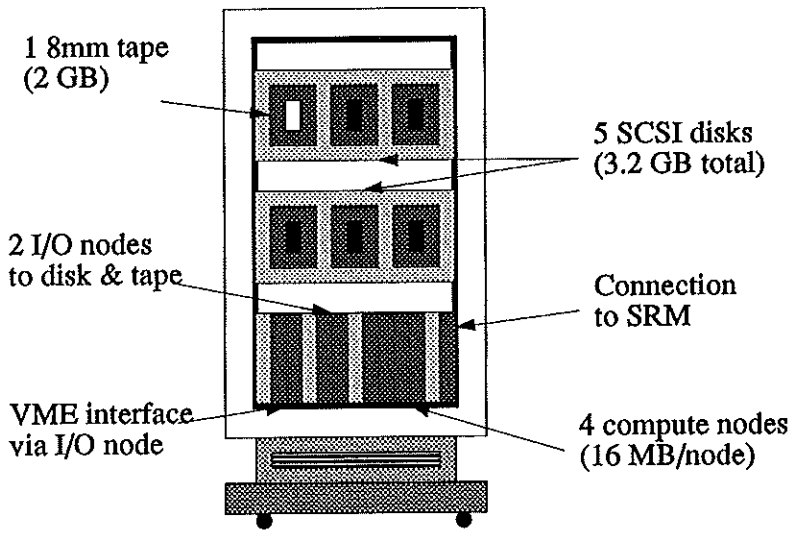


Figure 1: iPSC system configuration

memory management unit.

The DCM has a total of eight channels; seven to connect to other compute nodes and one to connect to other types of nodes such as an I/O node. The DCM can use all eight channels simultaneously and achieve data rates up to 2.8 MB/sec on each.

The DCMs define the hardware communication configuration. The present system is configured into a hypercube. To the programmer, the system appears to be **fully interconnected**. This appearance is achieved by the routing algorithm and the fact the DCM board will offload the communications task from the processor of the intermediate nodes. That is, a node processor is only involved in a communication cycle if it is the sending or receiving node.

Two of the I/O nodes have SCSI ports and connect to five hard disk drives for a total capacity of 3 GBytes and an 8200 Exabyte 8mm tape drive. The third I/O node is a service node that connects to a Bus Interface Adapter (BIA) that allows the user to plug in any 6U VME card. A device driver for the VME card needs to be written for the service I/O in order to access the card. Currently, an Intel supplied ethernet card is plugged into the BIA.

Figure 1 summarizes the configuration of the iPSC system.

Most communication and all process control passes through a front end machine. This front end is called the System Resource Manager (SRM) and currently is a 80386 IBM PC class machine running Intel's version of System V 3.2 UNIX. Requests for node allocations, program loads, etc. get processed by the SRM. This function can actually be performed on a Sun workstation by setting up and running provided software that communicates all such requests to the SRM to be performed on the behalf of the remote user. This requires that the SRM have access to the same disk where the user's code is located on the workstation. This is done with the Sun Network File System (NFS) software.

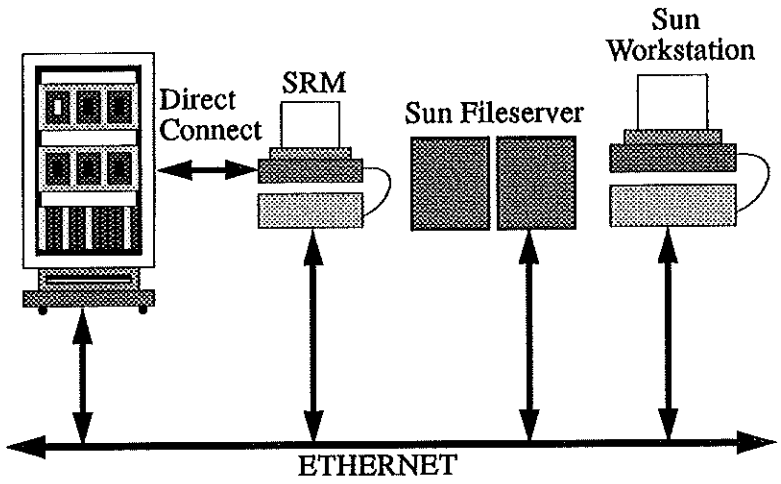


Figure 2: iPSC network configuration

All of the compilers and development software runs on Sun platforms as well as the SRM. This allows for very quick edit — compile — run debugging cycles because of the speed of the Sun workstations. The fact that all of the iPSC control and development software runs on Suns means that a user never has to log onto the SRM at all. At Penn, our workstations and the SRM are served from a central file server. This easily maintains consistency across the iPSC development machines and the SRM. Figure 2 shows the network configuration at Penn.

3 Generators and GEANT - Port to the Hypercube

In anticipation of massively parallel computing becoming available next year we have ported the majority of CERN physics libraries to the Intel iPSC/860 Hypercube at the SSCL. ISAJET, JETSET, and PYTHIA have also been ported. The software for the Hypercube and MESH machines are intended to be identical to the user.

Details of the iPSC/860 supercomputer and the i860 microprocessor can be found elsewhere[9]. A brief description follows.

The Intel iPSC/860 computer consists of compute and I/O nodes. The compute nodes each contain one Intel i860 RISC processor, between 8 and 64 Megabytes of memory and communications board (called a Direct Connect Module or DCM) that allow it to communicate with other nodes. Currently, the version of the i860 processor in the compute nodes runs at a clock rate of 40 MHz. The processor is superscalar (i.e. it can perform multiple instructions in one clock cycle, ax+b).

Each I/O node contains one Intel i386 processor, a DCM board and may contain support for specialized hardware such as a SCSI interface. VME boards may be connected into the machine through an adapter to an I/O node.

We have obtained a small iPSC/860 system to perform the actual code porting and to test various aspects of the system, including the possibility of using a similar machine for online, Level 3 triggering. The ethernet card in the machine allows a

process in the farm to communicate with other machines in the network by TCP/IP protocols via the UNIX socket mechanism. Support is provided for a variety of normal UNIX high level protocols including X11 clients. Also provided is a mechanism for incoming file transfer (ftp) access to the disk in the farm.

Much of the code in the CERN libraries is nonportable. Fortunately, the code already has been ported to a large number of platforms so that in most cases there exists a copy of the code for a machine that is similar to the target system. The changes that need to be made are usually minor and include, for example, specific syntax for common FORTRAN extensions such as hexadecimal constants, and changes in the way that I/O is handled. We are using the Intel Portland Group cross compilers.

3.1 Port of CERNLIB

Version CNL201 of the CERNLIB libraries was compiled and tested. The UNIX DECstation version was used as a basis for the port. This code was used as a starting point because both the processor in the DECstation and the i860 are Little Endian, both in byte and word order, i.e. the order is such that the least significant byte comes first in memory. The word and byte order is a significant difference because many of the mathematical routines and bit/byte manipulation routines take advantage of the byte order to increase their execution speed. Because both machines use the IEEE floating point formats, we did not encounter and difficulty with real number formats. By starting with the DECstation code, the port was made much easier.

Most of the routines compiled without difficulty. The packages that required modification were `COMIS`, `COMIST`, `CSPACK`, `EURODEC`, `GCORR`, `GEN`, `HBOOK4`, `HERWIGT`, `HIGZX11`, `KERNGEN`, `KERNNUM`, `KUIP`, `PAW`, `PAWM`, and `ZEBRA`. The changes required are itemized below.

- Error returns in `READ` and `WRITE` statements.
  Some of these were required because of specific return values from statements. In addition the compiler would not allow implicit goto's on error conditions in the statement.

- Syntax for hexadecimal constants.
  The DECstation FORTRAN compiler uses a syntax of the form `X'1234ABCD'`, whereas the i860 compiler uses `'1234ABCD'X`. Note that hexadecimal constants are not part of the ANSI FORTRAN language specification.

- MIPS assembly language.
  There are a few routines that have been coded in assembly language for the DEC machines. We used the C language versions of these routines extracted from the `PATCHY` deck.

- Use of double precision.
  It was necessary to use double precision in some mathematical routines in order for them to return the correct results.

- File status information.
  The structure returned by the C function `stat`, differs between the DECstation and the iPSC. This is a fundamental difference between SYSV and BSD flavors of UNIX.

- Timing routines.
  The routines that are used to calculate elapsed time were completely rewritten.

- System function call.
  The `system` function call does not exist for the iPSC. It was replaced by a dummy function.

- Word size for I/O and EOF file.
  The word size for RZ files in ZEBRA and the number that represents EOF when reading a file differ between the DECstation and the iPSC.

We have run all of the available tests on the packages. Most of the routines passed without any difficulty. The routines that failed are `C206` (Zeros of Complex Polynomials) in `GENT` and `C308` (Complete Elliptic Integrals K and E) in `KERNNUMT`. The test programs `EPT3L`, `EPT11` and `EPT3S` in `EPIO` also failed. We are looking into these problems. We have also run the BCD `GEANT` simulation [11, 12, 2] (including graphics output) without any problems.

The FORTRAN compiler has several option switches that enable various optimizations for increased speed. All of the routines were compiled with an option (–Knoieee) that allowed the compilers to avoid strictly following the IEEE conventions for floating point arithmetic. The mathematical routines passed the tests and therefore the results are within the accepted limits. The failure of the two routines (`C206` and `C308`) is not related to the use of this option.

The ANSI FORTRAN specification requires the programmer to explicitly issue a `SAVE` statement for the variables to be saved between `CALL`s to subroutines and functions. The FORTRAN routines were compiled with the option (–Msave) that forces the compiler to save the contents of all variables in subroutines and functions between `CALL`s. This was necessary because a large number of the routines in the libraries depend on this feature and assume that the compiler will do it by default.

Our port of the CERN libraries to the iPSC has been transmitted to CERN. It is expected that our code for the iPSC/860 platform will be provided in the next official release of the libraries as an available platform.

3.2 Port of Monte Carlo Event Generators

A number of frequently used physics generators have been compiled, run, and tested on the iPSC/860 platform. These generators are `ISAJET`, `JETSET`, and `PYTHIA`. There were four routines that did not compile due to compiler bugs. These routines are `DECME` and `EPF` in `ISAJET`, `LUDECY` in `JETSET`, and `PYRESD` in `PYTHIA`. In all of the routines except `EPF`, the compiler complains about an "Unmatched ELSEIF, ELSE or ENDIF statement," even though it is not true. The error generated by compiling `EPF` is "Internal compiler error. expand: bad ilm." All four of these routines can be made to compile by avoiding a particular continuation line in the code above the line that generates the error. This bug has been reported to Intel and we are awaiting information on its status.

Another problem can be demonstrated with `ISAJET`. If a user tries to read in an existing command deck that contains no spaces as in

```
PT
1.,20.,1.,20./
```

the read statements in `READIN` will fail and terminate the program. This problem has also been observed in a slightly different form on Sun machines. This can be fixed by inserting the 'space character' after commas. We have not yet determined the correct interpretation of the ANSI FORTRAN specification for this case.

3.3 SSCL Hypercube

The SSCL has purchased a 64-node iPSC/860 Hypercube for beam related studies. We have used this machine for the majority of the code ported by L. A. Roberts. We thank George Bourianoff and his group for use of the machine.

3.4 Benchmarks

We have run several benchmarks on one node of the iPSC/860 and compared them with several popular UNIX workstations. The results are summarized in Table 1.

The first benchmark is the first confidence test for `GEANT`. The second benchmark is the BCD `GEANT` simulation. `ISAJET` was the event generator that was used for the detector simulation. The `GEANT` command deck used was

```
LIST
KINE 2
CUTS 0.05 0.05 0.05 0.05 0.05
GEOM 'SVX ' 'STRA' 'RICH' 'TRD ' 'TOF ' 'ECAL' 'MUON' 'MAGN'
BFLD 1
TRIG 5
STOP
```

Note that the amount of I/O performed was minimal.

The third benchmark is determined by running ISAJET version 6.43 with the TWOJET option. The output data file on the iPSC was sent to the disk internal to the farm. Had we written to a disk on the front end machine or to an NFS mounted filesystem on the front end, the performance would be reduced by a factor of two. The simulation generated $b\bar{b}$ jets with $1.0 < P_T < 20.0$ GeV/c and $\sqrt{s} = 2$ TeV.

In all cases the times are CPU seconds per event that the program reported in its output.

CPU	GEXAM1 Sec/Event	BCD GEANT Sec/Event	BCD ISAJET Sec/Event
IBM	11.72	236.89	0.05863
Intel	12.33	204.14	0.09836
Sun	11.40	175.11	0.06938
SGI	6.11	139.34	0.04663
HP	4.01	81.18	—

Table 1: Results of several physics benchmarks. The times quoted are CPU seconds

The configurations used for these benchmarks are

- IBM — IBM RS/6000-320, 16MB, AIX 3.1.5, XL FORTRAN 2.1
- Intel — iPSC/860, 8MB, FORTRAN compiler PGFTN Sun4/4.0 Rel 1.3a
- Sun — Sun SPARCstation 2, 16MB, SunOS 4.1.1, SunFORTRAN 1.4
- SGI — Silicon Graphics 4D/35TG, 16MB, IRIX 3.3.2
- HP — HP Apollo 9000/720, 64MB, HP-UX A.B8.05

It should be noted that while the times for the iPSC are not spectacular, they are respectable. We expect that new versions of the compilers will take more advantage of the advanced features of the i860. This will improve the performance substantially. It is also important to note that these benchmarks were only run on one node.

3.5 Towards a Parallel ISAJET

We have started work on a parallel version of ISAJET. There are four main areas that need to be addressed.

- Program Control
- Input

- Output
- Random Numbers

The first three bullets state the standard problems of writing software for parallel processors. Program Control refers to the interaction with the user and controlling the actions of the individual processors. This involves specifying the files for I/O, arranging the output of intermediate results, obtaining and combining run statistics (eg. integrated luminosity), and controlling the number of events the various processors generate.

There are three types of input that need to be handled; command decks, static data files (eg. decay tables), and user inputs such as the filenames for command decks and I/O data files. Output data files and listings are examples of output that must be handled.

In order to produce large numbers of statistically meaningful events we need a good source of random numbers. Random numbers and their use in parallel applications are described in detail in section 5.

There is sufficient support in ISAJET to make the program control conversion relatively easy. A process running outside of the farm is set up as the controlling process. This process would be running on the host machine: either the SRM or a Sun workstation. The process would be responsible for controlling the event generation, specifying the files for I/O, and gathering and presenting intermediate results and final run statistics.

The subroutine that contains the event loop can be easily modified to allow an outside process to control the number of events generated and to monitor the event loop.

The routines that collect the run data and calculate the statistics are also easily modified. These functions can be implemented as an interrupt handler so that the controlling process can gather the intermediate statistics without disturbing the event generation.

The changes necessary to handle input in a parallel environment are slightly more complicated. Static data files can be handled by having each node read the file. This is preferable to reading the files once and passing the data between the nodes because it keeps the control and data flow structures simple. It is not clear that either method is more efficient so we choose the solutions that are the easiest to implement.

The specification of filenames by the user can easily be implemented either by the controlling process on the host or through an extension to the command deck mechanism.

The main problem with input is handling the command deck. Most of the commands that can be given are general enough that they can be passed through to the node processes. The problem lies in specifying the number of events to generate and the options for printing event data to the listing file. The user specifies these quantities on a single card that also contains the center-of-mass (CM) collision energy. There are several options for handling this card. It may be ignored (except for

the CM energy parameter) and specified to the controlling process via a keyboard input. Another option is for the controlling process to parse the command deck and regulate the number of events generated.

ISAJET already has facilities for modifying the output data stream; there are several levels of 'indirection' from the routine that decides to write an event and the routine that actually contains the FORTRAN WRITE statement. The iPSC system allows the user to specify an I/O mode where there is exactly one file pointer and each process that accesses the file updates the point appropriately. One output file can thus be written from several nodes without confusing the output stream or overwriting. The output data stream also contains a header and a trailer record that contains various information about the run. This data can be written by the controlling process.

The output data structures need to be modified to contain new data that is important to keep when running generators on several nodes simultaneously. This data includes, but is not limited to, expanded random number generator parameters and the number of nodes used to generate the events. This is a relatively minor change.

There are two problems with listing file output. First, there is a lot of duplicated data: run title, date, various parameters, etc. Second, it is impossible to tell which process generated any given record in the listing file. Also, in the current implementation, the data written to the listing gets written from a large number (about 30) of routines. In order to make a parallel version of the code, each of these WRITE statements would need to be modified. Instead, it is better to pass the data to written out to a single routine that will write it to the listing file. This routine can then decide would to do with the data. We have made this change to ISAJET and sent the results to Frank Paige to include in the regular distribution. A simple addition to this routine will write out the node number that generated the record along with the record to the listing file.

We have started writing the code necessary to make these changes and expect them to be completed by the end of September 1991. We hope to have the parallel iPSC version distributed along with the single threaded version. The concepts and implementation needed to convert the single threaded ISAJET into a parallel generator are the same for most generators and simulators. We expect to complete a parallel version of GEANT for distribution within the next 3 months.

4 Description of Intel Farm Simulator–Hypersim Overview

Hypersim was developed by Intel Scientific Computers to evaluate the performance of various architectural alternatives for its future parallel machines. Details of the simulator can be found in Bain [13]. A brief overview follows.

Hypersim implements some details of the hardware, microcode, and operating

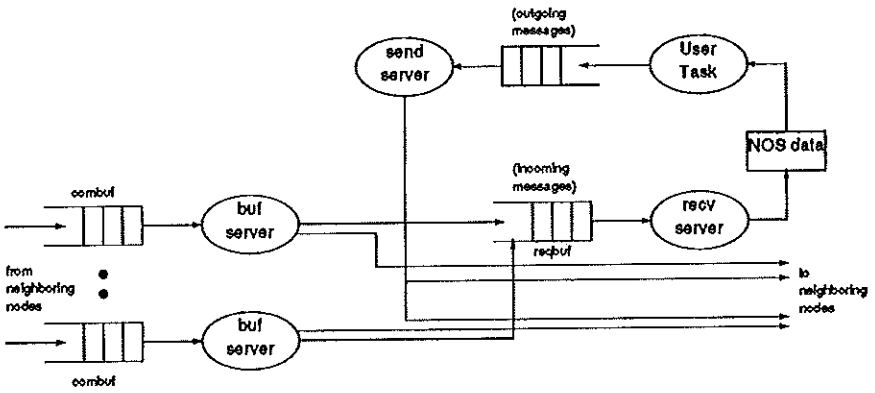


Figure 3: Message Flow Through a Simulated Node

system software of the iPSC machines. It is written in a modular fashion so that the choice of details to implement can change as the needs of the simulation change. This allows the developer to focus on a particular region of interest as the goals become better defined after early stages of the simulation.

The complete interface to the node operating system (NX/2) is provided. Using this interface, developers can construct actual iPSC applications and test their performance on various hardware and software configurations.

The simulator is built on top of the Interwork II [14] concurrent programming environment. Interwork II provides a framework for simulating a large number of nodes with a much smaller number of processors by implementing light weight processes or tasks. These tasks are much more efficient to run and are easier to control then normal heavy weight processes controlled by the CPU. A 1024 node systems can thus be simulated on a 32-node iPSC system with 4 MB of memory per node. This results in approximately 6000 tasks to simulate the architecture.

Interwork II also provides the global name space and time synchronization facilities needed to run the simulation.

Interwork II tasks represent software functions (both user and systems code) and hardware state machines. Global data objects provide the basis for inter–task communication and synchronization. Various parameters are available to change the important characteristics of the hardware and software system architecture.

Figure 3 shows the key Hypersim objects and tasks, and their access relationships. The data objects consist of:

- NOSDATA — This represents the NX/2 data structures. One object exists for each simulated node.
- REQBUF — These queues hold incoming and outgoing user and NX/2 system messages
- COMBUF — These queues buffer all messages passing through the node

The tasks consist of:

- USER TASK — This is the users application program.
- SEND SERVER — This task builds the network level message headers and models the movement of data from memory into the network.
- BUF SERVER — These model the microcode or hardware base function of receiving incoming message pieces and moving them through or into the simulated node.
- RECV SERVER — This task models the software interrupt handler which services received messages for the associate node.

Hypersim currently measures the average latency for messages sent between simulated user processes. This latency has three components: NX/2 overhead, transmission and routing delays due to passing through hardware, and transmission delays due to congestion.

5 Generating Reliable Random-Numbers

In many applications, such as high-resolution Monte-Carlo simulations, it is necessary to produce a large number of random numbers with minimal correlation or repetition. We discuss the conventional Linear Congruential Method and techniques for improving the random sequence. Based on this work, we propose a change to the present multiplier (in RANF) used by `ISAJET` that reduces correlations. A fast and portable random-number generator implemented in C is also presented. Finally we discuss the "leapfrog" algorithm for generating random numbers on an array of parallel processors.

5.1 Properties of a good Random-Number Generator

A good sequence of random numbers should have the following properties:

- There should be a mathematical description of the reliability of the random-number generator.
- It should pass empirical testing criteria, such as the gap test and the coupon collector's test. See Knuth [15] for a complete discussion of empirical tests.
- It should pass a theoretical test like the spectral test.
- It should have a long period. The desired length might depend on the particular application.
- It should be reproducible. In most applications it is necessary to test the program using the same sequence.

- The algorithm that produces the sequence should be efficient with regards to CPU usage.

The need for long period random-number generators is particularly acute for SSC simulations. Large numbers of events need to be generated and simulated for rare phenomenon searches and large statistics experiments. As an example, generating ISAJET events for six different processes are shown in Figure 4. These events were generated with ISAJET version 6.43 and the sample command deck provided with the distribution. In all cases 10,000 events were generated. The CM energy for all processes except HIGGS was 800 GeV, and for HIGGS 40 TeV. The jet P_T limits for all processes that generate jets was $50 < P_T < 100$ GeVand for HIGGS $50 < P_T < 20000$ GeV.

The results show that the average number of random numbers used varies between 1500 and 3000 except for MINBIAS (900) and HIGGS (6800). Hence, a safe upper bound on the number of random numbers needed by ISAJET is 10,000, except for very high-energy HIGGS. A 48-bit random-number generator, as is in common use, generates 10^{14} numbers. This gives approximately 10^{10} events. It is difficult to generate orders of magnitude more events without using 64-bit precision generators or different algorithms.

5.2 The Linear Congruential Method

The linear congruential method is the most common and the most misused random-number generator. This is the algorithm used in RANF, which is used by ISAJET. The formula for producing the next member of the sequence, X_{n+1} given X_n is:

$$X_{n+1} = (a \cdot X_n + c) \bmod m \tag{1}$$

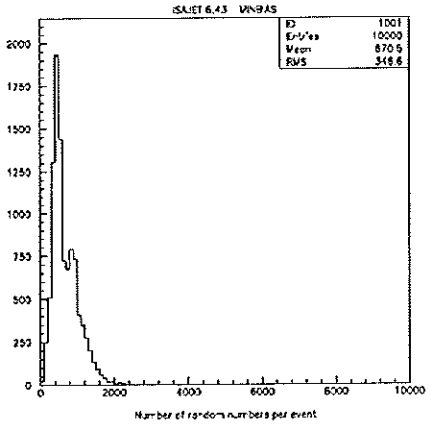
where m, is the modulus ($m > 0$), a is the multiplier ($0 \leq a < m$), and c is the increment ($0 \leq c < m$). Choosing these constants carefully can produce an excellent random-number generator. Unfortunately, the numbers used in most implementations are not satisfactory. However, this method has several advantages: it can be computed very efficiently, if m is chosen to reflect the underlying architecture (*e.g.*, a power of 2, on a binary computer). Furthermore, one can continue the sequence by just remembering the last number produced. In fact, the k^{th} number in the sequence can be computed:

$$X_{n+k} = (a^k \cdot X_n + (a^k - 1) \cdot c/b) \bmod m \tag{2}$$

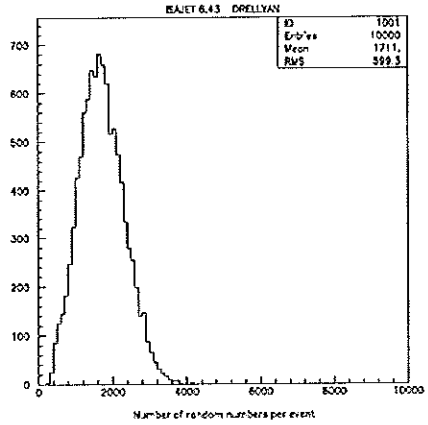
where $b = a - 1$.

For a complete discussion on the theory behind the generator and picking good constants see Knuth [15]. The next subsection summarizes Knuth's results.

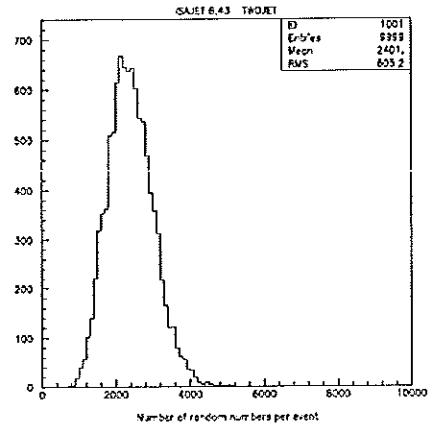
It is important to note that more complicated random-number generators have been proposed. See [16] for several generators achieving considerably longer periods



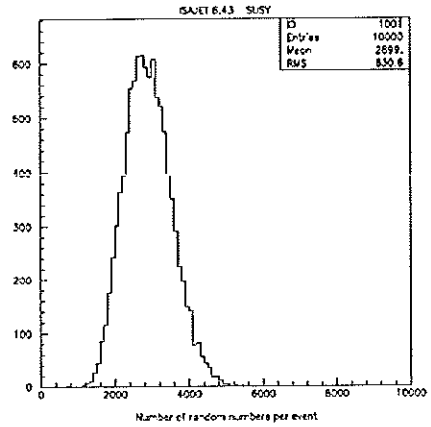
(a) MINBIAS



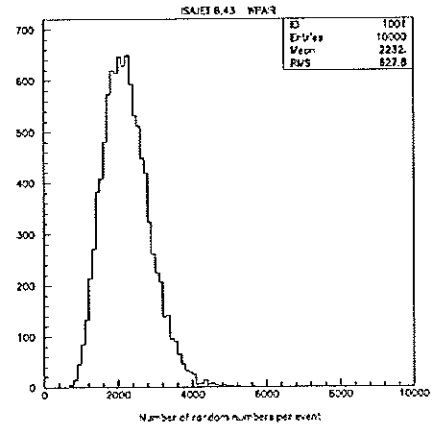
(b) DRELLYAN



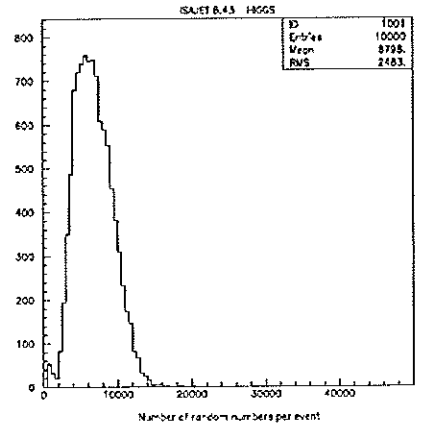
(c) TWOJET



(d) SUSY



(e) WPAIR



(f) HIGGS

Figure 4: Number of random numbers needed to generate events under ISAJET version 6.43

and better "randomness" than the Linear Congruential Method. The disadvantage of these generators is the slower calculation speed (one to two orders of magnitude slower than RANF) and more data must be stored to restart the sequence (typically two orders of magnitude more than RANF). Furthermore, given the methods discussed in [15] and [16], one can construct an acceptable random-number generator from a Linear Congruential Generator.

5.2.1 Choosing the Constants

According to Knuth,
"The linear congruential sequence defined by m,a,c and X_0 has period length m if and only if

1. c is relatively prime to m;
2. $b = a - 1$ is multiple of p, for every p dividing m;
3. b is a multiple of 4, if m is a multiple of 4."

The above is a theorem proven in [15](pages 16-18). It is noteworthy, that the last two criteria reduce to

$$a \bmod 8 = 5 \tag{3}$$

when m is a power of two. Knuth also asserts, "the 'seed' number X_0 may be chosen arbitrarily. ... The number m should be large, say at least 2^{30}. ... The value of c is immaterial when a is a good multiplier, except that c must have no factor in common with m. Thus we may choose $c = 1$ or $c = a$."
Choosing a good multiplier requires testing the multiplier with the spectral test, given in Knuth [15](pages 89-110). The spectral test, basically, measures the correlation of consecutive k-tuples in the sequence. Given a sequence of numbers produced by the linear congruential method, since the period length is finite, consecutive k-tuples form a grid in k-space. The set of points in the grid can be spanned by a set of parallel lines (or planes or hyperplanes, in general k-planes).

5.2.2 Spectral Test

The spectral test measures the distance between these k-planes. The reason for performing the spectral test is that many multipliers can generate sequences that pass the common empirical tests; yet few multipliers pass the spectral test. Correlations in k-space represent the most significant problem with the Linear Congruential Method. The spectral test will produce one number per dimension analyzed (ν_t^2, where t is the dimension, and $\frac{1}{\nu_t}$ is the distance between the t-planes). From ν_t^2 one can calculate the parameter μ_t which is independent of m. For a complete discussion of μ_t, see [15].

This generator will produce over 10^{14} (2^{48}) random numbers before repeating. Knuth asserts that if the multiplier achieves a $\mu_t \geq 1.0$, in every dimension (up to six) it has passed the spectral test. There are many multipliers that pass this criteria.

Therefore, we are able to choose a more strict criteria; see Table 3 for our criteria. For our particular implementation, we choose $m = 2^{48}$, $c = 1$ and a tuple-length k = 6.

t	Max μ_t	33%	50%	67%	75%
2	3.63	1.20	1.82	2.43	2.72
3	5.92	1.95	2.96	3.97	4.44
4	9.87	3.26	4.94	6.61	7.40
5	14.89	4.91	7.45	9.98	11.17
6	23.87	7.88	11.94	15.99	17.90

Table 2: Theoretical maximum μ_t for $2 \leq t \leq 6$, with percentages of maximum. Values of $\mu_t \geq 1.0$ pass the spectral test.

t	Cut-off ν_t^2	Corres. μ_t	% of μ_t
2	'DDA4D3EE8800'X	2.72	75
3	'F7006C00'X	3.97	67
4	'1002250'X	4.94	50
5	'8EFF8'X	3.51	24
6	'10C94'X	5.97	25

Table 3: Chosen Cut-off ν_t^2 with corresponding μ_t. Note, values of $\mu_t \geq 1.0$ pass the spectral test.

Applying the spectral test to over 2^{31} possible multipliers (out of 2^{45}) produced the Tables 5 to 12, included in Appendix B. This was a very CPU intensive task, using over 3650 i860 CPU hours on the Hypercube. All of these numbers passed the spectral test and could be used as the multiplier in RANF.

We have demonstrated that the multiplier used in RANF does not pass the spectral test. We assume, that running the spectral test at the time RANF was written was not practical since the CPU hours required is so large. Although CERN may not want to change RANF at this point, users with demanding requirements may want to consider such a change. We include the results of the spectral test using the multiplier used in RANF and show the results in Table 3.

5.3 Performance of New Random-Number C-Program

We have run our random-number C-program, presented in Appendix B, on both the Sun and iPSC/860. We find a 50% improvement on the Sun over RANF and a factor

t	ν_t^2	μ_t
2	'54BB1EEC93DA'X	1.04
3	'4663553A'X	0.604
4	'68217E'X	0.816
5	'44578'X	0.554
6	'6676'X	0.331

Table 4: Spectral test results for '2875A2E7B175'X used in RANF

of 5 improvement over RANF from the i860. Since ISAJET spends 20% of its time in RANF, our routines provides considerable speedup.

5.4 Generating Random Numbers on Concurrent Processors

5.4.1 Introduction

We are preparing to generate and simulate large numbers of Monte-Carlo events for SSC detector and physics studies. The available computing power of the parallel processor arrays is necessary for this task. We have begun to understand how best to perform this task. The distribution of the random numbers to several nodes was one of the first problems we addressed. A literature search provided some information in this active field and we present a brief summary of our own random-number work.

5.4.2 Method

Generating reliable random-numbers on concurrent processors presents several problems. The most severe problem is sequence overlap. There are several methods for producing reliable random number (using the Linear Congruential Method) on concurrent processors:

1. One processor can produce random numbers and deliver them to all processors.

2. Different nodes can use different multipliers.

3. If there are q processors, each will get $U_{n+q}, U_{n+2\cdot q}, U_{n+3\cdot q}, ...$, where n is the processor number and U_i represents the i^{th} member of the sequence produced. This is called the "leapfrog" algorithm and was originally suggested by Bowman and Robinson [19]. See Figure 5.

The first two methods produce results that pass the standard tests. However, the first method produces a bottleneck when the application requires a large number of random numbers. It also produces significant communications overhead, and the

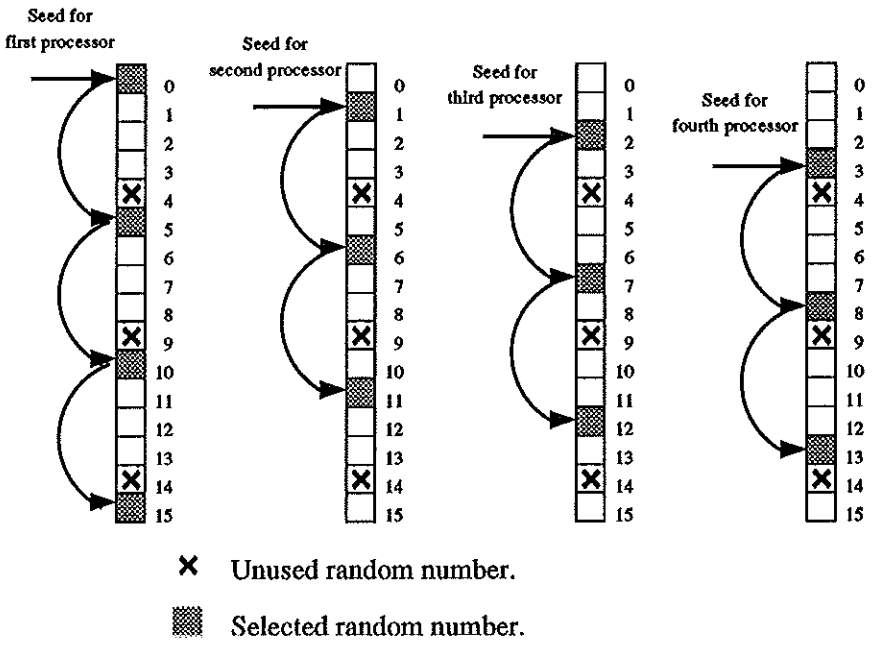


Figure 5: 'Leapfrog' algorithm (adapted from [17]).

results of the simulation might not be reproducible unless information on what processor produced the sequence. The second method has none of these disadvantages, but in order to obtain reproducible results, it is necessary to record the multiplier used to generate a particular set of data. The most significant problem with the third method is whether the sequences in (3) are as random as the original sequence. According to Knuth[15](page 71):
"Experience with linear congruential sequences has shown that these derived sequences rarely if ever behave less randomly than the original sequence, unless q has a large factor in common with the period length. On a binary computer with m equal to the word size, for example, a test of the subsequences for $q = 8$ will tend to give poorest randomness for all $q < 16$."
Therefore we propose to use the smallest prime number greater than the number of processors for our leap value. See Figure 5 for an example of 4 processors using a leap of 5. When using equation 2 to produce these sequences, care should be taken not to lose precision in calculating a^k. Any loss of precision will result in an unreliable generator.

There are other generators proposed in [16] that possess the property that, given appropriate seeds, non-overlapping (independently disjoint) subsequences can be generated. The disadvantage of the proposed generator is the amount of storage space required to record the state of the random-number generator and the long execution-time of the algorithm.

6 An Eight-Node Hypercube Simulation Using Verilog

6.1 Introduction

The message-passing-system of the hypercube was simulated using *Verilog* hardware-description-language. The message-passing-system is a combination of **wormhole** and **e-cube** algorithms. Detailed specifications were taken from information provided by Intel. The MESH machine uses the same routing algorithm as the Hypercube and therefore our experience with the hypercube translates easily to the MESH simulation.

6.2 Overview

The **wormhole** algorithm consists of establishing a complete path between the source and the destination nodes and then sending the message. The e-cube algorithm dictates (deterministically) the path to be taken between two nodes.

The **e-cube** algorithm states that the channels the path consists of must be sorted from lowest channel to highest. For example, if node 0 wants to send a message to node 7, the path will be established by going through node 1 (by channel 0) then through node 3 (by channel 1) then to node 7 (by channel 2). See Figure 6. When

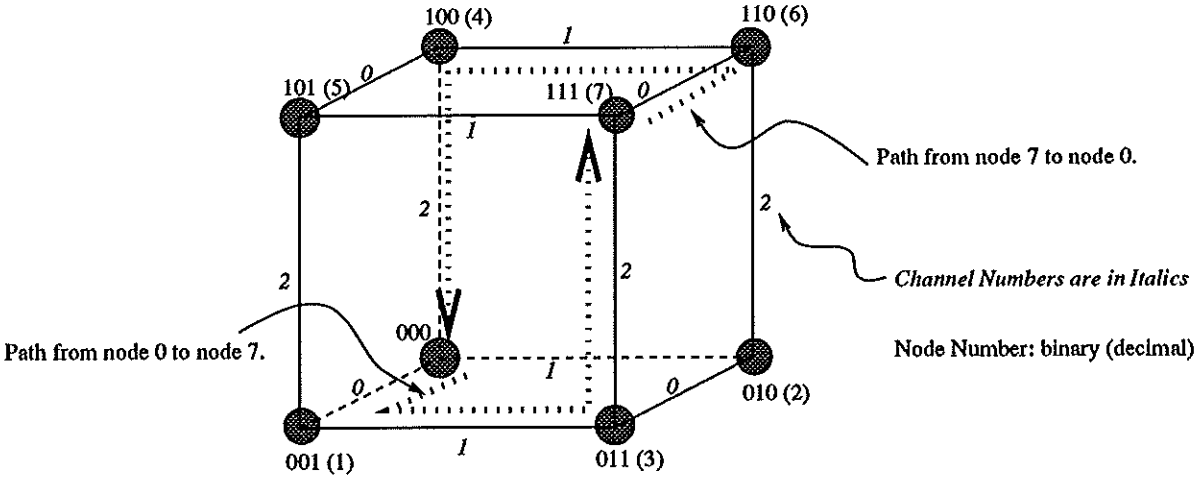


Figure 6: Paths of communication between nodes 0 and 7

node 7 sends a message to node 0, the message takes a completely different path, with no overlaps, since channel 0 coming out of node 7 goes to node 6 and so forth. This is a feature of the e-cube algorithm and one of the reasons deadlocks cannot occur. Messages passed between nearest-neighbors represent the only case when the path is identical.
Sending a message consists of three phases:

1. Establishing a path.

2. Sending the message.

3. Freeing the path.

The path is established by the source node sending a 'routing probe' (which contains the complete routing information, but not the message) to the node next in the path. That node in turn sends the routing probe to the next node, and so on, until the destination is reached. When the destination is ready to receive, it sends an acknowledge signal to the source node. The complete message is then sent immediately. The end of the message automatically frees the path.

6.2.1 The Simulation with no External I/O

The simulation consists of a behavioral model implemented in Verilog. The node-interconnections are not simulated by Verilog's hardware wires. The only information flowing between nodes is the 'routing probe' and 'freeing probe'. (The freeing probe is not in the hardware specifications, but it is convenient for simulation purposes.) In our simulation, there are no messages passed between nodes. The sender CPU simply waits and keeps the channels reserved according to the length of the message it is sending. The simulation-message-passing has the following structure:

1. Send the routing probe and wait for acknowledge signal.

2. Delay according to the length of message/(maximum bandwidth).

3. Send the freeing probe.

Sending of routing and freeing probes are simulated by two global-memory arrays that can be accessed by all nodes. If n is the number of dimensions of the hypercube, both arrays have $(1+n) \cdot 2^n$ elements. Each element of the routing probe has $2 \cdot n + 1$ bits. The MSB is set if there is a probe request. The next n-bits are the source node number. The last n-bits are the channels requested by the source node (which is the source node number XOR'ed with the destination node number; see Figure 6). The source-node number must be sent so that the destination node can send the acknowledge signal.

The freeing-probe array has just $(n+1)$ bits, because it is not necessary to send the source-node information. Acknowledgment is accommplished through a global n^2 element array. One bit is set when the routing probe reaches the destination node.

The routing probe consists of the 'exclusive or' of the source and the destination nodes. The lowest-order-bit that is set in the probe will determine what channel to go through to get to the next node in the path. The next node, in turn will use the next lowest order bit that is set and so forth. (To avoid confusion, when a bit in the routing probe is used, it is set to 0, so the next node examines the lowest-order-bit and does not have to know which node in the path it is.) The freeing probe functions in this fashion as well.

6.2.2 Results of Simulation

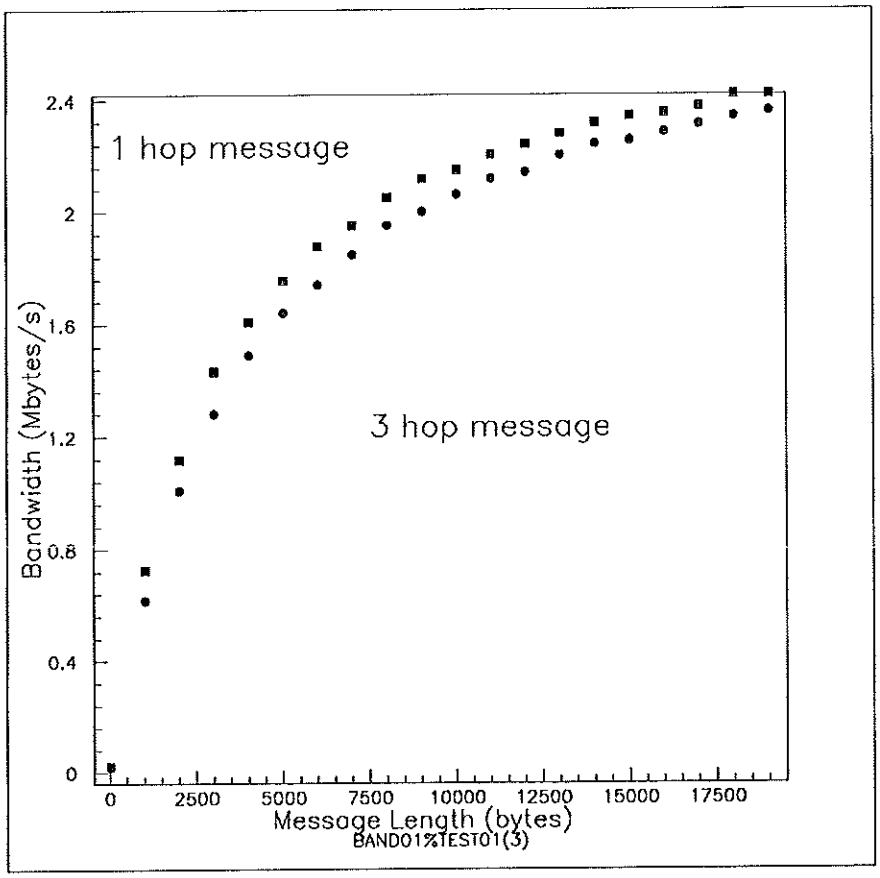


Figure 7: Message Length vs. Bandwidth

Comparing the results from Figures 7 and 8 to the measured bandwidth and message latency [20], shows an almost identical match. There are three parameters that were inferred from the measured graphs: the delay at the source DCM (Direct–Connect Module) to set up the channel, the delay at the destination and the delay at the intermediate nodes. These were set to 1 ms, 1 ms and 0.1 ms respectively.

6.3 Simulation with I/O–Event Building

6.3.1 Statement of the Problem

We are interested in distributing a large, continuous stream of data between nodes in the hypercube. The hypercube performs external I/O using special I/O nodes. The purpose of the simulation is to find the configuration of the I/O nodes, such that the data can be distributed to the compute-nodes as-fast-as possible.

Since the network is not completely connected *i.e.*, not every node has a direct connection to every other node, channel contention will result. To maximize the

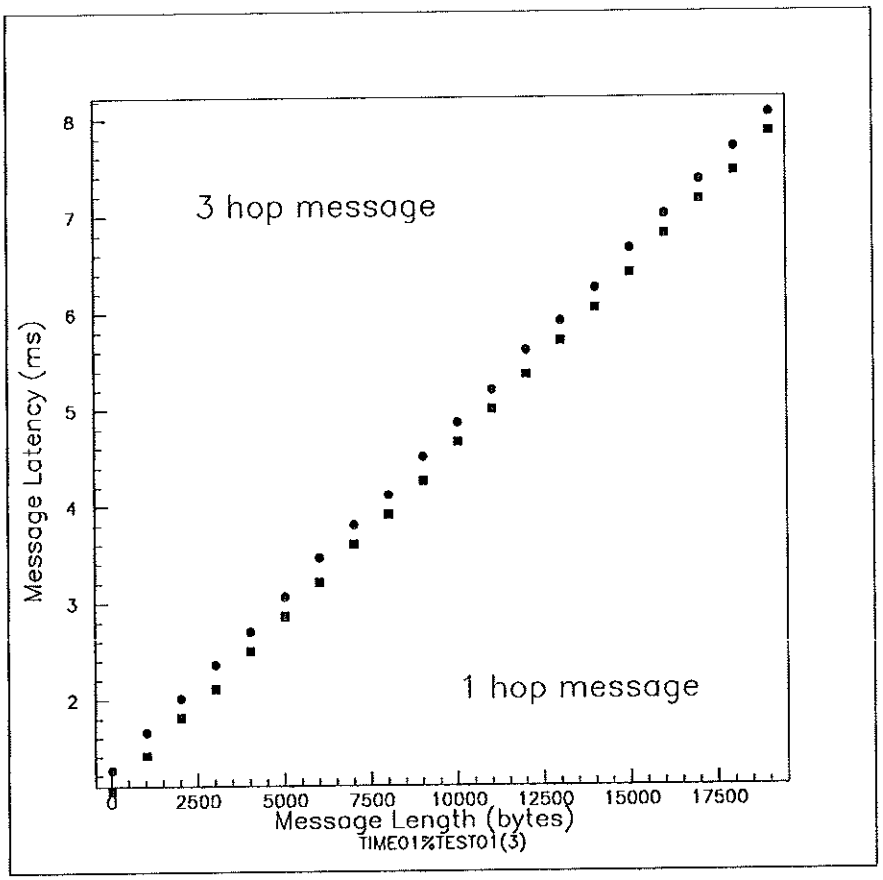


Figure 8: Message Length vs. Bandwidth

speed of data distribution, channel contention must be minimized. Placing the I/O nodes in different configurations results in different patterns of contention. We assume the incoming data consists of discrete sections spread over k input lines, such as that expected from the event data in a detector system. In order to analyze a single event, we collect data ($\frac{1}{k}^{th}$ per line) in a single compute node. For example, if we have 2 incoming data-lines and 6 compute-nodes the first and second halves of the event data from lines one and two respectively go to the first compute node, data from the second event to go to the second compute-node and so forth in a round robin fashion.

We have considered two basic models of data injection or I/O node configuration; the Injector Model and the Distributed Model. Although the simulation was based on the Hypercube architecture, the results will hold for any network that uses the E-Cube routing algorithm.

6.3.2 Injector Model Description

In the Injector Model there are three types of nodes: injector, processor, and the I/O. The detector injects event records directly into the system network through a

number of specially designed buffers, the injector nodes. These buffers balance the burst speed of the detector against the average speed of the network, and route each injected message to the appropriate (predesignated) processor. The injector node would take the place of a processor in an an otherwise completely-connected interprocessor network. The processor nodes are the i860 computers. The I/O nodes are also i860's and allow output to a permanent storage device or input for testing and debugging the network.

A schematic of the Injector Model is shown in Fig. 9.

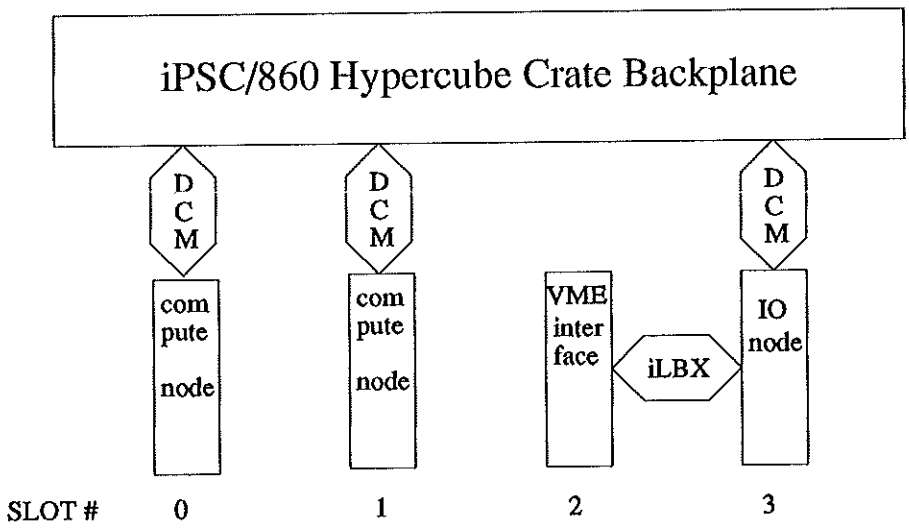


Figure 9: Preparatory Injector Model.
Shown is the VME interface connected through the iLBX to the I/O node. This is similar to the Injector Model and can be achieved by rewriting the software driver.

6.3.3 Description of Distributed Model and Fast-I/O-Board

The second approach is called the Distributed Model and is shown in Fig. 10. A special interface connects a selected number of processor nodes to the detector. This board is presently under design by Intel and this collaboration is charged with incorporating the board's functionality into Intel MESH simulator. In Fig. 10, the Fast-I/O-board is connected directly to the I/O node. This allows a more flexible arrangement of the input data channels, but more importantly, connects the data or event to the network, where transfer to any number of compute-nodes can take place without the overhead of transfers. This interface will allow standard processors to act as injectors as well as computation nodes. This would give a great deal of flexibility in setting up the optimal configuration for the network.

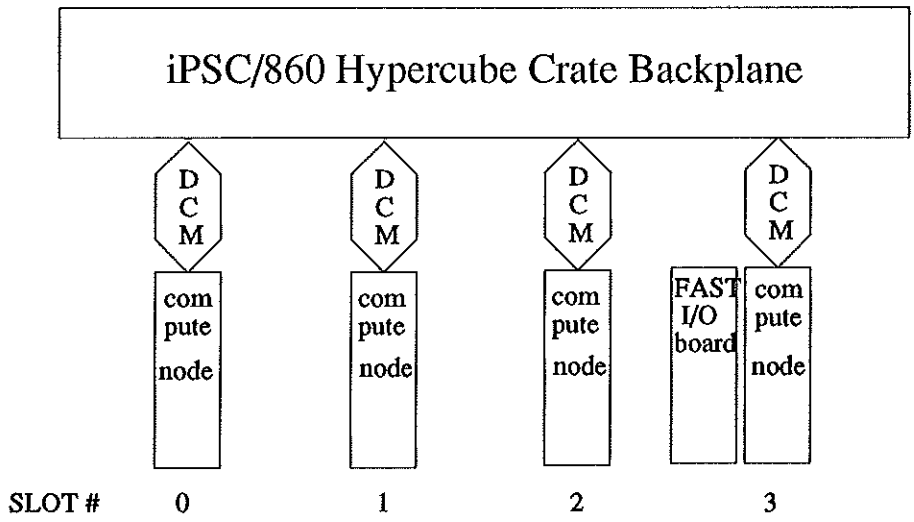


Figure 10: Distributed Model.

The Distributed Model connects the input data stream directly through the Fast I/O board to a compute-node. Implementing this scheme requires new hardware presently being designed by Intel and simulated by this collaboration.

6.3.4 Simulation of an 8 Node Hypercube with 2 or 4 I/O Nodes

Given an eight node hypercube and two or eight incoming data lines, how should we position the I/O nodes such that the data communications are most efficient? With 2 I/O nodes, there are three possible configurations. See Figure 11.

With 4 I/O nodes, there are six possible configurations. See Figure 12.

The simulation on two I/O node case shows that configuration (c) is the best. The result is more or less intuitive; the simulation simply provides confirmation. Since the two nodes are furthest apart, there is very little channel contention.

The simulation on the four I/O node case shows that configuration (f) is the best. Although this result is not as intuitive as two node case, it is an extension of the same criteria: the nodes are as far apart as possible. However, if we calculate the total distance in case (e) we arrive at the same number as in case (f), namely 12. The reason why case (f) is better than case (e) is because the nodes are more evenly distributed. Based on these examples, we propose the following criteria for selecting I/O nodes:

1. Maximize the total distance between the I/O nodes.

2. Minimize the standard deviation of the node distances.

(The total distance is defined as the sum of the distances between the I/O nodes, taken two at a time.)

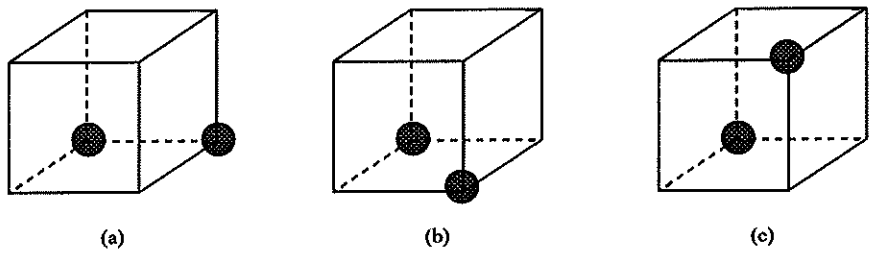


Figure 11: All possible 2 I/O node combinations.

6.4 Real Time Analysis of Data from SSC Detector and Data Acquisition Systems

6.4.1 Introduction

Figure 13 shows a schematic of the flow of data from the detector to storage.

The hardware trigger(s) eliminate uninteresting events using cursory criteria. The data-rate coming into the computing farm should be about 100 Gigabytes/s. The size of each event is approximately 1 Megabyte. So the farm must be capable of handling 10^5 events per second. The farm must determine if the events pass particular criteria, and reject those events that don't meet these criteria.

6.4.2 Two Possible Solutions to Event Building and Analysis

There are two very different solutions to the problem of event building and analysis in a parallel supercomputer. These methods are based on **data decomposition** and **functional decomposition**. Data decomposition, in its simplest form, consists of all nodes of the system running the same program on different data. In this case, data decomposition translates to every node evaluating complete events. If the problem is solved using functional decomposition, however, different nodes will be assigned different tasks. In this case, each node (or several nodes) could evaluate the data from a particular detector subsystem.

1. **Data decomposition:** collecting the data from the same event in a single node. It is important to note that existing systems build events using exotic (and expensive) hardware switches, similar to telephone network switches. At the data rates for the SSC detector system, this is not a very feasible option. Therefore, we must build the event in the farm. This model has the following

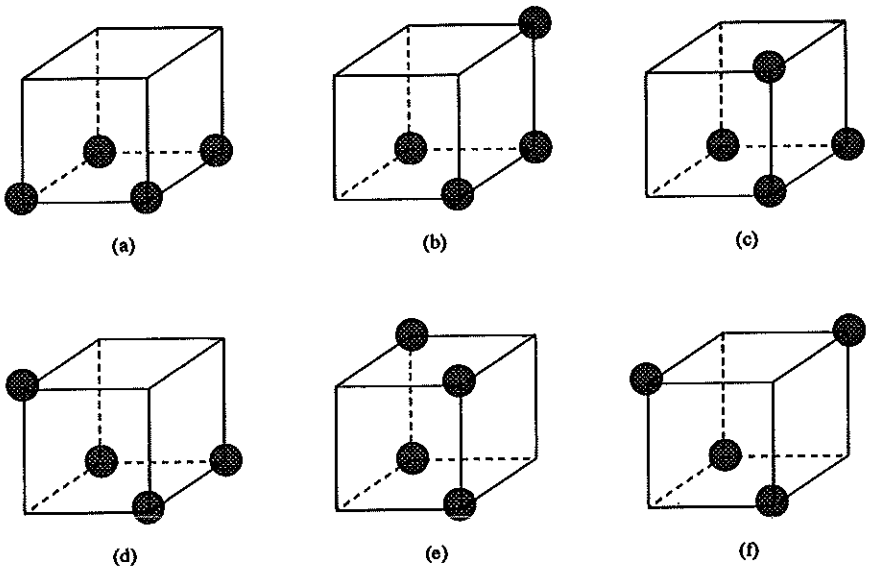


Figure 12: All possible 4 I/O node combinations.

advantages and disadvantages:
Advantages:

- Easy to understand/work with/program.
- Scalable.

Disadvantages:

- Number of compute nodes is limited by both the network bandwidth and compute speed. Choosing the maximum of the two will waste resources, since these two parameters are unlikely to require same or similar number of nodes.
- Very large amount of communications is required. Small message sizes corresponding to parts of events can cause a reduction in real network bandwidth.
- Further delays are caused when many I/O nodes are trying to communicate with the same compute node, as is the case most often in this model. Interleaving the communications will not help appreciably.
- Since all compute nodes get the same number of events/second, easy rejections are not taken advantage of.

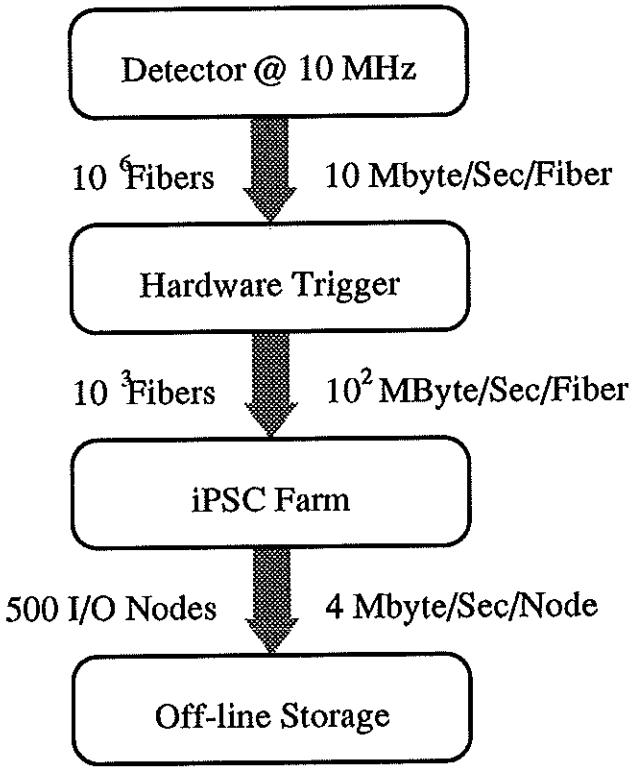


Figure 13: Data flow from detector to off-line storage.

2. **Functional decomposition**: distributing the data by detector subsystem. This model would require a-priori knowledge of relative data rates from detector subsystems.
   Advantages:

   - Rejections can save computation and therefore time, since the computation on a single event is done in parallel
   - The modularity of the model lends itself well to large grain scalability.

   Disadvantages:

   - Harder to work with/program.

   Parameters to be investigated:

   - Amount of communication required to assess a particular subsystem.
   - Distribution of data among processors dealing with the same subsystem.
   - Number of I/O and compute nodes per detector subsystem. Dynamic allocation of nodes per subsystem could be an option.

7 Concurrent Tracking-Trigger-Algorithm

7.1 Introduction

We have written a tracking-trigger algorithm that runs concurrently on the iPSC/860. The algorithm is based on the format of data that comes from the large drift-chamber of CDF. Using the CDF as the tracking detector permits comparison with custom hardware processors that perform the identical task. High-P_t track segments are found by searching for hits which follow the wire *vs.* time patterns expected for straight (i.e. large P_t) trajectories. The track segments are then linked to form the entire particle trajectory and obtain a measure of the particle P_T. Presently, we have invoked a 2-D algorithm. In the future we will extend the algorithm to 3-D.

Two-dimensional track reconstruction with good momentum resolution can produce a factor of 1000 reduction in fake electron trigger rates when combined with calorimetric information. Successful implementation of an algorithm that reconstructs charged tracks in < 50 μs would influence the design the of the Level-2 trigger for the SSC.

7.2 Implementation and Algorithm

The algorithm begins by forming patterns of hits from the data. It then compares these patterns with a library of interesting patterns and stores the the patterns or candidate track segments that match well. In more detail, the algorithm compares arrays of 8-byte words from a Data-file, representing hits from a drift-chamber cell, with a similarly formatted Mask-file, which is composed of calculated arrays of numbers representing interesting patterns of hits from the drift-chamber.

The following conditions are to be met before a number can be selected to be compared with the numbers of the mask file:

1. Two adjacent bits of a number cannot differ by more than 4 time bins. This is a P_T cut on the track in the cell.

2. The time bins must follow roughly a chevron shape. We require any change in direction to take place in time bins 0,1,2,3 or 4. This is explained in the Fig. 14 and results from the fact the sense-wire plane is rotated 45° with respect to the radial direction.

3. There should be at least 4 non-zero entries in the word. This is a simple requirement on the minimum number of hits in a cell and helps reduce spurious track segments.

The data file consists of arrays of 8 byte numbers, each representing the time bins of the hits. A cell consists of up to 12 wires and each cell has a maximum of 8-entries for each wire. We then form all possible combinations of these entries for

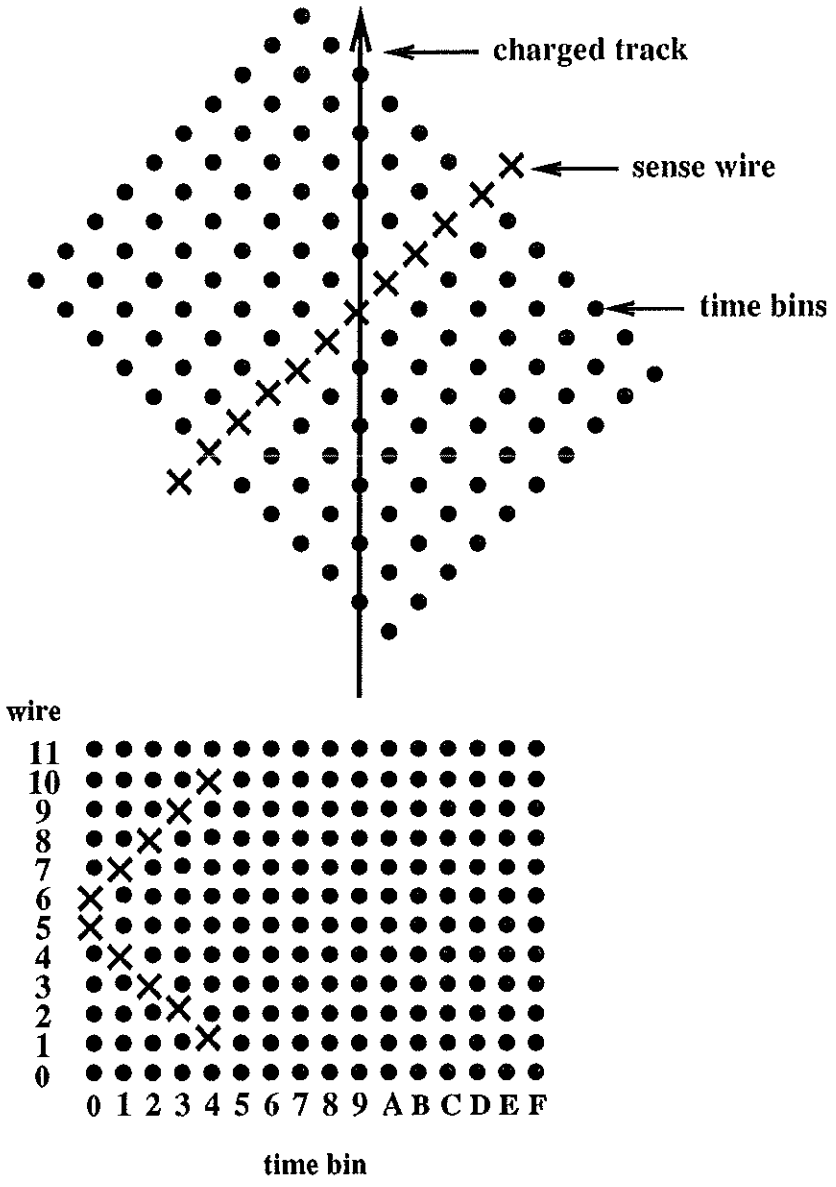


Figure 14: Schematic of Wire number *vs.* Time Bin.

each cell. This formation of combinations or candidate track segments has used the following algorithm:

1. The total number of patterns that can be formed are calculated first.

   The total no. of patterns = product of no. of entries in each wire.

   Example:

Wire No.	Word	Wire Length
Wire 1	00001a2	3
Wire 2	0001212	4
Wire 3	0000001	1

   Hence, the total number of patterns possible = 3*4*1 = 12

2. - All wires of the same cell are stored in a link-list. The digits of that wire are stored as arrays (string). The array index of each element in the list is decreased till it reaches zero.
   - After reaching zero, the array index of the predecessor of this element is decreased and the array index of this element is initialized to its number of wires.
   - The array index of this element is decreased until it reaches zero.
   - The procedure is repeated till the array index of the first element in the list reaches zero, indicating that all combinations have been formed.

3. After forming the linked-list, the combinations are stored in a different structure (struct compare). The first combination is stored and the number is searched. Then it is overwritten by the next combination and the loop continues till all the combinations are completed.

4. Matching is done using the logical AND. The output is considered a valid match if there are at least four non-zero bytes.

7.3 Super Cell by Super Cell Parallelization

The Intel Hypercube or MESH architecture permits a simple and obvious parallelization of the tracking data. We have divided the tracking detector into cells. We consider each cell as independent and assign the cell to its node. The algorithm then finds all track segments associated with that cell. We have run the algorithm and are presently optimizing the performance.

8 Budget Proposal for FY92

This last year we purchased a 4-node Hypercube system with software under the Intel Universities Partners Program. The Partners agreement permitted a discount of $91,330 in addition to the 10% discount given to universities. In order to qualify for the Partners Program, we have undertaken a joint project with Intel. We have agreed to write an I/O section for their MESH simulation program Hypersim.

In order to purchase the machine, Intel carried part of the cost over to FY92 so we could receive and use the minimal system. The attached letter from Intel states this arrangement and indicates the second payment is $56,245. The letter from Intel also states we will receive 100% credit of our present iPSC/860 System price toward the purchase of the MESH machine. The present arrangement was very favorable to us and we thank Intel for their flexibility.

In the past year, funds from DOE have been augmented with PYI funds from the NSF to purchase two Sun workstations associated with the iPSC/860 system. In addition we received an award through the TNRLC which was used to hire one FTE programmer, K. Anupindi, who works 100% on parallel programming issues for this project.

If funding is approved in FY92, we plan to purchase a small prototype MESH machine. A limited number (about 10) of these prototype MESH machines will be available from Intel to selected research partners. The MESH, when released as a product, is intended to be sold only as large systems. The research machine cannot be described in detail since we have signed a non-disclosure agreement with Intel. It is a 6-node i860+ node MESH architecture machine that will include hardware that permits data to be injected at high rates i.e. rates comparable to the internode bandwidth.

With the funds requested for FY92, we propose to demonstrate the Level 3 trigger capabilities of the MESH for low and high P_T events. We will measure data injection rates into the MESH machine and implement the event builder algorithm. Then based on concurrent algorithms, we will estimate the rejection capabilities of the MESH as a Level 3 trigger. The prices are estimates from Intel. We list the items and the estimated costs.

1. Permanent Equipment

    1. MESH machine with 6 i860+ nodes $119k
    2. Two Sun IPC SPARC stations $10k

Total Equipment $129k

2. Salaries

    1. Programmer (U. Penn) $45k
    2. Computer Science Graduate Student (U. Penn) $15k

Total Salaries **$60k**

Indirect Costs on Item 2 ..... **$40k**

3. Final Installment on Hypercube **$56k**

4. Materials, Supplies, and Travel

    1. Travel for U. Penn personnel $10k
    2. Travel for Princeton personnel $6k

Total Materials, Supplies, and Travel **$16**

Indirect Costs on Item 4 ..... **$11k**

Total **$312k**

We also present the budget summarized by institution:

1. U. Penn

    1. Equipment $129k
    2. Salaries $60k
    3. Travel $10k
    4. Indirect Costs $47k
    5. Final Payment on Hypercube $56k
    6. **Total U. Penn** **$302k**

2. Princeton U.

    1. Equipment $0k
    2. Travel $6k
    3. Indirect Costs $4k
    4. **Total Princeton U.** **$10k**

3. **Total** **$312k**

9 Responsibilities and Personnel

9.1 Pennsylvania

P. Keener (programmer) works 75% time and K. Anupindi (programmer) works 100% time on this project. M. Rezaei is a computer-science student and will perform his senior thesis on the data-flow simulations. L. Gladney and N. Lockyer spend 25% time on this project.

- P. Keener will continue development of High Energy Physics software for use with the MESH and Hypercube machines. The goal is to develop a version of generators and `GEANT` that runs on many nodes simultaneously. Documentation will be provided for the user community.
- P. Keener and L. D. Gladney will be responsible for the data injection I/O studies into the MESH.
- M. Rezaei and N. Lockyer will demonstrate the capabilities of the event builder on the MESH and determine the Level-3 trigger rejection capabilities.
- Demonstration of a vertex-trigger concurrent algorithm will be performed by L. D. Gladney and K. Anupindi.
- `GEANT` simulations will be done by J. G. Heinrich and K.T. McDonald.
- The Intel Hypersim I/O-simulation will be demonstrated by P. Keener and N. Lockyer.

It is planned that the MESH system will be located at the University of Pennsylvania.

- We have occupied 600 square feet of floor space and have installed the iPSC/860 System.
- Fire protection and climate-controlled environment exist.

9.2 Princeton University

J. Heinrich and K.T. McDonald spend 20% time on simulation studies.

The use of the hypercube in this past year was possible because of the interest of Professor S. A. Orszag of the Program of Computational and Applied Mathematics. We are very grateful for his support. This machine contains 32-nodes and is still very useful for studies. We hope to continue to use this machine for the next several months at 5% level to study the injection of data from the array local disk to the network and develop software.

9.3 SSCL

- L. A. Roberts will continue development of High Energy Physics software for use with the MESH and Hypercube machines. Documentation will be provided for the user community.

9.4 Intel Corporation

The Intel Corporation considers the development of large processing farms a major part of their program. Intel Scientific Computers has:

- Provided an expert consulting and assistance in developing simulations and models of the processor farm. (W. Bains)
- Made available a system expert during the installation stage at the University of Pennsylvania. (R. Enchelmeyer)
- Provided access to a very large system at Caltech for testing of algorithms and network simulation tests. (P. Messina)
- Provided access to the MESH machine at Intel.
- Provided a lead scientist to co-author scientific papers that summarize the work done during this proposal.
- Provided parallel-programming seminars.

A A Fast Random-Number Generator in C

As mentioned earlier, one of the advantages of the Linear Congruential Method is that it can be implemented in a very CPU efficient manner. We would like to return a normalized random number (between 0 and 1). Since we need to reseed the sequence with the same number that the generator routine returns, all significant digits must fit within the precision of the returned number. The data representation with the most significant figures is a double precision floating point number, with 52 bits of mantissa precision (ANSI/IEEE 754-1985 standard). So choosing $m = 2^{48}$ is quite safe. Unfortunately, doing 48–bit multiplies requires 96–bit precision. Therefore, we must represent the numbers (a, m, X) in base 2^{24}, *i.e.*using two doubles. To minimize the number of divisions and multiplies by 2^{24}, we represent the numbers as:

$$X = seed_1 \cdot 2^{48} + seed_0 \cdot 2^{24} \tag{4}$$

$$a = a_1 \cdot 2^{24} + a_0 \tag{5}$$

$$c = c_1 \cdot 2^{48} + c_0 \cdot 2^{24} \tag{6}$$

where

$$0 \leq seed_0 < 1$$
$$0 \leq seed_1 < 1$$
$$0 \leq a_0 < 2^{24}$$
$$0 \leq a_1 < 2^{24}$$

We can pick $c_1 = 0$ and $c_0 = 2^{-24}$ to speedup computation.

To further decrease the computation time, we can take advantage of the ANSI/IEEE floating point representation as follows:

- Division by 2^n is equivalent to decreasing the exponent by n.
- Multiplication by 2^n is equivalent to increasing the exponent by n.
- floor(x)[2] is equivalent to adding 2^{52} and subtracting 2^{52} and possibly decrementing by one, since ANSI/IEEE standard specifies rounding, not truncating of lost digits.

Care should be taken when increasing/decreasing the exponent to have the correct byte ordering (little–endian versus big–endian [3]). Also, the exponent of zero should not be increased/decreased.

[2] floor(x) is the largest integer less than x.

[3] On most computers memory is organized in 8-bit chunks (bytes) in consecutive memory locations. When a quantity with more than 8-bits, such as a 32-bit integer, is to be stored, it is possible to store either the lowest order byte first (little–endian) or the highest order byte first (big–endian)

The program is written to be portable, with no modifications, to any platform that supports double floating-point-numbers with at least 49 bits of mantissa precision. The hardware specific features can be turned off by defining the FLOOR and PORT preprocessor variables.

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

/*
   You can define three variables to choose the target machine
   architecture:

   BIGENDIAN: for bigendian byte order and ANSI/IEEE compliant
              double (64 bit) floating point numbers, e.g. i860.
   FLOOR: to use floor() as opposed to adding and subtracting
          2**52 to get the floor of a double
   PORT: to use division instead of bit twiddling to divide by 2**24

   Default is for little endian ANSI/IEEE 64 bit floats and not using floor.

   for exmaple
   cc -c random.c -DPORT -DFLOOR
   should work on any machine that supports double of at least 48 bit mantisa

   while

   cc -c random.c -DBIGENDIAN
   should work only on machines with bigendian ANSI/IEEE doubles.

M. Rezaei  August 1991
*/


double seed0=16651885.0/(1L << 24);
double  seed1=2868876.0/(1L << 24);

double a1=2651554.0;
double a0=15184245.0;

double mr_ran()
{

  double y1,y2;
  static unsigned short pow24=0x180;
```

```
  static double two52=4503599627370496.0;
  static double c=1.0/(1L << 24);
  double t;

  y1=a0*seed0+c;

  if (y1>1.0)
    {
#ifdef FLOOR

      y2=floor(y1);

#else

      y2=y1+two52;
      y2-=two52;
      if (y2>y1) y2--;

#endif

      y1-=y2;

#ifdef BIGENDIAN
      *(((unsigned short *) &y2)+3)-=pow24;
#elif defined PORT
      y2 /= ((double) (1L << 24));
#else
      *((unsigned short *) &y2)-=pow24;
#endif


      y2+=a0*seed1+a1*seed0;
    }
  else
    {
      y2=a0*seed1+a1*seed0;
    }

#ifdef FLOOR

  t=floor(y2);
  y2-=t;
```

```
#else

  t=y2+two52;
  t-=two52;
  if (t<=y2) y2-=t;
  else
    y2=y2-t+1.0;

#endif

  seed0=y1;
  seed1=y2;

  if (y1>0)
    {
#ifdef BIGENDIAN
      *(((unsigned short *) &y1)+3)-=pow24;
#elif defined PORT
      y1 /= ((double) (1L << 24));
#else
      *((unsigned short *) &y1)-=pow24;
#endif

      y2+=y1;
    }
  return(y2);
}

void mr_reseed(newseed)
     double newseed;
/* 0 < newseed <= 1 , just like the output of mr_ran*/
{
  double y1,y2;
  static unsigned short pow24=0x180;
  static double two52=4503599627370496.0;

  y2=newseed;

#ifdef BIGENDIAN
  *(((unsigned short *) &y2)+3)+=pow24;
#elif defined PORT
  y2 *= (double) (1L << 24);
#else
```

```
  *((unsigned short *) &y2)+=pow24;
#endif

#ifdef FLOOR

  y1=floor(y2);

#else

  y1=y2+two52;
  y1-=two52;
  if (y1>y2) y1--;

#endif

  seed0=y2-y1;
  if (y1>0)
    {
#ifdef BIGENDIAN
      *(((unsigned short *) &y1)+3)-=pow24;
#elif defined PORT
      y1 /= ((double) (1L << 24));
#else
      *((unsigned short *) &y1)-=pow24;
#endif
    }

  seed1=y1;
}


void mr_seta(s)
     char *s;
{
  char s2[20];
  int i,j;
  long val;

  for(i=0;s[i]!='\0';s2[i]=toupper(s[i]),i++);
  s2[i]='\0';
  if (i<=6) fprintf(stderr,"too small a multiplier, %s\n",s2);
  i--;
  a0=a1=0;
```

```
  for(j=0;j<6;j++)
    {
      if (s2[i-j]>='A' && s2[i-j]<='F')
val=((long)(s2[i-j] - 'A' + 10)) << (j*4);
      else val=((long)(s2[i-j]-'0')) << (j*4);
      a0+=(double) val;
    }
  for(j=0;j+6<=i;j++)
    {
      if (s2[i-j-6]>='A' && s2[i-j-6]<='F')
val=((long)(s2[i-j-6] - 'A' + 10)) << (j*4);
      else val=((long)(s2[i-j-6]-'0')) << (j*4);
      a1+=(double) val;
    }
}
```

B 'Good' 48-bit Linear Congruential Multipliers

Tables 5 through 12 show the values of multipliers for 48-bit linear congruential multipliers that pass our criterion and their associated values of μ_t.

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'80005AA7375'X	2.766	4.444	6.359	3.700	6.516
'8000F2461E5'X	2.726	4.100	6.671	4.142	6.290
'8000F61C16D'X	3.307	5.304	5.367	3.925	6.132
'80018086835'X	3.321	3.988	5.702	3.761	6.231
'80026A3DDA5'X	2.998	4.275	5.596	3.676	6.019
'80029F4E535'X	2.896	4.189	5.835	3.726	6.929
'8002FF6A7C5'X	3.026	4.798	5.295	3.552	6.003
'800368A5755'X	3.016	5.043	5.110	4.090	6.088
'8003899B4BD'X	2.789	4.086	4.975	3.990	7.481
'80038BEF745'X	3.112	4.272	5.689	3.707	6.333
'80044C387AD'X	2.856	4.278	4.997	3.512	6.137
'8004C8417F5'X	2.793	4.055	6.964	4.236	6.571
'800518D85BD'X	3.058	4.284	4.956	3.619	6.459
'80054A0FA35'X	2.999	4.374	6.205	4.269	7.105
'800589A7C8D'X	3.116	4.045	5.783	3.863	7.233
'8005D2F5E0D'X	3.341	4.031	6.114	4.143	7.029
'8007BB0AE75'X	2.931	4.594	5.179	3.675	7.108
'8008DFF948D'X	2.758	4.384	6.530	3.781	6.846
'800B1530415'X	3.048	5.096	6.415	3.549	6.879
'800B3653AED'X	3.165	5.009	5.358	3.773	6.861
'800BD10428D'X	3.080	4.229	6.097	4.894	7.419
'800BF21410D'X	3.007	4.527	5.158	3.568	6.022

Table 5: Good multipliers between '80000000005'X and '800E1E94085'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'F000A1BE415'X	3.023	4.421	5.525	4.787	7.147
'F000A3D391D'X	3.158	4.506	5.764	3.895	6.297
'F000D7DDBC5'X	2.750	4.145	5.541	3.592	9.015
'F00131032B5'X	3.236	4.042	5.802	4.667	6.087
'F0017950E45'X	3.245	4.596	5.143	3.555	6.208
'F0019C8E03D'X	2.832	4.327	5.278	4.833	6.586
'F002C5260F5'X	2.949	4.529	5.162	4.302	7.481
'F0046463E75'X	3.200	5.092	5.386	4.060	7.811
'F0053B6E65D'X	3.011	4.080	5.097	4.198	6.585
'F005550BA0D'X	2.852	4.105	6.526	3.824	6.189

Table 6: Good multipliers between 'F0000000005'X and 'F0057727985'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'1000005882BD'X	3.105	4.166	5.402	4.281	4.202
'100006C76B6D'X	2.761	4.135	5.937	3.957	10.119
'1000197C7C6D'X	2.949	5.053	5.075	3.907	6.605
'100021BD86BD'X	3.157	4.278	5.001	3.696	6.573
'100024F4DD6D'X	3.246	4.363	5.227	3.509	6.163
'10002D8A7FC5'X	2.802	4.371	5.101	3.932	9.722
'1000352F96DD'X	3.135	4.073	5.069	3.806	6.738
'1000415F8485'X	3.135	4.225	5.913	3.609	6.059
'100044FA312D'X	2.949	4.614	5.329	4.280	6.367

Table 7: Good multipliers between '100000000005'X and '100047A4F145'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'2000001F27C5'X	2.978	4.635	4.612	3.755	5.402
'2000007C9FE5'X	3.261	4.009	5.187	3.744	7.128
'20000A067A15'X	3.493	4.363	5.688	3.798	6.504
'20000AAA25A5'X	2.889	4.704	5.979	4.364	6.715
'20001A03BFF5'X	3.007	4.773	6.310	3.523	6.567
'20001F2DA35D'X	3.218	4.763	5.817	4.315	6.651

Table 8: Good multipliers between '200000000005'X and '20003161B945'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'2875B0EC5B2D'X	3.227	4.161	5.324	5.066	6.458
'2875BA91FC55'X	3.085	4.119	5.347	3.886	6.039

Table 9: Good multipliers between '2875A2ED2FB5'X and '2875BA9F39F5'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'40000053ED8D'X	3.129	4.160	5.102	4.354	8.054
'400002F7FE5D'X	2.834	4.933	5.009	4.287	7.362
'40000A6F9795'X	3.047	4.069	5.222	3.553	7.733
'40000DC5CD35'X	3.251	4.614	4.962	3.724	7.142
'400012144C5D'X	2.733	4.501	5.450	4.099	6.872
'400016AE84A5'X	2.856	4.117	5.040	4.275	6.120
'400019A72EAD'X	2.724	4.222	5.422	3.716	6.682
'40001E011115'X	2.873	4.042	5.533	3.887	6.547
'400020901F05'X	2.976	4.216	5.164	3.591	6.485
'400029794715'X	2.875	4.465	5.119	3.565	7.809
'400036C0E98D'X	2.813	4.248	5.252	4.236	8.558
'40003834E9F5'X	3.294	4.576	5.103	3.682	7.684
'40003999B535'X	2.920	4.434	5.216	3.768	6.668
'40003E87B36D'X	3.213	4.116	5.089	5.034	6.863
'40004BC0C50D'X	3.266	4.007	5.307	3.678	8.586
'40005355AD6D'X	2.908	4.003	5.077	3.738	6.660
'400054E5001D'X	2.768	4.237	5.251	3.768	7.029
'400059511E65'X	3.167	4.063	5.136	3.722	7.755
'40006769650D'X	2.804	5.025	6.213	3.729	7.389
'40007DF82875'X	3.332	4.470	6.273	4.255	6.870

Table 10: Good multipliers between '400000000005'X and '400083745808'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'800003378EA5'X	2.771	4.632	5.409	4.022	6.474
'800004817935'X	3.021	3.990	5.347	3.557	6.179
'80000547CA45'X	3.421	3.990	5.080	4.683	6.815
'8000187DD9DD'X	2.943	4.168	5.142	4.216	6.817
'80001B355195'X	3.509	4.473	5.295	4.146	6.148
'80001CABDFDD'X	3.006	4.234	5.599	4.481	7.103
'80001E0E5035'X	3.044	4.579	5.017	3.818	7.146
'8000221E6B6D'X	3.240	4.402	5.207	3.541	7.066
'80002233C195'X	3.100	4.040	6.333	3.957	6.335
'800027C75B7D'X	2.928	4.032	5.668	3.650	6.447
'80002E1E90C5'X	3.071	4.000	5.680	3.986	6.925
'80003520F66D'X	2.782	4.057	5.571	3.529	6.491
'80003D7C5675'X	2.759	4.329	5.485	4.763	7.119
'800050E41F9D'X	2.953	4.337	5.538	3.857	7.676
'800050EEE9A5'X	3.229	4.176	6.714	3.553	7.120
'80005A9ED725'X	2.931	4.028	5.288	3.823	6.651
'80006E1503ED'X	3.230	4.517	6.031	3.555	6.883
'800076A85005'X	3.076	4.477	5.014	4.790	6.612
'80007947425D'X	3.206	4.798	5.968	3.656	6.012
'800081FB1DDD'X	3.303	4.260	6.217	3.659	9.389

Table 11: Good multipliers between '800000000005'X and '8000856AC905'X

Multiplier	μ_2	μ_3	μ_4	μ_5	μ_6
'F0000199DDFD'X	2.905	4.941	5.390	3.684	8.007
'F0000A603375'X	2.986	4.470	6.671	3.810	7.827
'F00012669195'X	2.830	4.059	6.073	4.478	6.095
'F000178D7015'X	3.544	4.239	4.971	4.435	6.767
'F00019145F0D'X	3.142	4.419	5.275	3.721	6.364
'F0001C9994CD'X	2.900	4.672	5.757	4.256	6.154

Table 12: Good multipliers between 'F00000000005'X and 'F000304D99C5'X

References

[1] Barsotti, E. *et al.*, *Digital Triggers & Data Acquisition Using New Microplex & Data Compaction IC's*, Proceedings of the Workshop on High Sensitivity Beauty Physics at Fermilab (Nov. 11-14, 1987) J. Slaughter, N. Lockyer, M. Schmidt, editors, p. 369.

[2] Roberts, Lee A. **BCD/CPS** *An event-level GEANT3 parallelization via CPS* SSCL-413

[3] Bowden, M. *et al.*, *A High-Throughput Data Acquisition Architecture Based on Serial Interconnects*, Fermilab preprint (Nov. 1988).

[4] Gladney, L. D. *et al.*, *Proposal to SSC Laboratory for Research and Development for a Parallel Computing Farm* (Oct. 1989).

[5] Gladney, L. D. *et al.*, *Initial Experience with the Intel i860 Microprocessor*, U. Penn preprint UPR-0184E (March, 1990).

[6] R. Belusevic, G. Nixon, and D. Shaw, *An 80 Mbytes/sec Data Transfer and Processing System* RAL-90-028; R. Belusevic and G. Nixon, Nucl. Instr. and Meth. **A277**, 513 (1989).

[7] Hance, R. *et al.*, IEEE Trans. Nucl. Sci. **NS-34**, No.4. August, 1987.

[8] N. S. Lockyer, J. Wiener, E. Barsotti, M. Bowden, J. G. Morfin, S. Hansen, and C. Swoboda. *A Proposal for Generic Detector R&D for the SSC- Processor-Farms.*

[9] Rattner, J., in Proceedings of the Workshop on B Physics at the SSC (DeSoto, June 1989).

[10] Dally, William J., *A VLSI Architecture for Concurrent Data Structures*, (Kluwer, Hingham, MA, 1987).

[11] BCD Collaboration, *Bottom Collider Detector: A Low- and Intermediate-P_t Detector for the SSC*, SSC-240 (Sept. 30, 1989); *Bottom Collider Detector, Expression of Interest*, EOI008, submitted to the SSCL May 25, 1990.

[12] L. A. Roberts, *Monte Carlo Simulation of Silicon Vertex Detector for Bottom Collider Detector*, Fermilab preprint FN-488 (June, 1988)

[13] Bain, W. and Arshi, S. 1988. Hypersim: A Hypercube Simulator for Parallel Systems Performance Modeling. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications* (Pasadena, CA., Jan 19-20). ACM, Los Angeles.

[14] *Interwork II Concurrent Programming Toolkit Referencee Manual,* Block Island Technologies, Portland, OR, 1988.

[15] Knuth, D.E., *Seminumerical Algorithms, The Art of Computer Programming,* 1981, Addison-Wesley.

[16] James, F. Computer Physics Communications 60 (1990) 329

[17] Kubaska, Ted, "Parallel Random Number Generator", Internal Intel Document, 1990. (unpublished)

[18] Fox, Geoffrey C., Johnson, Mark A., Lyzenga, Gregory A., Otto, Steve W., Salmon, John. K., Walkesr, David W., *Solving Problems on Concurrent Processors*, 1988, Volume 1, Prentice Hall.

[19] Bowman, Kimiko O. and Robinson, Mark T., "Studies of Random Number Generators for Parallel Processing," *Hypercube Multiprocessors* (1987).

[20] S. Nugent *The iPSC/2© Direct-Connect™ Communications Technology,* Proceedings, 3rd Conference on Hypercube Concurrent Computers and Applications, 1988.