Modern Dataflow in Experimental Nuclear Science and Tcl

Ron Fox, Giordano Cerizza, Sean Liddick, Aaron Chester
National Superconducting Cyclotron Laboratory
Michigan State University

Abstract— **Advances in nuclear electronics can result in data rates several orders of magnitude higher than those of traditional, legacy electronic systems. Several technical developments are therefore required for both online and offline data handling. Included in those developments is a new Tcl MPI package and the retrofitting of NSCLSpecTcl to use that package to perform massively parallel histogramming.**

I. Introduction

This paper will introduce the types of science done at the National Superconducting Cyclotron Laboratory (NSCL). Analog data acquisition electronics will be described and contrasted with modern "digital" electronics. Modern electronics support experiments that we are not able to perform with legacy analog electronics. This capability comes at a cost. Digital electronics can result in significantly higher data rates than legacy analog electronics.

An upcoming experiment at the NSCL focused on decay spectroscopy will be instrumented with modern digital electronics and cannot be performed with traditional electronics. This experiment is anticipated to take data at continuous rates of up to 200MB/sec resulting in an aggregate data set of over 100TB of data. The planned online and "near-line" data flow of this experiment provides challenges (pronounced opportunities) to explore methods of handling data both online and "near-line". This experiment will be described, as well as the data flow and the challenges it presents.

Massively parallel computing will feature heavily in data for digital electronics. We have written libraries and frameworks to support parallel computation that can be easily switched between threaded parallelism and massively parallel cluster parallelism. We have also written an extended Tcl shell that supports massively parallel Tcl driven applications.

II. The NSCL and the Science we do

In this section we'll describe rare isotope experimental nuclear science performed at the NSCL.

This work is supported by the National Science Foundation grant PHY-1565546

A. *Nuclear Science with Rare Isotope Beams at NSCL*

The bulk of experimental nuclear science is done by colliding an accelerated beam of ions onto a target or, in the case of colliders, accelerated ions moving the opposite direction. For much of the history of experimental nuclear science, the accelerated particles have been stable isotopes that are common in nature.

Two techniques accelerate isotopes that are not stable; projectile fragmentation [MOR98], and isotope separation online (ISOL)[LIN04]. In projectile fragmentation, a stable beam strikes a *production target* conservation of momentum implies that the resulting reaction products will continue in the beam direction with most of the stable beam momentum. A reaction product separator then selects the desired isotope which is transported to the experiment. With ISOL, the reaction products from the production target are stopped in a thick target, chemically extracted and then re-accelerated. The NSCL produces its rare isotope beams via projectile fragmentation.

Rare isotope beams provide for several areas of scientific study that are not possible with stable beams. These include explorations of nuclear structure and shape far from stability. Furthermore, many of the isotopes we experiment with are believed to exist in the dense matter of supernova formed from stellar core collapse as well is in the crusts of neutron stars. These environments are where astrophysicists believe that elements heavier than iron are formed.

B. *The NSCL*

The NSCL is a running experimental facility on the campus of

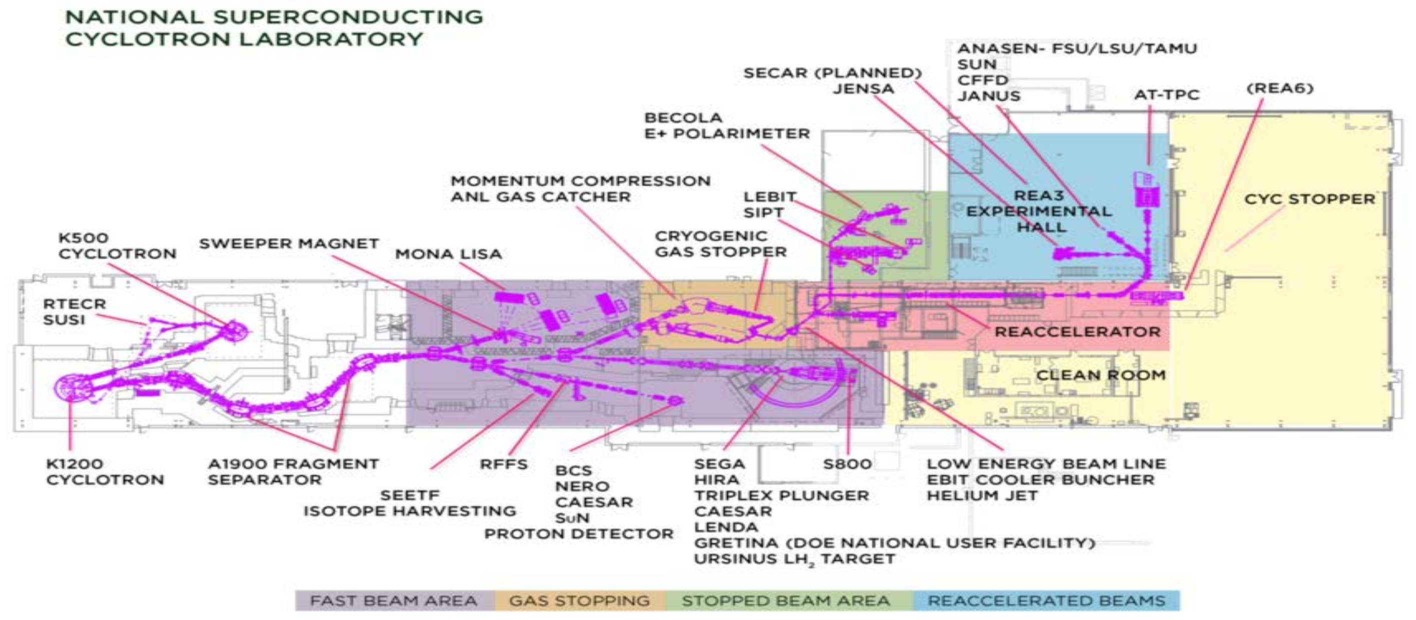

Figure 1 Facility layout for the National Superconducting Cyclotron Laboratory along with many of the available experimental devices. Colors indicate the energy of the rare isotope delivered to experimental systems in those areas.

Michigan State University. A schematic of the facility is shown in Figure 1. Funding for the operation of the NSCL is provided by the National Science Foundation.

The coupled cyclotrons at the left of Figure 1 produce stable primary beams of elements ranging from Oxygen to Uranium at energies from 80MeV/A to 170MeV/A. The ions from these stable beams strike a production target, typically Be, near the entrance of the A1900 fragment separator[Mor03]. The A1900 selects the desired rare isotope which is then transported to one of several expermiental areas shown to the right of the A1900 in Fig. 1.

Figure 2 shows a chart of the nuclides with the stable primary beams that have been produced and used at the NSCL. The black squares on that diagram represent stable isotope beams. The gray squares represent unstable beams of rare isotopes. Over 900 rare isotope beams have been used in experiments while a total of about 1000 have been produced or observed. Of these beams 47 have been stopped and used either in stopped beam experiments or reaccelerated low energy experiments.

III. Data Acquisition Then and now

This section will examine how data taking has evolved in nuclear science. We start by describing how data taking has been historically done. We then describe advances in electronics that we believe will push our data rates to as much as two orders of magnitude faster than they are at the NSCL.

The products of nuclear collisions are not directly

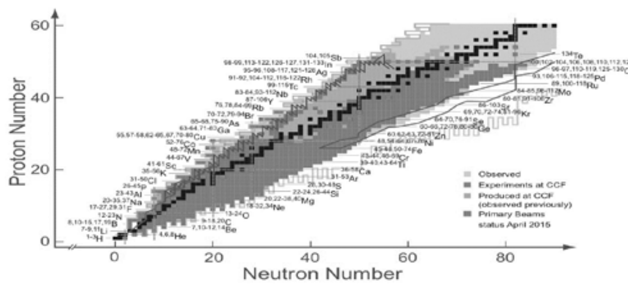

Figure 2 Chart of the nuclides showing beams produced at the NSCL to date.

observable. Instead we rely on the interaction of those products with bulk matter. For example, charged particles passing through detectors leave behind an ionization trail due to the electrons they knock out of atoms in the detector material. An electric field can then be used to cause those electrons to drift and be collected. The result of this collection is an electric pulse.

These pulses are quite weak and fast, perhaps on the order of nanoseconds (10^{-9} seconds) wide. In traditional "analog" systems, these pulses are amplified and shaped into pulses a few volts high and perhaps a microseconds across. In a parallel electronics path, discriminators are used to convert the analog pulses into logic pulses that are suitable as inputs to trigger logic and, with proper timing, as gates to peak sensing or charge integrating ADCs. Each of those ADCs produces one value per pulse and then, only if the pulse lies within the ADC gate. The electronics required to condition raw detector signals and time gates appropriately can be quite complex.

The advent of fast, high resolution flash ADCs (100+ MHz at 14 bits wide) and large FPGAs has made it possible to discard most of the analog electronics described above, replacing it with flash ADCs and logic and digital signal processing implemented in FPGAs. This results in a simplified signal chain for one channel that looks like Figure 3

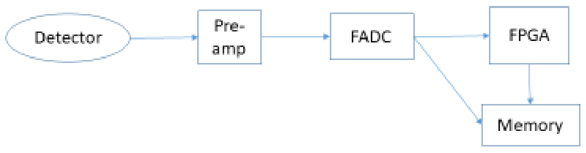

Figure 3Modern digital electronics chain

In this design, the FADC is continuously digitizing. When the FPGA determines a trigger condition has occurred, it can look back in time at the samples recorded to ensure the full signal pulse is captured. This ability to look back in time at a digitized waveform eliminates most, if not all signal timing delays. The FPGA can then perform the peak sensing and peak integration operations performed by early analog digitizers. If needed, the full signal trace is also available for further processing. These electronics chains are called digital electronics by experimental nuclear scientists because the FADC and FPGA replace what used to be complex chains of analog electronics.

It is the ability, and sometimes need, to capture signal traces that inflates the bandwidth and storage requirements of modern nuclear science experiments. Instead of providing one or two values for each event or each channel, each channel may provide 100 or more trace samples depending on the length of the capture interval and the frequency of the FADC. The ability to capture traces allows us to perform experiments that are not possible otherwise.

While a typical NSCL experiment lasts one week and captures between 10GB and 10TB of data, we anticipate that experiments with digital electronics that same week will capture between 1TB and 1PB of raw data. Furthermore, during an experiment it's important to process this data online and near-line (near-line processing refers to the analysis of data that was taken in an early stage of the experiment while the experiment is still running). This processing is used, not only to ensure the experimental systems are working but also to enable mid-run decision making.

IV. E17011 – Lifetime measurements in Neutron Rich Nuclei

E17011[Cri] is an experiment scheduled to run at the NSCL in early 2020. We'll first examine the science goals of this experiment and look at the experimental apparatus. We'll then examine the full online and near-line data processing chain and its performance requirements. Online processing refers to analysis done on the data as it passes through the data acquisition system. Near-line data processing refers to analysis done during the experiment on data files already acquired. Finally we will describe some of the methods we will use to meet the performance requirements of those demands. This part will segue into the next part which describes in detail mpitcl, which provides support for massively parallel computing for Tcl based applications.

A. *Experimental aims and setup*

Understanding the interplay between single-particle and collective effects is a continuing thrust in nuclear science to address questions articulated in the recent long range plan regarding the organization of subatomic matter and the emergence of simple patterns in complex nuclei. The relative energies between different single-particle orbits is a continuing question and the size of the energy gap located at a neutron number of N = 50 shell gap is of interest for its role in the r-process [Bar08] and nuclear existence [Erl12]. The energy gap can be probed by identifying nuclear states corresponding to the normal filling of single-particle levels and those that involved an excitation of neutrons across the N = 50 energy gap. The coexistence of such states at similar excitation energies has been observed in the 80Ge nucleus [Got16]. The two types of states to be studies both have a spin and parity of 0+ and can be populated through the beta decay of a neutron-rich 80Ga nucleus. The beta-decay process will turn a neutron in 80Ga into a proton and convert it to 80Ge while populating a range of excited nuclear states that will subsequently decay emitting photons or, potentially, electrons.

Rare isotope research facilities such as the NSCL can create these neutron rich nuclei and allow researchers to perform experiments with them.

E17011 examines the properties of 80Ge, a neutron rich isotope of Germanium. A beam of 80Ga(neutron-rich Gallium) will be brought to rest in a monolithic $CeBr_3$ (Cerium Bromide) scintillator detector read out with a 16 by 16 pixelated photomultiplier detector.

The β^- emitted by the decay of 80Ga will result in a pulse in the $CeBr_3$ and occasionally the beta-decay process will populated the excited 0_+^2 state. When the 0_2^+ state de-excites it will emit a second electron that will result in a pulse slightly delayed in time with respect to the beta-decay electron (see Figure). Measuring the time between these two pulses gives an exponential distribution from which the lifetime of the 0_2^+ state can be extracted as has been done previously [Cri16, Suc14]. This lifetime provides information about difference in the mean square charge radius between these two states in 80Ge.

Theory predicts that this lifetime is about 50ns. Since traditional analog digitizers have dead-times of microseconds, they cannot easily be used to perform this measurement. What we can do, however, is capture signal traces from the $CeBr_3$

detector using modern digitizers of the sort shown in Figure 3.

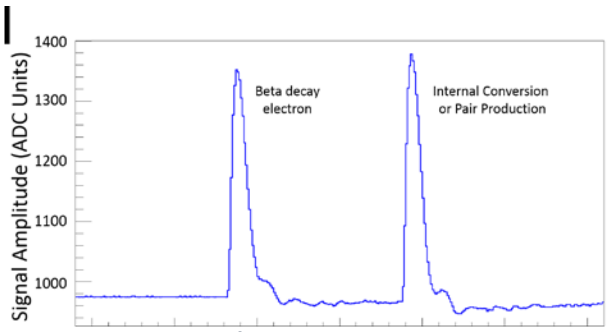

Figure 4 Detector trace from a similar experiment showing the beta-decay electron between 68Co populating the excited 0_2^+ state in 68Ni which decays a short time later through the emission of an internal conversion electron or internal pair formation [Cri16].

The originally proposed experimental setup is shown schematically in Figure 5. The beam goes from left to right in this figure. The CeBr$_3$ detector is at the center of 16 detectors of the Segmented Germanium (SeGa) array [Mue01] shown as the blue cylinders. SeGA, and an array of 16 LaBr$_3$ [Lon19] detectors are used to directly measure the gamma-ray transition energies.

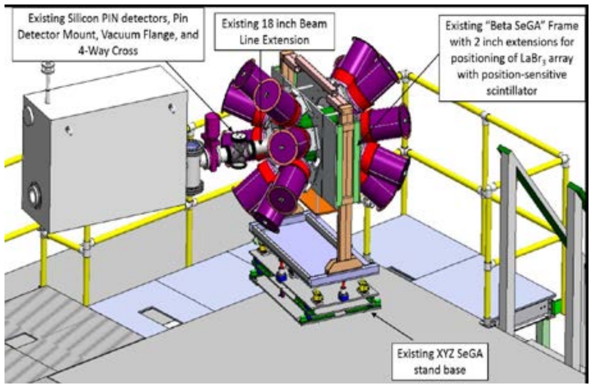
Existing Silicon PIN detectors, Pin Detector Mount, Vacuum Flange, and 4-Way Cross
Existing 18 inch Beam Line Extension
Existing "Beta SeGA" Frame with 2 inch extensions for positioning of LaBr$_3$ array with position-sensitive scintillator
Existing XYZ SeGA stand base

Figure 5 E17011 experimental setup

Upstream of SeGA a stack of PIN diode detectors that do particle-by-particle identification of the beam particles being deposited in the CeBr$_3$ detector based on energy loss and time-of-flight measurements. This particle identification allows us to know exactly what is being implanted into the CsBr$_3$ detector since the beam delivered to the final setup will contain a range of isotopic species

B. *The online and near-line analysis chains.*

In order to understand the data from the experiment sufficiently to know if the experiment is working properly we need to perform the following analysis in a combination of online and near-line processes.

- Hits from the digitizers must be combined in time coincidence into events varying in size depending on experimental need between a few hundred nanoseconds and a few microseconds. This process is known as event building. Note that the digitizers we use emit data we call hits that is timestamped but these hits are not guaranteed to be time ordered
- The data flow can be reduced by requiring that PIN detector hits have a minimum threshold energy indicating the particle was a heavy ion. This process is known as software triggering. We do not know in advance the fraction of data software triggering will retain.
- The traces from the CeBr$_3$ detectors must be analyzed to determine if they are single or double pulses (the bulk will be single pulses). For the double pulses, the time between pulses and the heights of both pulses must be determined.
- Implantation events must be correlated with the corresponding decays.
- Live data must be recorded to disk for near-line and offline analysis. The files recorded contain time sequenced data for each event allowing a complete reconstruction of the experiment in offline analysis.

Figure 6 Shows our proposed online data flow. While the trigger rates are expected to be a relatively modest 3 KHz, the data flow is expected to be about 200MB/sec. This is due to traces we will take and the high expected channel multiplicities. Note that the NSCLDAQ data flow scheme allows us to treat each of the data flow arrows in Figure 6 as a test point with no impact on data flow. Note that the pulses shown in Figure 4 can overlap heavily.

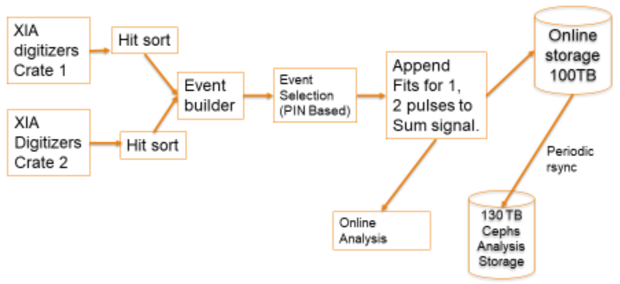

Figure 6: Data flow for E17011

C. *Meeting rate requirements*

The first bottleneck in Figure 6 is the process labeled ("Append fits for 1, 2 pulses to the sum signal"). In addition to signals for each pixel, digitized at 250MHz, the detector provides a signal summing the energies in all pixels which we will digitize at 500MHz. Online, our goal is only to fit the energy sum signal. The pixel signals will be fitted in near-line analysis.

Fitting the traces is used both to extract the key features of those pulses (time and peak height) as well as to provide determine if the traces contain a single or double pulse. The following empirical functional forms are used to fit the pulses:

$$y = C + \frac{Ae^{-k1(x-x0)}}{1+e^{-k2(z-x0)}} \tag{2}$$

$$y = C + \frac{A_1 e^{-k1(x-x0)}}{1+ e^{-k2(z-x0)}} + \frac{A_2 e^{-k3(x-x1)}}{1+ e^{-k4(x-x1)}}$$

The first of these functions represents a single pulse and the second a double pulse. The trace has a DC offset C. For each pulse, the denominator is a logistic function that is used to model the rising edge of the pulse, the numerator contains a scale factor related to the pulse height and an exponential decay that models the falling edge of the pulse. The values of importance are A, A1, A2, x0 and x1. For a single pulse the actual pulse amplitude is gotten by evaluating the pulse formula at:

$$x_{max} = x_0 + \ln\left(\frac{k2}{k1}\right)/k2 \qquad (3)$$

Fitting the sum trace requires in excess of 3ms/trace using the Levenberg-Marquardt[Lev] fit routines in the GNU Scientific Library. The ratios of the chi square errors of the fit between the single and double pulse provide a heuristic for determining if the pulses are double or single.

The next bottleneck is our ability to extract parameters and implantation/decay correlations from the data so that histograms can be generated and visualized. The actual histogram operations are not a bottleneck. It is clear that to achieve our online analysis goals we'll need to make use of parallel processing.

1) Parallel libraries and programs – classifying and fitting

In order to meet the performance requirements of online analysis while allowing us to re-use code for near-line analysis, we have written libraries to support parallel processing and framework programs that hide the nature of parallel processing from the user. While these are targeted at E17011 they are, in fact quite general.

The libraries themselves abstract communication into communication patterns (e.g. fanout, fanin, and pipeline) and separate these patterns from the underlying transport. At present we support 0MQ[Zmq] and the Message Passing Interface[Mpi] (MPI) as transports. 0MQ and its high performance *inproc* transport targets threaded parallelism while MPI targets process parallelism on massively parallel cluster computing. Note that nothing in our libraries precludes the use of 0MQ in distributed parallel computing.

Online analysis will use threaded parallelism, while near-line and offline will use MPI parallelism. The libraries allow single programs to be written that can select their parallelization strategy (threaded/0MQ or cluster/MPI) at run-time. We have written several programs that provide the capability to plug-in user, application specific code that is unaware of the parallel nature of the underlying program. These programs can be told on the command line whether or not to use threaded or MPI parallelism. These framework programs accept user code built into a shared object library which is loaded at run-time.

Two programs we use in the online data flow are:

- A classifier: User code assigns a uint32_t value for each event. This is the first stage of event selection. The second stage is a generic selection of events whose classifications meet specific criteria. Classification can be arbitrarily complex while selection is usually quite simple.
- An event editor. User code is given an event fragment and generates a replacement for that fragment. This is used online to append fits information to hits and offline to strip traces from event fragments that have already been fit.

Since for all of our programs, events can be processed independently, the framework programs all have the same general structure shown in Figure 7

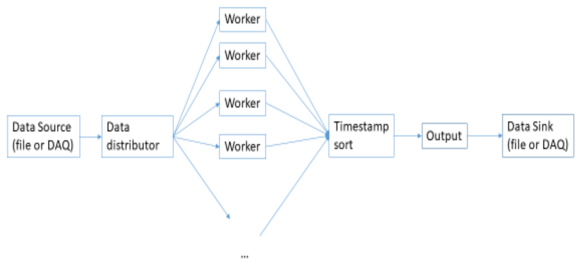

Figure 7: Structure of event parallel programs.

The programs are parallel pipelines with data flowing from left to right. The data distributor, fans out blocks of events to workers that operate on them in parallel. The workers fan in blocks of modified events to a thread/process that re-sorts the data by timestamp and then passes them on to a thread/process that outputs them.

- In most cases each event can be operated on independently of all other events allowing the workers to be embarrassingly parallel.
- In order to efficiently balance messaging and processing, the programs have a command option to let the user select the number of events in each block of events sent to workers.
- A pull protocol is used by the workers to get data from the distributor.
- To allow the correlation of implantations with corresponding decays, we re-sort the data back into its original order before outputting it.

We used the event editor with user code that fits the energy sum signal to check the online performance of the fitting part of the dataflow. The program was run on a system with 40 cores of Xeon 6148 clocked at 2.4Ghz. This is a system we intend to use for the experiment. Figure 8 shows the event processing rate as a function of the number of worker threads:

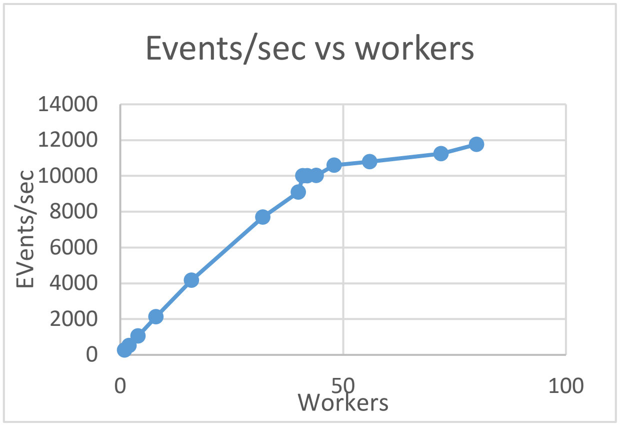

Figure 8: Threaded fit performance

This figure shows that:

- We meet the online rate requirements using between 4 and 6 worker threads.
- There is a knee when we hit the number of actual processor cores. Additional performance gains due to hyper threading don't add nearly as much as additional cores do.

Near-line, we'd like to fit all 256 pixels at least at the incoming data rate. Due to the monolithic nature of the central CeBr3 detector, most pixels are expected to trigger for each event. This will require about 2500 cores. The event editing program fitting can run either threaded parallel or MPI parallel on a massively parallel cluster. The NSCL has a modest cluster of around 200 cores called Fireside.

Michigan State University, has a unit called the Institute for Cyber Enabled Research[Icer] (ICER). Among other things, ICER provides a cluster of over 23,000 cores available for use by MSU researchers. We hope to make use of this facility for near-line analysis during the run.

Figure 9 shows the performance of the same fit program using cluster parallel computing both on Fireside and at ICER.

Several points to note:

- The ICER cores are faster than the NSCL cores.
- Both plots have performance knees. This is because the cluster nodes are heterogeneous. As the number of cores required increases we get relegated to the more plentiful, older cores.

Our performance measurements at ICER duplicated the NSCL software environment using Singularity[Sylabs] containers. Singularity is MPI aware. Unfortunately the version of singularity available at ICER has a bug that makes running MPI applications with more than 64 processes unreliable. ICER is in the process of updating their singularity runtime to allow us to extend our scaling measurements.

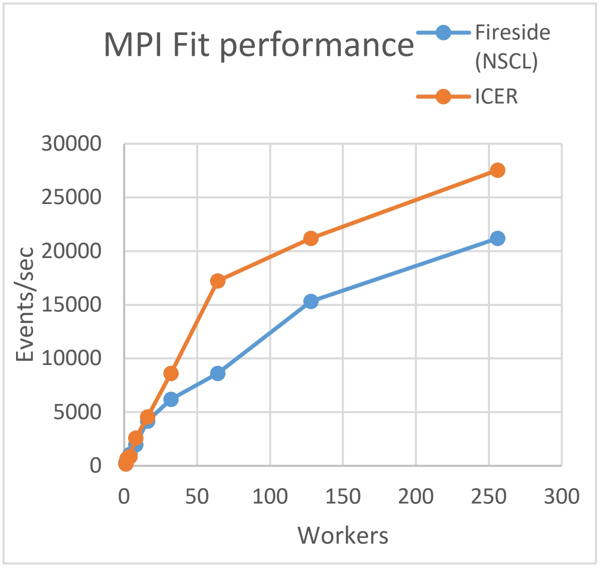

Figure 9MPI fit performance

2) *Parallel online histograming – NSCLSpecTcl.*

At NSCL, our primary histograming and visualization tool for online data analysis is NSCLSpecTcl[Fox04]. This is a Tcl driven program. The experimenter supplies a logical pipeline of code that accepts raw events as input and produces parameters, both raw and synthetic as output. A generic histograming kernel increments histograms. NSCLSpecTcl specific commands allow users to dynamically create histograms, create conditions and apply arbitrary logical combinations of those conditions to control which events cause histograms to be incremented. Tk allows users to create sophisticated application specific GUIs.

The structure of NSCLSpecTcl is shown in Figure 10.

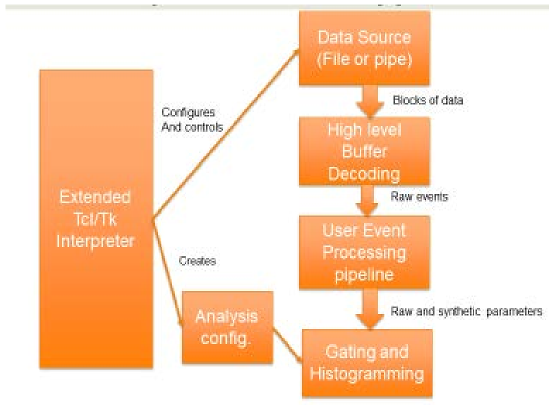
Data Source (File or pipe)
Blocks of data
High level Buffer Decoding
Raw events
User Event Processing pipeline
Raw and synthetic parameters
Gating and Histogramming
Extended Tcl/Tk Interpreter
Configures And controls
Creates
Analysis config.

Figure 10 NSCL SpecTcl block diagram

Geordano Cerizza has forked the NSCLSpecTcl codebase and created a threaded version of NSCLSpecTcl that completely retains its interactive nature. His work allocates one thread to the data source and buffer decoding. User event then become a set of parallel workers. Each worker gets a block of events and, when parameters for that block of events have been extracted, the resulting lists of event parameters are sent to the main thread via Tcl_ThreadQueueEvent.

One problem with this and other event parallel schemes is that for these implantation/decay experiments, events are not completely independent of each other. At some point in the analysis, implantation events must be correlated with their associated decays. In this and other fully event parallel programs, implantations may have their decays in different blocks of events, processed by other workers.

For E17011, where the 80Ga half-life is a bit less than 1.7 seconds, we can make the blocks of data sufficiently large (an 800Mbyte data block represents about 4 seconds of data) to allow a sufficient number of implantation and decay correlations to be made to understand the experiment online and near-line.

Figure 11 shows the performance of threaded SpecTcl on a simple analysis. This analysis just unpacks the raw parameters from the fragments of each built event. Note that the actual online analysis will be quite a bit more complex.

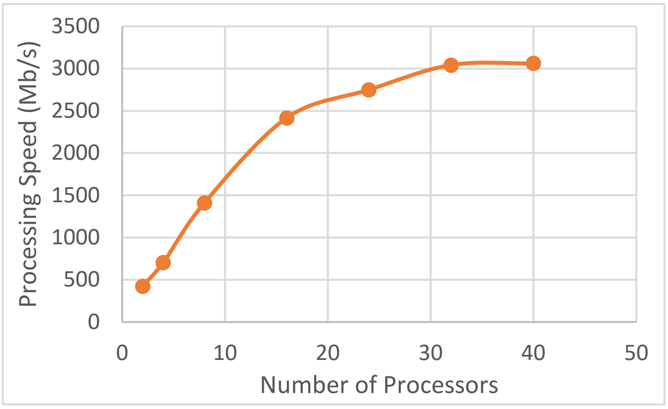

Figure 11Threaded SpecTcl performance

The roll-off after about 16 processors at about 3.5-3Gbytes/Sec represents data transfer limits from the local SSD on which the event data were stored.

V. MPITCL AND MASSIVELY PARALLEL TCL PROGRAMMING

This section introduces an MPI aware Tcl interpreter. The Message Passing Interface (MPI) is introduced. We'll describe existing work to make MPI accessible to Tcl. We'll then describe the features of mpitcl and provide simple examples of mpitcl scripts. Finally we'll present a real application of mpitcl to an existing program, NSCLSpecTcl allowing it to run in a cluster parallel environment. MPI consists of a specification and language bindings that make the functions and data types described by that specification available to specific programming languages.

A. Message Passing Interface

The message passing interface, or MPI is a distributed parallel programming system. MPI programs run the same process in parallel on multiple nodes or cores within those nodes. MPI is supported on virtually all massively parallel cluster computing environments. OpenMPI and MPICH are the two common implementations of the MPI specification. Both are open source implementations of the MPI standard. Our tests all use the OpenMPI libraries.

In MPI processes exchange messages using an opaque structure called a "communicator" When the MPI program is run a single communicator: MPI_COMM_WORLD is defined. Subsequently the program can define additional communicators and use these communicators to create process groups that communicate internally.

Within a communicator a process has a *rank.* A process's rank is just an integer number that runs from 0 to n-1 where n is the number of processes in that communicator.

Sending a message requires:

- A communicator.
- The rank of the receiving process within the communicator.
- A message tag that can be used to identify the type of message being sent.
- A buffer.
- A data type indicating the type of data in the block
- The number of data items in the block.

When receiving messages, processes can either specify exactly the tag and rank from which to receive a message, allowing for complex application protocols, or provide wild cards for either the tag or the rank (note that even with a wild card source rank messages receipt will be restricted to those processes within the communicator specified in the receive call.

MPI defines primitive data types and allows users to define their own data types.

1) Existing MPI Tcl packages.

The MPI Specification is large, rich and complex. Version 3 of the specification defines approximately 600 API functions. The size and complexity of MPI that requires some choices when exposing the MPI to Tcl applications.

Axel Kohlmeyer created a Tcl MPI package[Kohl]. He decided to encapsulate a subset of the MPI specification. His work is somewhat SWIG providing Tcl verbs that represent bindings to raw MPI calls.

Kohlmeyer's tclmpi package provides Tcl bindings to 21 MPI API functions. Tclmpi also provides a tclmpi namespace with values for the predefined communicators as well as MPI data types and a Tcl specific data type.

While the subset of MPI implemented by tclmpi reduces somewhat the complexity of of the MPI specification, raw MPI is still exposed to script writers.

2) mpitcl – NSCL style.

In designing the specification for mpitcl, an MPI aware shell, we wanted to hide MPI as much as possible from the script author. We thought a bit about what Tcl-ish MPI encapsulation might allow script to do.

We came up with the following requirements:

- Tcl processes must be able to send each other scripts to execute. These scripts would be executed asynchronously. Sending these scripts is the primary mechanism that will be used to initiate computations in the application.
- Tcl processes must be able to send each other Tcl data. The receiving process must be able to establish a Tcl script to handle the receipt of data.
- Applications must be able to determine the number of processes in the application and each process should be able to determine its rank within the application.
- Compiled code must be able to take over the messaging completely. Most Tcl MPI applications will actually be Tcl driven with compiled cores.

Our mpitcl interpreter provides an **mpi** namespace which, in the current version defines an **mpi** command ensemble. The subcommands of this ensemble are:

- **size** - returns the number of processes in the application.
- **rank** – returns the rank of the process in the MPI_COMM_WORLD communicator.
- **execute *rank script* -** executes the script in the rank specified by *rank* two special values for *rank* are **all** and **others.** These execute the script in all processes (including the sender) and in all other processes respectively.
- **send *rank data* –** sends *data* to *rank* the special values **all** and **others** are supported with the same meaning as the **execute** subcommand.
- **handle *script*** – invokes *script* the process receives data. *script* is invoked with two parameters, the sender's rank and the data that was received.
- **stopnotifier** – Only legal in rank 0 – stops the event loop message notification thread.
- **startnotifier –** Only legal in rank 0 – starts the event loop message notification thread (starts by default).

In mpitcl,

Only rank 0 runs an interpreter main loop. All other ranks will run a loop probing for messages and handling them. Tags are used to distinguish between message types. In this implementation there is only support for the MPI_COMM_WORLD communicator and therefore no support for the creation of process groups.

When rank 0 initializes, it starts a thread that probes for MPI messages. When a message is received, Tcl_ThreadQueueEvent is used to notify the interpreter's event loop a message is available for processing. The event handler reads and processes the MPI messages in exactly the same manner as non-rank 0 threads. Once the main thread processes this event, the notification thread is restarted. A special MPI tag tells the notification thread to stop without notifying the interpreter event loop.

Thus we have a model where the interpreter in rank 0 is a master and all other ranks are slaves. Initially rank 0 will send scripts and data to other ranks. Those scripts may direct non rank 0 processes to communicate with each other in order to collaborate on the required computation.

Note that the **execute** and **send** commands don't use message passing when the target is the invoking process. Instead we just directly perform the operation. EXAMPLE 1 shows a minimal mpitcl script. This application just exits after starting.

```
mpi::mpi stopnotifier
mpi::mpi execute all exit
```

Example 1A mimimal mpitcl script

The script shown in EXAMPLE 2 is a bit more complex. It shows how mpitcl scripts gather data sent to them from other nodes. This script demonstrates a pattern commonly used by the rank 0 process to get the contributions to the result of a computation.

```
set slaves [mpi::mpi size]
incr slaves -1;          # number of slave processes.


proc receiver {rank data} {
   puts "Received from $rank '$data'"
   incr ::slaves -1
}

mpi::mpi handle receiver

mpi::mpi execute others {
   mpi::mpi send 0 "Rank [mpi::mpi rank] is alive"
}

while {$slaves} {
   vwait slaves

}

mpi::mpi stopnotifier
```

Example 2 Requesting and receiving data.

Each process is told to execute a script that sends a sign on message back to rank 0. The rank 0 then enters the event loop while there are still slaves that have not responded. As data are received, the **receiver** proc is invoked from the event loop. It handles the data and terminates the **vwait** command. The program then stops the notifier thread and tells al processes to exit.

B. Massively parallel NSCLSpecTcl

This section describes what was necessary to take the existing interactive NSCLSpecTcl application and port it to run under the mpitcl MPI aware tcl shell.

Our first task was to create a batch version of NSCLSpecTcl. We replaced the vertical boxes in Figure 10 NSCL SpecTcl block diagram with an abstract data source and an abstract data sink. We initially implemented a file data source and a data sink that contained the analysis pipeline and histogramming. We added the following commands:

- **filesource** sets the data source to be the file data source connected to a specific data file.
- **analysissink** specified the data sink to be the the analysis pipeline and histograming engine.
- **analyze** – takes chunks of events from the data source passing them to the data sink until the data source is exhausted.

This batch version of SpecTcl was turned into a Tcl loaded package called **spectcl**. The user analysis pipeline invoked by the analysis sink was isolated into a shared library the user created from a skeleton and loaded as a Tcl package. The pipeline elements do not require any code changes from interactive NSCLSpecTcl. Note that the analysis configuration is a Tcl script that can be created with interactive SpecTcl.

Batch then was made to run under mpitcl. This was done by creating another package called **mpispectcl**. This package added the following commands:

- **mpisource** specifies the data source to be blocks of events received via MPI messages.
- **mpisink** specifies that the data sink was a distributor using MPI messaging to send data to processes on request from MPI data sources.

In MPI Spectcl each rank has a complete copy of NSCLSpecTcl. A run of MPI NSCLSpecTcl is involves the rank 0 process:

1. Ensuring all required packages are loaded in the all ranks.
2. Ensuring the analysis configuration script is loaded into all ranks.
3. Configuring all non-rank 0 processes to use an mpi data source and an analysis data sink.
4. Configuring rank 0 to use a file data source and an mpi data sink
5. Analyzing the file.
6. Collecting, summing and writing the histograms produced by all non-rank 0 processes. These histogram files can then be visualized using interactive Spectcl.

Example 3 shows how this is done with mpitcl:

```
mpi::mpi execute all {
   package require spectcl
   package require mpispectcl
   package require MyPipeline
   source defs.tcl
}
mpi::mpi execute others {
   mpisource
   analysissink
}
filesource run-0003-00.evt
mpisink
mpi::mpi stopnotifier
mpi::mpi execute others analyze
analyze

mpi::mpi startnotifier

writeSpectra

mpi::mpi stopnotifier;
mpi::mpi execute others exit
exit
```

Example 3 MPI SpecTcl anlayzing a data file.

The proc **writeSpectra** is not shown. It uses the technique demonstrated in Example 2 to request all non-rank0 processes send the histograms they have computed. Histogram contributions from each rank are summed and then written to file.

NSCLSpecTcl's MPI scaling is shown in Figure 12 NSCL SpecTcl MPI scaling. We reached I/O bandwidth limits after 8 workers (9 processes). The analysis was the same simplistic event unpacking software used in Figure 11: Threaded SpecTcl performance. During E17011, the analysis pipeline will be considerably more complex.

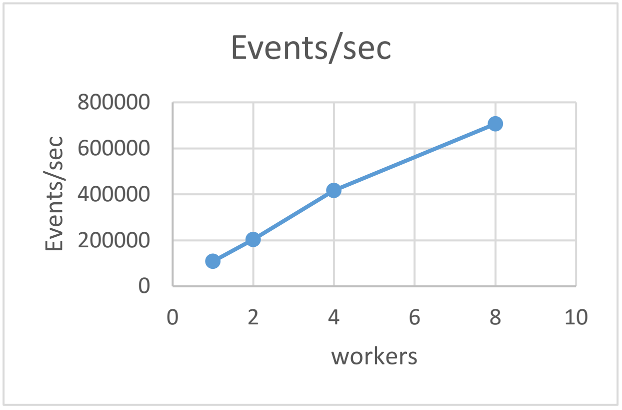

Figure 12: NSCL SpecTcl MPI scaling

VI. CONCLUSIONS

Modern data acquisition electronics have the potential to increase the data flow requirements of experiments in nuclear science. Experiment E17011 is an example of this trend.

In order to meet the demands, of that, and other experiments, we have developed parallel libraries and programs that make it easy for people not experience in developing parallel software to make use of either threaded or cluster parallelism. We will be applying these techniques to E17011 specifically and adapting them to even more complex experiments in the future.

REFERENCES

[MOR98] *Radioactive Nuclear Beam Facilities Based on Projectile Fragmentation*, D.J. Morrissey and B.M. Sherrill, Proc. Royal Soc. **A** 356 (1998) 1985

[LIN04] *Review of ISOL-Type Radioactive Beam Facilities* M. Lindroos Proceedings of EPAC 2004 online at https://accelconf.web.cern.ch/accelconf/e04/PAPERS/TUXCH01.PDF

[Cri] Experimental proposal for E17011, B. Crider et al. unpublished.

[Lev] *A Method for the Solution of Certain Non-Linear Problems in Least Squares* K. Levenberg Quarterly of Applied Mathematics 2(2) 164-168

[Zmq] http://zeromq.org

[Mpi] https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[Icer] https://icer.msu.edu/

[Sylabs] https://sylabs.io/singularity/

[Fox04] *NSCLSpecTcl Meeting the needs of Preliminary Nuclear Physics Data Analysis* R. Fox et al. Tcl 2004 New Orleans available online at https://www.tcl.tk/community/tcl2004/Papers/RonFox/fox.pdf

[Kohl] https://sites.google.com/site/akohlmey/software/tclmpi

[Cri16] B.P. Crider, et al., Phys. Lett. B 763, 108-113 (2016).
[Suc14] S. Suchyta et al.,Phys Rev C 89, 067303 (2014).
[Mue01] W. F. Mueller, et al., Nucl. Instrum. Methods Phys. Res. A 466, 492 (2001).
[Mor03] D. J. Morrissey, B. M. Sherrill, M. Steiner, A. Stolz, and I.Wiedenhoever, Nucl. Instrum. Methods Phys. Res. B 204, 90 (2003).
[Bar08] S. Baruah et al., Phys. Rev. Lett. 101, 262501 (2008).
[Got16] A. Gottardo et al., Phys. Rev. Lett. 116, 182501 (2016).