

# SAND REPORT

SAND2003-1898

Unlimited Release

August 2003, last updated September 2007

## Trilinos Developers Guide<sup>a</sup>

Michael A. Heroux, James M. Willenbring and Robert Heaphy

Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, NM 87185-1110

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,  
a Lockheed Martin Company, for the United States Department of  
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

---

<sup>a</sup>For Trilinos Release 8.0

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

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: reports@adonis.osti.gov  
Online ordering: <http://www.doe.gov/bridge>

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

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: orders@ntis.fedworld.gov  
Online ordering: <http://www.ntis.gov/ordering.htm>



# Trilinos Developers Guide<sup>†</sup>

Michael A. Heroux James M. Willenbring Robert Heaphy  
Scalable Algorithms Department

Sandia National Laboratories  
P.O. Box 5800  
Albuquerque, NM 87185-1110

## Abstract

The Trilinos Project is an effort to facilitate the design, development, integration and ongoing support of mathematical software libraries. A new software capability is introduced into Trilinos as a *package*. A Trilinos package is an integral unit usually developed by a small team of experts in a particular algorithms area such as algebraic preconditioners, nonlinear solvers, etc.

The Trilinos Developers Guide is a resource for new and existing Trilinos package developers. Topics covered include how to configure and build Trilinos, what is required to integrate an existing package into Trilinos and examples of how those requirements can be met, as well as what tools and services are available to Trilinos packages. Also discussed are some common practices that are followed by many Trilinos package developers.

---

<sup>†</sup>For Trilinos Release 8.0

# Acknowledgments

The authors would like to acknowledge the support of the ASCI and LDRD programs that funded development of Trilinos and recognize all of our fellow Trilinos contributors: Chris Baker, Teri Barth, Ross Bartlett, Paul Boggs, Erik Boman, Todd Coffey, Jason Cross, David Day, Clark Dohrmann, David Gay, Michael Gee, Ulrich Hetmaniuk, Robert Hoekstra, Russell Hooper, Vicki Howle, Jonathan Hu, Tammy Kolda, Kris Kampshoff, Sarah Knepper, Joe Kotulski, Richard Lehoucq, Kevin Long, Joe Outzen, Roger Pawlowski, Mike Phenow, Eric Phipps, Marzio Sala, Andrew Rothfuss, Andrew Salinger, Paul Sery, Paul Sexton, Chris Siefert, Ken Stanley, Heidi Thornquist, Ray Tuminaro and Alan Williams.

# Contents

<b>1</b>	<b>Introduction</b> .....	<b>10</b>
1.1	How To Use This Guide .....	11
1.2	Typographical Conventions .....	12
<b>2</b>	<b>Getting Started</b> .....	<b>12</b>
2.1	Obtaining a Copy of Trilinos .....	12
2.2	Recommended Build Directory Structure .....	13
2.3	Configuring Trilinos .....	16
2.4	Trilinos Configuration Options .....	19
2.5	Building and Installing Trilinos .....	22
2.6	Configure, Make and Install: Common Problems .....	23
2.7	Tips for Making the Configure, Make, and Install Processes More Efficient .....	24
2.8	Adding and Removing Source Files .....	26
<b>3</b>	<b>Services Available to Trilinos Packages</b> .....	<b>30</b>
3.1	Configuration Management .....	30
3.2	Regression Testing .....	31
3.3	Test Harness .....	31
3.4	CVS Repository .....	31
3.5	Bonsai .....	32
3.6	Bugzilla .....	32
3.7	Mailman .....	35
3.8	Package Website Template .....	36
3.9	Portable Interface to BLAS and LAPACK .....	36
<b>4</b>	<b>Trilinos Package Requirements</b> .....	<b>36</b>
4.1	Add Package to Trilinos Repository .....	38
4.2	Port Package to All Supported Platforms .....	38
4.3	Respond to All Relevant Configure Options .....	38
4.4	Respond to Issue Reports in a Timely Manner .....	39
<b>5</b>	<b>Suggested Software Engineering Practices</b> .....	<b>39</b>
5.1	Preliminary Steps .....	39
5.2	Practices to Support the Software Lifecycle .....	40
5.3	Requirements .....	41
5.4	Specification/Design .....	41
5.5	Implementation .....	42
5.6	Integration .....	42
5.7	Maintenance .....	42
5.8	Retirement/Replacement .....	43

<b>6</b>	<b>Petra, Thyra, and Teuchos: Special Trilinos Libraries</b> .....	<b>44</b>
6.1	Petra .....	44
6.2	Thyra .....	45
6.3	Teuchos .....	45
<b>7</b>	<b>Integrating a Package into Trilinos</b> .....	<b>46</b>
7.1	Improving Portability .....	49
<b>8</b>	<b>Interoperability Status for Existing Trilinos Packages</b> .....	<b>49</b>
	<b>References</b> .....	<b>52</b>

## Appendix

<b>A</b>	<b>Commonly Used CVS Commands</b> .....	<b>53</b>
<b>B</b>	<b>The Trilinos Release Process</b> .....	<b>57</b>
<b>C</b>	<b>Creating a New Trilinos (Release) Branch with CVS</b> .....	<b>59</b>

## Figures

1	Recommended Layout for Trilinos Build Directories .....	15
---	---	----

## Tables

1	Typographical Conventions for This Document. ....	12
2	Trilinos Package Requirements and Suggested Practices. ....	37
3	Useful Terminology for Section 7. ....	46

- Trilinos** The name of the project. Also a Greek term which, loosely translated means “a string of pearls,” meant to evoke an image that each Trilinos package is a pearl in its own right, but is even more valuable when combined with other packages.
- Package** A self-contained collection of software in Trilinos focused on one primary class of numerical methods. Also a fundamental, integral unit in the Trilinos framework.
- Didasko** The tutorial of Trilinos. Offers a quick introduction to several Trilinos packages. It contains a printable PDF document and a variety of well-commented browsable examples that illustrate how to use Trilinos.
- Didasko contains examples for the following Trilinos packages: Teuchos, Epetra, EpetraExt, TriUtils, Galeri, AztecOO, IFPACK, ML, NOX, Anasazi, TPe-  
tra.
- new\_package** A sample Trilinos package containing all of the infrastructure to install a new package into the Trilinos framework. Contains the basic directory structure, a collection of sample configuration and build files and a sample “Hello World” package. Also a website.
- Amesos** The Direct Sparse Solver Package in Trilinos. The goal of Amesos is to make  $AX=B$  as easy as it sounds, at least for direct methods. Amesos provides clean and consistent interfaces to several popular third party libraries.
- Anasazi** An extensible and interoperable framework for large-scale eigenvalue algorithms. The motivation for this framework is to provide a generic interface to a collection of algorithms for solving large-scale eigenvalue problems.
- AztecOO** Linear solver package based on preconditioned Krylov methods. A follow-on to the Aztec solver package [18]. Supports all Aztec interfaces and functionality, but also provides significant new functionality.
- Belos** A Greek term meaning “arrow.” Belos is the next generation of iterative solvers. Belos solvers are written using “generic” programming techniques. In other words, Belos is written using TSF abstract interfaces and therefore has no explicit dependence on any concrete linear algebra library. Instead, Belos solvers can be used with any concrete linear algebra library that implements the TSF abstract interfaces.
- Claps** Claps is a collection of domain decomposition preconditioners and solvers.
- Epetra** See description under “Petra”.

- EpetraExt** A set of extensions to Epetra. To allow Epetra to remain focused on its primary functionality as a Linear Algebra object support, EpetraExt was created to maintain additional support for such capabilities as transformations (permutations, sub-block views, etc.), coloring support, partitioning (Zoltan), and I/O.
- Galeri** Contains a suite of utilities and classes to generate a variety of (distributed) linear systems. Galeri's functionalities are very close to that of the MATLAB's gallery() function.
- Ifpack** Object-oriented algebraic preconditioner, compatible with Epetra and AztecOO. Supports construction and use of parallel distributed memory preconditioners such as overlapping Schwarz domain decomposition, Jacobi scaling and local Gauss-Seidel relaxations.
- Isorropia** A repartitioning/rebalancing package intended to assist with redistributing objects such as matrices and matrix-graphs in a parallel execution setting, to allow for more efficient computations. Isorropia is primarily an interface to the Zoltan library, but can be built and used with minimal capability without Zoltan.
- Jpetra** See description under "Petra".
- Kokkos** A collection of the handful of sparse and dense kernels that determine much of the performance for preconditioned Krylov methods. In particular, it contains function class for sparse matrix vector multiplication and triangular solves, and also for dense kernels that are not part of the standard BLAS. Kokkos is not intended as a user package, but to be incorporated into other packages that need high performance kernels.
- Komplex** Complex linear equation solver using equivalent real formulations [2], built on top of Epetra and AztecOO.
- LOCA** Library of continuation algorithms. A package of scalable stability analysis algorithms (available as part of the NOX nonlinear solver package). When integrated into an application code, LOCA enables the tracking of solution branches as a function of system parameters and the direct tracking of bifurcation points.
- Meros** Segregated preconditioning package. Provides scalable block preconditioning for problems that couple simultaneous solution variables such as Navier-Stokes problems.
- ML** Algebraic multi-level preconditioner package. Provides scalable preconditioning capabilities for a variety of problem classes.

- Moertel Supplies capabilities for nonconforming mesh tying and contact formulations in 2 and 3 dimensions using Mortar methods.
- MOOCHO MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization) is designed to solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems (i.e. nonlinear programs) using reduced-space successive quadratic programming (SQP) methods.
- NOX A collection of nonlinear solvers, designed to be easily integrated into an application and used with many different linear solvers.
- Petra A Greek term meaning “foundation.” Trilinos has three Petra libraries: Epetra, Tpetra and Jpetra that provide basic classes for constructing and manipulating matrix, graph and vector objects. Epetra is the current production version that is split into two packages, one core and one extensions.
- Epetra Current C++ production implementation of the Petra Object Model. The “E” in Epetra stands for “essential” implying that this version provides the most important capabilities that are commonly needed by our target application base. Epetra supports real, double-precision floating point data only (no single-precision or complex). Epetra avoids explicit use of some of the more advanced features of C++, including templates and the Standard Template Library, that can be impediments to portability.
- Tpetra The future C++ version of Petra, using templates and other more advanced features of C++. Tpetra supports arbitrary scalar and ordinal types, and makes extensive use of advanced C++ features.
- Jpetra A Java implementation of Petra, supporting real, double-precision data. Written in pure Java, it is designed to be byte-code portable and can be executed across multiple compute nodes.
- Pliris An object-oriented interface to a LU solver for dense matrices on parallel platforms. These matrices are double precision real matrices distributed on a parallel machine.
- PyTrilinos A set of python wrappers for selected Trilinos packages. Allows a python programmer to dynamically import Trilinos packages into a python script or the python command-line interpreter, allowing the creation and modification of Trilinos objects and the execution of Trilinos algorithms, without the need to constantly recompile.
- RTOp RTOp (reduction/transformation operators) provides the basic mechanism for implementing vector operations in a flexible and efficient manner.

- Rythmos** A transient integrator for ordinary differential equations and differential-algebraic equations with support for explicit, implicit, one-step and multi-step algorithms. Aimed at supporting operator-split algorithms, multi-physics applications, block linear algebra, and adjoint integration.
- Sacado** A set of automatic differentiation tools for C++ applications. Provides templated classes for forward, reverse and Taylor mode automatic differentiation.
- Stratimikos** Contains a unified set of Thyra-based wrappers to linear solver and preconditioner capabilities in Trilinos. Can also be used for unified testing of linear solvers and preconditioners.
- Teuchos** A collection of classes and service software that is useful to almost all Trilinos packages. Includes reference-counted pointers, parameter lists, templated interfaces to BLAS, LAPACK and traits for templates.
- Thyra** A set of interfaces and supporting code that defines basic interoperability mechanisms between different types of numerical software. The foundation of all of these interfaces are the mathematical concepts of vectors, vector spaces, and linear operators. All other interfaces and support software is built on the basic operator/vector interfaces.
- Tpetra** See description under "Petra".
- TriUtils** A package of utilities for other Trilinos packages.
- WebTrilinos** A scientific portal; a web-based environment to use several Trilinos packages through the web.

# 1 Introduction

The Trilinos Project is an effort to facilitate the design, development, integration and ongoing support of mathematical software libraries. Trilinos provides a framework and set of tools for document and source code control, software issue tracking, developer and user communication, automatic testing, portable configuration and building, and software distribution. Trilinos also provides a set of core utility libraries that provide common vector, graph and matrix capabilities, as well as a common abstract interface for applications to access any appropriate Trilinos package.

A new software capability is introduced into Trilinos as a *package*. A Trilinos package is an integral unit usually developed by a small team of experts in a particular algorithms area such as algebraic preconditioners, nonlinear solvers, etc.

The overall objective of Trilinos is to promote rapid development and deployment of high-quality, state-of-the-art mathematical software in an environment that supports interoperability of packages while preserving package independence. The Trilinos design allows individual packages to grow and mature autonomously to the extent the algorithms and package developers dictate.

The Trilinos Developers Guide is meant to assist new and existing Trilinos package developers. Topics covered include how to configure and build Trilinos, what is required to integrate an existing package into Trilinos and examples of how those requirements can be met, as well as what tools and services are available to Trilinos packages. Also discussed are some common practices that are followed by many Trilinos package developers. Finally, a snapshot of current Trilinos packages and their interoperability status is provided, along with a list of supported computer platforms.

For a higher-level view of the Trilinos project, please see An Overview of Trilinos [12]. The Trilinos Tutorial is a useful resource for new and existing Trilinos users and developers [15].

An overview of current Trilinos package capabilities can be found on the Trilinos website at <http://trilinos.sandia.gov/capabilities.html> . A summary of the packages included in every Trilinos release is also on the website at [http://trilinos.sandia.gov/packages/release\\_status.html](http://trilinos.sandia.gov/packages/release_status.html) .

## 1.1 How To Use This Guide

Although all sections of this guide will be useful to most developers, it is worth mentioning that this guide supports three types of development activities:

1. New Project: Development of a new package using little or no existing software as a base. All sections of this guide are appropriate reading.
2. Integration of existing third-party software: In this case, existing software is being imported into the Trilinos framework. Section 7 is particularly important, as are Sections 3, 4 and 6.
3. Ongoing development: For existing Trilinos package developers, Sections 3 and 4 are designed as a reference for software engineering practices and policies for Trilinos development.

## 1.2 Typographical Conventions

Our typographical conventions are found in Table 1.

Notation	Example	Description
Verbatim text	<code>../configure --enable-mpi</code>	Commands, directory and file name examples, and other text associated with text displayed in a computer terminal window.
CAPITALIZED_TEXT	CXXFLAGS	Environment variables used to configure how tools such as compilers behave.
<text in angle brackets>	<code>../configure &lt;user parameters&gt;</code>	Optional parameters.

**Table 1.** Typographical Conventions for This Document.

## 2 Getting Started

This chapter covers some of the basics that a developer will need to know when beginning to work on the Trilinos project. We address how to obtain, configure and build Trilinos, as well as how to add files to an existing package.

**Tip:** Check out the Trilinos Developer Home Page at <http://software.sandia.gov/Trilinos/developer> .

### 2.1 Obtaining a Copy of Trilinos

Trilinos can be obtained in two different ways. Developers should obtain a copy of Trilinos via the Trilinos CVS repository. To access the repository, an account on software.sandia.gov is required. In addition, the user account must be a in the “cvs” group on software.sandia.gov. Those who are granted write access must also be in the “trilinos” group or, in some cases, some other package-specific group. To request an account, send a note to `trilinos-help@software.sandia.gov` . The following two environment variables must be set to access the repository:

**Command:** `CVSROOT :ext:your_user_name@software.sandia.gov:/space/CVS`

**Command:** `CVS_RSH ssh`

(Replace “your\_user\_name” with your user name on software.sandia.gov.)

To checkout a working copy of the development branch of Trilinos in the current directory, type

**Command:** `cvs checkout Trilinos`

To checkout a working copy of only one package of Trilinos in the current directory, type

**Command:** `cvs checkout <package_name>`

(Replace “package\_name” with the name of the package.)

If the machine that is downloading Trilinos from software.sandia.gov has gzip available, it is possible to speed up the checkout process using the `-z` option. For example, to checkout all of Trilinos using the most aggressive compression, type

**Command:** `cvs -z 9 checkout Trilinos`

For those not familiar with CVS, a brief discussion covering some of the most common CVS commands is available in Section A. For a more complete listing of CVS commands, see the GNU CVS Home Page [8].

Trilinos can also be obtained in the form of a tarball from the Trilinos website at <http://trilinos.sandia.gov/download/> .

## 2.2 Recommended Build Directory Structure

Via Autoconf and Automake the Trilinos configuration facilities provide a great deal of flexibility for configuring and building the existing Trilinos packages. However, unless a user has prior experience with Autotools, we very strongly recommend the following process to build and maintain local builds of Trilinos.

To start, we defined two useful terms:

- **Source tree** - The directory structure where source files are found. A source tree is obtained by expanding a distribution tar ball, or by checking out a copy of the Trilinos repository.

- Build tree - The directory structure where object and library files, as well as executables are located.

Although it is possible to run `./configure` from the source tree (in the directory where the configure file is located), we recommend separate build trees. The greatest advantage to having a separate build tree is that multiple builds of the libraries can be maintained from the same source tree. For example, both serial and parallel libraries can be built. This approach also eliminates problems with configuring in a 'dirty' directory (one that has already been configured in).

**Key Point:** ... we recommend separate build trees ... multiple builds of the libraries can be maintained from the same source tree ... problems with configuring in a 'dirty' directory (are eliminated) ...

Setting up a build tree is straight-forward. Figure 1 illustrates the recommended layout. First, from the highest directory in the source tree (Trilinos for a repository copy, `trilinos-8.0.1` for a distribution), make a new directory - for an MPI build on a Linux platform, a typical name is `LINUX_MPI`. Finally, from the new directory, type

**Command:** `../configure --with-mpi-compilers`

(Note that various configure options might be necessary, see Section 2.3 for details.) Finally, type

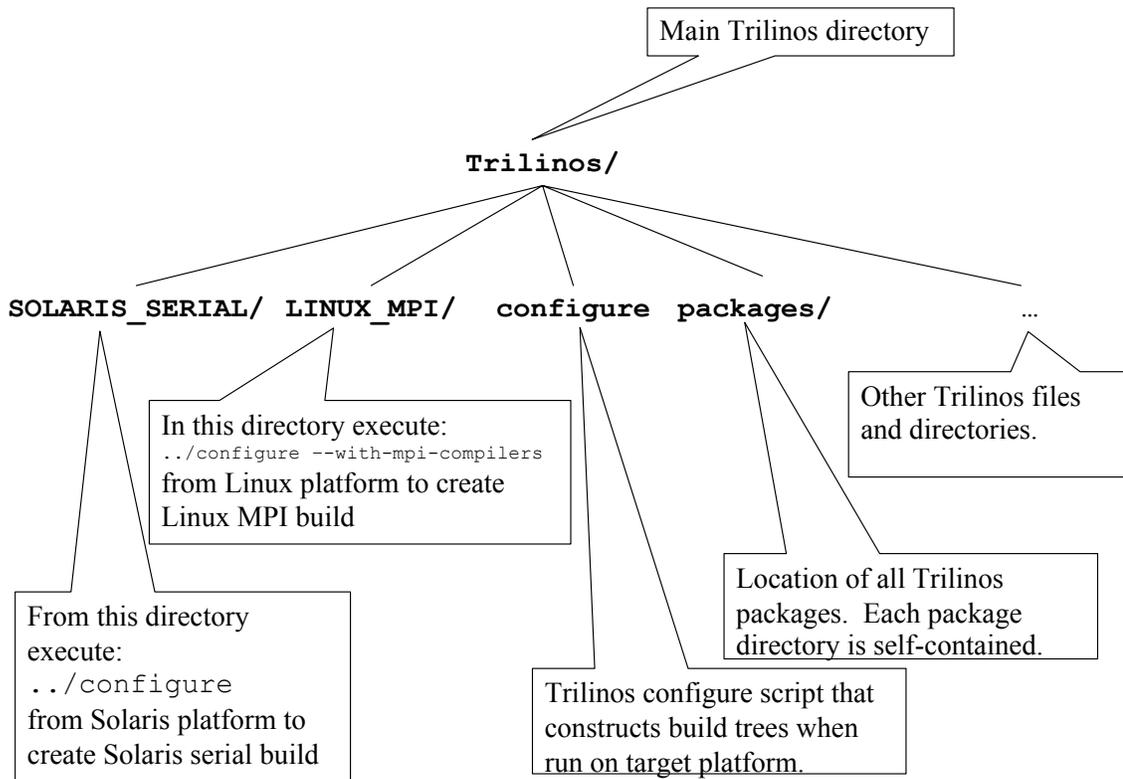
**Command:** `make everything`

In summary:

```
cd Trilinos
mkdir LINUX_MPI
cd LINUX_MPI
../configure --with-mpi-compilers
make everything
```

At this point, the MPI version of Trilinos on a Linux platform is built and completely contained in the `LINUX_MPI` directory. No files outside this directory have been modified. This procedure can be repeated for any number of build targets.

**Note:** Although we recommend the above location for build trees, they can be set up anywhere.



**Figure 1.** Recommended Layout for Trilinos Build Directories

**Note:** `make everything` builds all of the libraries, tests, and examples that were enabled at configure time. For more information about make targets, see Section 2.5

## 2.3 Configuring Trilinos

The most common issue encountered when configuring Trilinos is that it is nearly impossible to determine what caused configure to fail based on the standard output. If the output from configure is inadequate, look at the `config.log` file (in the buildtree) for the package that failed to configure properly.

**Key Point:** ... to determine what caused configure to fail ... look at the `config.log` file ...

To determine which package failed to configure, look at the bottom of the output from the `configure` command. One of the last lines will say something like:

```
configure: error: /bin/sh '././././packages/epetra/configure'
failed for packages/epetra
```

This particular error indicates to look in `packages/epetra/config.log`.

To configure from a remote build tree, simply run the configure script in source tree from the root of the build tree. In the example above, `cd` to the `SOLARIS_SERIAL` directory and type

**Command:** `./configure <configure options>`

A detailed list of configure options can be seen by typing

**Command:** `./configure --help=recursive`

from the top level of the source tree. This will display the help page for the Trilinos level as well as all Trilinos packages that use Autoconf and Automake. The output from this command is quite extensive. To view the help page for an individual package, `cd` to the home directory for the package in the source tree and type

**Command:** `./configure --help`

This command will also display the help page for Trilinos level options when used from the Trilinos home directory in the source tree.

Many of the Trilinos configure options are used to describe the details of the build. For instance, serial or mpi, all of the packages, or just a proper subset.

To configure for serial libraries, no action is necessary, but to configure for parallel libraries, a user must append appropriate arguments to the configure invocation line as described in “Trilinos Configuration Options”, section 2.4.

Also, to build the default set of Trilinos libraries, no action is necessary, but to exclude a package that is built by default, AztecOO for example, append `--disable-aztecoo` to the configure invocation line. Similarly, to include a package that is not currently built by default, Komplex for example, append `--enable-komplex` to the configure invocation line. Most Trilinos packages are not built by default; virtually all users will have to enable additional packages.

Users are strongly encouraged to build only the packages that are necessary because configuring and building can take a long time. It is well worth the time to look at which packages are built by default enable and disable packages as necessary.

**Key Point:** ... build only the packages that are necessary because configuring and building can take a long time.

It is recommended that users always configure from the Trilinos level and use `--disable-<package>` as required, rather than trying to configure from a lower level. To see which packages are built by default and which ones aren't, simply cd to the Trilinos home directory and type

**Command:** `./configure --help`

#### NOTES:

1. **Enabling/Disabling package builds:** The configure process is set up to detect when a `--disable-<package>` option would break a package dependency. For example, Ifpack depends on Epetra, so if a user wants to build Ifpack, but types `--disable-epetra`, Epetra will be configured and built anyway.
2. **Installing libraries and header files:** To install libraries and header files in a particular location, use `--prefix=<dir>` on the configure line. If this option is used, libraries will be located in `<dir>/lib` and header files in `<dir>/include/<package>`.

3. **Providing additional information to Autotools:** Although Autotools will try to determine all configuration information, the user must provide anything that Autotools needs and cannot find. Also, if Autotools selects, for example, the wrong BLAS library by default, the user must indicate which BLAS library to use. Other issues such as standards non-compliance are also commonly dealt with using configure options. If all required libraries (often the BLAS and LAPACK) are located in standard places and no special compiler flags are required, try configuring without providing additional information.
4. **Sample configure invocation scripts:**

Sample configure invocation scripts for a wide variety of platforms can be found in `Trilinos/sampleScripts`. These scripts are generally named using the following convention: `arch_comm_machine`. For example, `sgi64_mpi_atlantis`.

**Key Point:** Sample configure invocation scripts for a wide variety of platforms can be found in `Trilinos/sampleScripts`.

Note that these scripts are examples only and are primarily useful for the values of options such as `LDFLAGS`, `CPPFLAGS`, and `CXXFLAGS`. Do not expect to be able to find a script that can be used without modification; try to find a script for a similar machine to use as a guide. Users will also have to enable the correct set of packages based on their individual needs.

The scripts in the repository are not always up to date. If a user submits a script for a machine that few Trilinos developers have an account on, that script may become obsolete if it is not updated by the user who submitted it.

Users who create scripts for other machines are encouraged to check them into the repository for the benefit of other users. Users who do not have access to the repository can send scripts to `trilinos-help@software.sandia.gov`.

The following is an example configure invocation script for an SGI machine:

```
../configure --enable-mpi --with-mpi-libs=-lmpi \
--with-cflags=-64 --with-fflags=-64 \
--with-cxxflags="-64 -LANG:std -LANG:ansi-for-init-scope=ON \
-ptused -DMPI_NO_CPPBIND" \
LDFLAGS=" -64 -L/usr/lib64/mips4/r10000 -L/usr/lib64/mips4 \
-L/usr/lib64 " \
--enable-epetraext --enable-new_package \
--disable-komplex --enable-thyra
```

## 2.4 Trilinos Configuration Options

The following options apply to all Trilinos packages unless an option doesn't make sense for a particular package (for example, a package that does not include any Fortran code will not be sensitive to `F77=g77`), or otherwise noted. For options specific to an individual package, `cd` to the home directory of the package and type

**Command:** `./configure --help`

### Basic Options

- `--with-gnumake`

Gnu's make has special functions we can use to eliminate redundant paths in the build and link lines. **Use this option if you use GNU make to build Trilinos.** This requires that perl is in your path or that you have specified the perl executable with `--with-perl=<perl executable>`. For example `--with-perl=/usr/bin/perl`.

- `--with-blas=<lib>`

Specify which BLAS library to use. For example `--with-blas=/usr/path/lib/libblas.a`.

- `--with-lapack=<lib>`

Specify which LAPACK library to use. For example `--with-lapack="-L/usr/path/lib/ -llapack"`.

- `--enable-examples`

Build examples for all Trilinos packages. By default, this option is enabled. Note that after building the libraries, `make examples` can be used to compile all of the examples that were enabled at configure time.

- `--enable-tests`

Build tests for all Trilinos packages. By default, this option is enabled. Note that after building the libraries, `make tests` can be used to compile all of the tests that were enabled at configure time.

- `--enable-debug`

(NOX only.) This turns on compiler debugger flags. It has not been fully tested. As an alternate, specify `CXXFLAGS` on the configure line.

- `--enable-opt`  
(NOX only.) This turns on compiler optimization flags. It has not been fully tested. As an alternate, specify CXXFLAGS on the configure line.
- `--with-cppflags`  
Specify additional preprocessor flags (e.g., "-Dflag -ldir")
- `--with-cxxflags`  
Specify additional C++ flags
- `--with-ldflags`  
Specify additional linker flags (e.g., "-Ldir")
- `--with-ar`  
Specify a special archiver command, the default is "ar cru".

### Influential Environmental Variables

- CC  
C compiler command.
- CFLAGS  
C compiler flags.
- CXX  
C++ compiler command.
- CXXFLAGS  
C++ compiler flags.
- LDFLAGS  
Specify linker flags.
- CPPFLAGS  
C/C++ preprocessor flags.
- CXXCPP  
C++ preprocessor.
- F77  
Fortran 77 compiler command.

- `FFLAGS`  
Fortran 77 compiler flags.

### MPI-Related Options

- `--enable-mpi`  
Enables MPI mode. Defines `HAVE_MPI` in the `(Package)_Config.h` file. Will test for the ability to preprocess the MPI header file and may test ability to link with MPI. This option is rarely necessary as many of the below options also turn MPI on.
- `--with-mpi-compilers`  
Sets `CXX = mpicxx` (or `mpiCC` if `mpicxx` not available), `CC = mpicc` and `F77 = mpif77`. Automatically enables MPI mode. To use compilers other than these, specify MPI locations with the below options. If none of these options are necessary, use `--enable-mpi` to enable MPI mode. In this case, `CXX`, `CC`, and `F77` have to be set if the correct compilers are not chosen by default.
- `--with-mpi=MPIROOT`  
Specify the MPI root directory. Automatically enables MPI mode. If this option is set, `--with-mpi-incdir` and `--with-mpi-libdir` should not be used. `--with-mpi` is a shortcut for setting `--with-mpi-libdir=MPIROOT/lib` and `--with-mpi-incdir=MPIROOT/include`.
- `--with-mpi-libdir=DIR`  
Specify the MPI libraries location. Defaults to `MPIROOT/lib` if `--with-mpi` is specified. If multiple directories must be specified, try `--with-ldflags="-L<dir1> -L<dir2>"` instead.
- `--with-mpi-libs="LIBS"`  
Specify the MPI libraries. Defaults to `"-lmpi"` if either `--with-mpi` or `--with-mpi-libdir` is specified.
- `--with-mpi-incdir=DIR`  
Specify the MPI include files location. Defaults to `MPIROOT/include` if `--with-mpi` is specified. If multiple directories must be specified, try `--with-cppflags="-I<dir1> -I<dir2>"` instead.

### Developer-Related Options

- `--enable-maintainer-mode`  
Enable make rules and dependencies not useful (and sometimes confusing) to the casual installer.

## 2.5 Building and Installing Trilinos

If the configure stage completed successfully, just type

**Command:** `make`

to compile the libraries. To compile the tests and examples, type

**Command:** `make tests`

and

**Command:** `make examples`

For more about how to run the installation test suite, see Section 3.3. Finally, if `--prefix` was specified, type

**Command:** `make install`

to install libraries and headers.

Note that any code that links to Trilinos libraries must define `HAVE_CONFIG_H`.

<b>Key Point:</b> ... any code that links to Trilinos libraries must define <code>HAVE_CONFIG_H</code> .
--

Below is a listing of the make targets that will be most useful to users:

- `make` - Builds just the libraries
- `make install` - Installs just the libraries
- `make examples` - Makes just the examples (assumes libraries are built)
- `make install-examples` - Install the examples (assumes libraries are installed)
- `make tests` - Makes just the tests (assumes libraries are built)

- make everything - Builds libraries, tests, and examples
- make install-everything - Installs libraries and examples

## 2.6 Configure, Make and Install: Common Problems

Provided below is a list of common problems associated with configuring, building and installing Trilinos. A more complete FAQ list may be found online at

**Key Point:** A more complete FAQ list may be found online...

<http://trilinos.sandia.gov/faq.html> .

- Any code that links to Trilinos libraries must define `HAVE_CONFIG_H` .
- It is difficult to build Trilinos with different compiler versions. For example, if using g++ 4.1.1, it is best to use gcc 4.1.1 and gfortran 4.1.1. On some systems, multiple sets of compilers are available. If, for instance, there is one GCC 4 compiler set and one GCC 3 compiler set, it is possible that configure will pull the g++ and gcc compilers from GCC 4 and the g77 compiler from GCC 3.

This will usually result in a `gxx_personality` error in the `packages/teuchos/config.log` file. Another possible clue in the `config.log` file is paths for linking to libraries that refer to multiple compiler sets. For example, `-L/usr/lib/gcc/x86_64-redhat-linux/4.1.1` and `-L/usr/lib/gcc/x86_64-redhat-linux/3.3.2` .

The easiest way to avoid these problems is to explicitly specify which compilers you want to use in the list of configure arguments. For example, `CXX=/path/g++` , `CC=/path/gcc` , and `F77=/path/gfortran` .

- Trilinos should be built with the `--with-gnumake` option whenever GNU make is used to build Trilinos. On most systems, "make" will point to GNU make. It is easy to check, simply type `man make` on any UNIX-like machine. When building a large number of packages without using the `gnumake` option, the following error is common:

```
execv: Argument list too long
```

If GNU make is not available and this error is encountered, it will be necessary to disable the test or example that was linking when the error occurred.

- Do not attempt to specify optimization flags using the `--with-cxxflags`, `--with-cflags`, or `--with-fflags` options. Use `CXXFLAGS`, `CFLAGS` and `FFLAGS` instead. Use `--with-cxxflags`, `--with-cflags`, and `--with-fflags` to specify flags that are to be used in addition to the default optimization flags.
- When creating a configure invocation script, be sure to use line continuation characters properly. The characters should be at the end of every line, except the last line, and should not be followed by any spaces.
- To verify that the entire configure invocation script has been parsed by Autoconf, open the `config.status` file in the top level of the build tree and grep for the string "with options". Here you will find all of the options that Autoconf pulled from the invoke configure script.
- Autoconf cannot detect most spelling mistakes in configure invocation scripts.
- When experiencing problems during the make phase, it is often useful to `make clean` before attempting to `make` again. Sometimes it even helps to blow away the entire build tree and start over.
- When building with LAM under RH9 Linux, configure complains that it cannot find `mpi++.h`. The message in the `config.log` file is:

```
/usr/include/mpi.h:1064:19: mpi++.h: No such file or directory
```

The following modified configure invocation works:

**Command:** `../configure --enable-mpi CXX="mpiCC -DLAM_BUILDING"`

- The build process will fail on OSX if "DropZip" is used to unzip the Trilinos tarball. This utility truncates long file names.

## 2.7 Tips for Making the Configure, Make, and Install Processes More Efficient

Trilinos has grown to become a large piece of software. Not surprisingly, it can take a very long time to configure and build all of Trilinos. Below are some tips for speeding up the process:

- Only build the Trilinos libraries that are necessary.

An easy way to do this is to use the `--disable-default-packages` option. This option allows users to easily specify exactly which packages should be built. Packages that enabled packages are dependent on will be turned on automatically, so don't shy away from disabling all packages that are not used directly. If a package configures and builds that was not enabled explicitly, keep in mind that a package that was enabled probably depends on that package.

- Cache configure results.

Adding `--cache-file=config.cache` to the list of configure arguments will store the results of many of the tests that configure performs in a file called `config.cache`, located in the top level of the build tree.

Using this option will allow the first package that is configured to store many test results that can be used by all subsequent packages. If it is later necessary to reconfigure with a different set of packages, all packages (including the first) can use these cached values. If configure options are changed that are associated with configure test results that have already been cached, or a relevant environment variable is changed, it is necessary to remove the `config.cache` file before reconfiguring. Failing to do so can cause the changes to be ignored.

When configuring even a handful of packages, during the first configure a speedup of greater than two is attainable and a speedup of greater than three is possible for subsequent configures.

- Decrease build time on some machines by creating multiple jobs.

If `-j` (jobs) is a valid option for `make`, specifying the `-j` option with a value of two times the number of processors that the machine has will typically result in a faster build process. For example, on a dual processor machine, try replacing `make` with

**Command:** `make -j 4`

during the build step.

On a single processor machine, the speedup is minimal; on a machine with multiple processors, the speedup can be quite significant. For example, speedups of 1.73 and 2.45 were observed on a dual processor machine and a four processor machine, respectively. Using two times the number of processors for the argument to the jobs option is only a suggestion based on observed performance; those who are interested in achieving optimal performance are encouraged to experiment with various values and to report their findings to [trilinos-help@software.sandia.gov](mailto:trilinos-help@software.sandia.gov). The `-j` can also be passed without a corresponding value, in other words

**Command:** `make -j`

When used in this way, the number of jobs created is unlimited. For machines with a large number of processors this appears to work in some situations, but machines with fewer than four processors tend to get bogged down with overhead.

## 2.8 Adding and Removing Source Files

Commonly a developer needs to add files to or remove files from a Trilinos package.

This process can be divided into the following steps:

1. Obtain the supported versions of Autoconf and Automake
2. Update source code from Trilinos repository
3. Add new files to or remove obsolete files from the Trilinos repository
4. List new files in or remove obsolete files from Makefile.am
5. If adding a new Makefile.am, update configure.ac and the parent Makefile.am
6. Bootstrap
7. Test the new code
8. Update source code from Trilinos repository
9. Commit the changes to the Trilinos repository

We describe these steps below in greater detail. The steps assume a simple addition or removal of source files from a Trilinos package that uses Autotools. Special situations such as adding header file or library dependencies or conditionally compiling new files require a more complicated process. In addition, many of the restrictions listed below apply only to development and release branches. If a branch is established for a separate purpose (for example, to attempt an experimental restructuring of existing code), the restrictions do not apply. However, in this case, the restrictions do apply when any changes from the branch are to be merged back into the development branch.

The section frequently mentions various CVS commands. For more information on these CVS commands, see Section A. For a more complete listing of CVS commands, see the GNU CVS Home Page [8].

1. Obtain the supported versions of Autoconf and Automake.

The current supported versions of Autoconf and Automake are documented in `Trilinos/config/AutotoolsVersionInfo`, which can be found in the Trilinos repository. Do not assume that the most recent versions Autoconf and Automake are supported. The supported versions of Autoconf and Automake can always be found on [software.sandia.gov](http://software.sandia.gov). This makes software a good machine to bootstrap on.

2. Update source code from Trilinos repository

Obtain the most current version of Trilinos (for the branch being worked on). From the top Trilinos directory type

**Command:** `cvcs -q update -dP`

3. Add new files to or remove obsolete files from the Trilinos repository

If a whole new directory, `abkdir`, is to be added, type

**Command:** `cvcs add abkdir`

to add `abkdir` to the repository. This must be done before adding any of the contents of `abkdir`.

To add new files `abc.cpp` and `abc.h` to the Trilinos repository, type

**Command:** `cvcs add abc.cpp abc.h`

in the directory where the files are located (in a checked out version of the Trilinos repository). To remove the same files, type

**Command:** `cvcs remove abc.cpp abc.h`

Note that directories cannot be removed from the repository using `cvcs remove .`

4. List new files in or remove obsolete files from Makefile.am

New source files should be placed into a category in the appropriate `Makefile.am`. Typically, the directory in which the new files are located will contain a `Makefile.am`, but sometimes source files are listed in the `Makefile.am` one directory above the files. When adding a new directory that will require a `Makefile` (for example, when adding a new test directory), create a new `Makefile.am`. Usually a developer can find a `Makefile.am` to use as a template in the `new_package` package. To remove files from the build process, delete the file names from the appropriate `Makefile.am`.

## 5. If adding a new Makefile.am, update configure.ac and the parent Makefile.am

When adding a new Makefile.am, the corresponding Makefile must be listed in the configure.ac file that is located in the same package as the Makefile.am (if the Makefile.am is not a part of any package, ie part of the general Trilinos framework, list the corresponding Makefile in the Trilinos level configure.ac). The list of Makefile's is near the very bottom of the configure.ac file.

Any new directories that contain a Makefile.am must be listed in the Makefile.am in the directory immediately above the new directory. List the new directory on the SUBDIRS line. If the parent directory does not contain a Makefile.am, create it and add the name of the parent directory to the SUBDIRS line in the Makefile.am in the directory above the parent directory (repeat as necessary to reach an existing Makefile.am).

Use  `cvs add`  to add all new Makefile.am's.

## 6. Bootstrap

First, from the top-level directory of the appropriate Trilinos package (for example Trilinos/packages/epetra), type

**Command:**  `./bootstrap`

The bootstrap should complete without any errors. Add any Autotools files that are generated during the bootstrap using  `cvs add .`

## 7. Test the new code

Reconfigure and rebuild the Trilinos package. Perform tests associated with the new code, as well as the rest of the tests for the package to insure that both the new code works and existing code has not been broken. When changes could possibly affect other packages, tests for affected packages should also be run. The simple way to run all of the required tests is to use the `runtests` utility, which is a component of the Trilinos test harness and can be accessed via the `runtests-serial` and `runtests-mpi` make targets. To run the test suite (the tool will only attempt to run tests that were built), simply build the libraries and tests and then type:

**Command:**  `make runtests-serial`

The `runtests-mpi` target is very similar, but expects to you supply the variable `TRILINOS_MPI_GO`, either in the environment or as an `;` argument to `make`. Here is an example of how to call the `runtests-mpi` make target:

**Command:**  `make runtests-mpi TRILINOS_MPI_GO="'mpiexec -np '"`

Alternatively, the following set of arguments can be used:

`TRILINOS_MPI_MAX_PROC=2 TRILINOS_MPI_GO="' /bin/linux/mpirun -np '"`

. Note that in either case, the `TRILINOS_MPI_GO` argument must be double quoted.

#### 8. Update source code from Trilinos repository

There are two good reasons to update the source code again. First, other developers could have committed changes during the past several steps of this process. Though this is unlikely, it is worth checking. If changes were committed, minimally the testing step will need to be redone. If files related to configuring or building were modified, more will have to be done if collisions occur. Some of the possibilities are beyond the scope of this introductory document; however, we will briefly discuss the most common collision scenario. Typically the generated files will contain collisions (for example `configure`, `Makefile.in`, or `aclocal.m4`), while the changes in the files created by developers (for example `configure.ac` or `Makefile.am`) will be successfully merged by CVS. In this case, the best course of action is to remove the files with collisions, `cd` to the top level of the Trilinos package, perform a `cvsexec update`, and then begin the above process again from “Bootstrap” step. As long as the changes are merged in the non-generated files, bootstrapping should resolve the problem.

A second reason to update again before committing changes is to avoid confusion. After a bootstrap, all of the generated files will get an updated timestamp, but in most cases only some of the files will actually be modified. If a developer commits changes before updating, all of the generated files will be considered to have been modified. This is bad for several reasons. One of the most important is that when committing changes, a developer should always verify that the list of files that are about to be committed makes sense. A `cvsexec update` will check to see if the file has really been changed or if it simply has a new timestamp.

#### 9. Commit the changes to the Trilinos repository

Once all of the above steps are completed, the final step is to commit the changes to the repository. Start by typing

**Command:** `cvsexec commit`

Next verify the list of files that appears and enter an appropriate log message. Developers who are unfamiliar with the process of committing changes should see Section A for a more detailed description of this process.

Finally, save the file and exit the CVS editor to commit the changes.

When using the above process to commit new source code, the new source must be functioning properly, otherwise the repository will not be stable. At the same time, developers are encouraged put new code into the repository

during initial development. This will ensure that work is backed up and provide version control. When adding unstable code to the repository, only two steps are necessary. First, use the `cv`s `add` command as mentioned above, and then modify the commit command slightly to commit only the new source by typing

**Command:** `cv`s `commit` `newfile1.cpp` `newfile2.cpp`

Provided that the new files are not added to the make structure, the addition of the new files should not negatively affect the repository. Distribution tarballs will even skip over the new source. A common log message for this type of commit is simply “Checking in for safe keeping; code is not yet functioning”. Developers are encouraged to include a short description of what the code will do when it is complete.

## 3 Services Available to Trilinos Packages

A number of services exist for Trilinos packages. Many of these services relate directly to the requirements and suggested practices for Trilinos packages. For example, the CVS repository is discussed below, and Trilinos packages must make use of this repository. Also, Bonsai, Bugzilla and Mailman are all tools that relate to suggested practices. (It should be noted that these services are not simply meant to satisfy SQE requirements. Rather, Bonsai, Bugzilla and Mailman have proved to be very useful tools. Using these tools together, along with the CVS repository, has led to a more time and cost effective code development process.) For more information about any of the below services, please contact the Trilinos Project Leader.

### 3.1 Configuration Management

Autoconf [6], Automake [7] and Libtool [11] provide a robust, full-featured set of tools for building software across a broad set of platforms (see also the “Goat Book” [20]). Although these tools are not official standards, they are commonly used in many packages. Nearly all existing Trilinos packages use Autoconf and Automake. However, use of these tools is not required.

Package developers who are not currently using autotools, but would like to, can get a jump start by using a Trilinos package called “new\_package”. This trivial

package exists for one primary purpose. It walks a developer through the process of setting up a package to configure and build using autotools. General instructions for how to get started can be found in Section 7.

Trilinos provides a set of M4 [9] macros that can be used by any other package when its Autoconf and Automake configure and build system is being setup. These macros perform common configuration tasks such as locating a valid LAPACK [1] library, or checking for a user- defined MPI C compiler. The macros can be found in the Trilinos CVS repository in Trilinos/config. These macros minimize the amount of redundant effort in using Autotools, and make it easier to apply a general change to the configure process for all packages.

## 3.2 Regression Testing

Trilinos provides a variety of regression testing capabilities. Within a number of Trilinos packages, we employ “white box” testing where detailed information about the software is used and probed. For some packages, “black box” testing is performed via the Thyra virtual class interfaces. Any package that implements the Thyra interfaces (see Section 6.2) can be tested via this black box test environment.

## 3.3 Test Harness

Trilinos packages that configure and build using Autotools can easily utilize the Trilinos test harness. The test harness builds Trilinos and runs tests on multiple platforms; most often the test harness is run nightly. Packages that have not ported to a particular platform can be excluded from the testing process on that platform. Packages that do not have any tests integrated into the test harness can still benefit by testing that libraries build without any errors.

The process for integrating existing tests into the testharness is discussed here:  
[http://software.sandia.gov/trilinos/developer/test\\_harness/test\\_definitions.html](http://software.sandia.gov/trilinos/developer/test_harness/test_definitions.html)

## 3.4 CVS Repository

Trilinos source code is maintained in a CVS [8] repository. It is very easy to add new packages to the repository. Packages that already use CVS can even retain their CVS history! Instructions for obtaining a copy of Trilinos via the CVS repository can be found in Section 2

For those not familiar with CVS, a brief discussion covering some of the most common CVS commands is available in Section A. For a more complete listing of CVS commands, see the GNU CVS Home Page [8].

## 3.5 Bonsai

The CVS history of the Trilinos project is accessible via a web-based interface package called Bonsai [16]. This tool can be found on the web at <http://software.sandia.gov/bonsai> .

Bonsai gives developers the ability to view the changes made to the files in the repository. Developers can search based on filename, directory, branch, date, user who made the change, or any combination of these criteria. The differences between any two versions of a file may also be viewed, which can be very helpful when debugging.

<p><b>Key Point:</b> The differences between any two versions of a file may also be viewed, which can be very helpful when debugging.</p>
---

## 3.6 Bugzilla

Feature and issue reports are tracked using Bugzilla [17]. Bugzilla can be found on the web at <http://software.sandia.gov/bugzilla> . A Bugzilla account is necessary for submitting bugs. Those interested can sign up at the website. All issues related to any Trilinos package that uses Bugzilla should be submitted to Bugzilla. This even applies to cases in which one developer diagnoses and fixes a bug within a short period of time. A bug report is still very valuable in this case because it provides an artifact that outlines the problem and explains how the problem was fixed. A bug report should be filled out with as much detail as possible. Likewise, after a bug has been resolved, the developer should also provide a detailed description of the solution that was used. Below we will discuss two underutilized features of Bugzilla as well as the concept of a meta bug, which has proven to be very useful to the Trilinos development team.

Two commonly underutilized features of Bugzilla are the ability to assign a priority and a severity to a bug. Taking advantage of these features helps users more clearly express their concerns and helps developers organize tasks. Below are recommended usage guidelines for the priority and severity settings for Bugzilla bugs that are filed against Trilinos.

**Severity:** Severity should be determined by the person who files the bug (reporter) and should be based solely on how severe the reporter perceives the bug to be

at the time the bug is filed. Considerations for selecting a proper severity include whether or not the bug is currently directly impeding progress, and if it has the potential to impede progress in the future. There is also an important distinction between enhancements and faults in existing functionality. Severity can be updated if conditions warrant a change. Possible severity settings include blocker, critical, major, normal, minor, trivial, and enhancement. The severity of a bug is not necessarily related to its priority.

**Priority:** Priority should be determined by the owner of the bug. Priority can be chosen based on the relative importance of other existing bugs, as well as on how important the bug is to Trilinos and Sandia. Other factors can come into play when selecting a priority for a bug. The priority of a bug can be updated if conditions warrant a change. Possible priority settings include P1, P2, P3, P4, and P5. P1 is the highest priority, P5 is the lowest.

Trilinos reserves the P1 classification for bugs that meet the definition of an ASC defect. ASC defines a defect to be “A flaw in a system or a system component that causes the system or component to fail to perform its required function.” In this context, this definition (and therefore the P1 priority designation) applies only to Trilinos packages that are currently funded by ASC. The reason the P1 priority is reserved is that ASC requires that Trilinos report defect metrics on a quarterly basis, and using the P1 designation significantly reduces the cost of gathering this data. Note that the P1 designation also only applies to ASC required capabilities present in the current release version of Trilinos, or scheduled to be a part of the next release version and does not apply to enhancements.

Here are some examples of bugs that should be classified as defects, and therefore assigned P1 priority:

- A compile error, on a release branch or the development branch, in the source code of a package that is required to build on the platform that the error is occurring on.
- Any issue (scaling, correctness, etc) with a required feature on a required platform that prevents the feature from performing as required, provided the feature is not under initial development, which would make it an enhancement.
- A compile error involving a non-required feature on a required platform that makes it impossible to compile all required features on the platform. (Ex - an experimental feature that wont compile and makes it impossible to compile the rest of the library without manually modifying the build system.)

Here are some examples of bugs that should not be classified as defects:

- A compile error in the examples or tests of a package, unless the example or test is itself a specific requirement.
- A compile error in the source code of a package that is not currently released or scheduled for release in the next release cycle.
- A compile error in the source code of any package on a platform that is not a required platform.
- A compile error involving a non-required feature that is not compiled by default, but can be enabled at configure time (provided the configure option doesn't also enable any required features that therefore cannot be built on a required platform).
- Any issue involving the release process itself, documentation, software quality requirements, or any other issue not related to the delivery of requirements.

A good rule of thumb for priority designations P2-P5 is to assign priorities P2 and P3 to bugs that should be fixed before the next release and priorities P4 and P5 to bugs that do not necessarily need to be addressed prior to the next release. The priority of a bug is not necessarily related to its severity.

Bugzilla allows a bug to be listed as dependent on another bug. If bug 1 is dependent on bug 2, bug 2 is called a “blocker” for bug 1. The ability to make one bug dependent on another is useful for multiple reasons. Trilinos developers often make use of what are informally called “meta bugs”. Meta bugs are larger tasks that are broken into several smaller tasks (there are no special requirements for filing a meta bug, a meta bug is just a bug that depends on other bugs). For example, a meta bug with the summary “Port to all ASC platforms” could be filed against a new package. A separate bug could be filed for porting to each individual ASC platform, and then all of those bugs would block the meta bug. The meta bug concept is used during the Trilinos release process. One meta bug is filed for the Trilinos release process. That bug depends on the bugs for each package release process and the Trilinos web release process. Each package also has a meta bug associated with its release process. The use of meta bugs makes the Trilinos release process much more manageable at the framework and package levels.

NOTE: In the context of Bugzilla, “bug” can refer not only to an error in existing code, but also to a desired enhancement. For example, a bug report should be submitted to Bugzilla to report a segmentation fault that occurs when using an

existing Ifpack preconditioner, and a bug report should also be submitted to request a new Ifpack preconditioner. “Issue” and “bug” are used interchangeably in the discussion of Bugzilla in this guide.

## 3.7 Mailman

Email lists are maintained for Trilinos as a whole and for each package through Mailman [10]. This tool can be found on the web at <http://software.sandia.gov/mailman/listinfo>. Those interested in signing up for one or more lists may do so at the website. Non-Sandians are able to sign up for the “Users” and “Announce” lists. Sandians should keep this in mind when posting to these lists.

Lists for new packages can be set up very easily. Each package usually has five mailing lists. The example mailing lists mentioned below are to be used for issues relating to all of Trilinos. The names for the lists pertaining to individual packages follow the same naming scheme, simply replace “Trilinos” with the name of the package. For example, the list for Trilinos users is called Trilinos-Users and the email address is `trilinos-users@software.sandia.gov`. The list for Epetra users is called Epetra-Users and the associated email address is `epetra-users@software.sandia.gov`.

**Tip:** While those who use Epetra (or any other Trilinos package) are also “Trilinos users”, the lists are not set up to recognize this. In other words, those who subscribe to the Epetra-Users mailing list do not necessarily form a subset of those who subscribe to the Trilinos-Users mailing list. This is also true of all other list types. Keep this in mind when subscribing and posting to lists.

- **Trilinos-Announce** `trilinos-announce@software.sandia.gov`  
All Trilinos release announcements and other major news.
- **Trilinos-Checkins** `trilinos-checkins@software.sandia.gov`  
CVS commit log messages that are related to Trilinos in general or packages that have not had separate lists established.
- **Trilinos-Developers** `trilinos-developers@software.sandia.gov`  
All discussions related to Trilinos-specific development (not specific to a Trilinos package) are conducted via this list. Important development decisions that originate in other places (regular email, discussions, etc) should also

be posted to this list (or to the appropriate package list). By doing this, the list archive can provide a record explaining why various changes were made over time.

- Trilinos-regression `trilinos-regression@software.sandia.gov`  
All regression test output that is not specific to a package.
- Trilinos-Users `trilinos-users@software.sandia.gov`  
List for Trilinos Users. General discussions about the use of Trilinos.
- Trilinos-Leaders `trilinos-leaders@software.sandia.gov`  
Mailing list for representatives for each Trilinos package. There are no leaders lists for individual packages.

## 3.8 Package Website Template

A template is available at the `new_package` website [http://trilinos.sandia.gov/packages/new\\_package](http://trilinos.sandia.gov/packages/new_package) for creating websites for new packages. Package developers are free to use this template or create their own website from scratch. The `new_package` website also contains information about many of the other services that are available to Trilinos packages.

## 3.9 Portable Interface to BLAS and LAPACK

Portable interface to BLAS and LAPACK: The Basic Linear Algebra Subprograms (BLAS) [14, 4, 3] and LAPACK [1] provide a large repository of robust, high-performance mathematical software for serial and shared memory parallel dense linear algebra computations. However, the BLAS and LAPACK interfaces are Fortran specifications, and the mechanism for calling Fortran interfaces from C and C++ varies across computing platforms. Epetra, Tpetra, and Teuchos each provide a set of simple, portable interfaces to the BLAS and LAPACK that provide uniform access to the BLAS and LAPACK across a broad set of platforms. These interfaces are accessible to other packages.

# 4 Trilinos Package Requirements

The philosophy of the Trilinos project is to minimize the number of explicit requirements placed on packages. Instead, we attempt to describe high-level requirements coupled with *suggested practices*. This approach allows freedom to define

how requirements are satisfied yet, at the same time, provides guidance and support for packages that do not have a full set of established software engineering practices. In rare cases, requirements may be waived for packages on a case-by-case basis with the approval of the Trilinos Project Leader.

Package requirements can be split into two basic categories:

1. Interoperability mechanisms: Depending what a new Trilinos package does, it should be able to interact with one or more other Trilinos packages. Often this means being able to accept an application matrix and vector objects as either TSF objects or Epetra objects, and that the package implements relevant TSF abstract interfaces. Response to Trilinos configuration options also falls in this category.
2. Software engineering processes: This category includes formal support for software design, implementation and support, including processes for capturing user requirements, documenting design, source control, user documentation, issue tracking and product release.

Trilinos package requirements and suggested practices are summarized in Table 2.

<b>Requirement</b> Package must:	<b>Suggested Practice</b> Package can:
Keep source files as a self-contained collection in a single directory under the <code>Trilinos/packages</code> directory in the Trilinos CVS repository. Change logs must be archived and communicated to interested Trilinos developers.	Utilize Trilinos Mailman lists to archive and communicate software change logs.
Have process in place to port to all supported platforms	Use the Trilinos Autotools environment and leverage the existing portability facilities already used by numerous packages.
Respond to all relevant configure options	Use Autoconf and Automake, utilizing the collection of Trilinos M4 macros to minimize extra effort.
Respond to software faults in a timely manner	Use Trilinos Bugzilla site to record and track software issues, responding to issues in order of priority.
Provide unit and regression testing	Register test scripts with the Trilinos test harness, which runs nightly on a variety of supported platforms and can be used by developers before checking in changes.

**Table 2.** Trilinos Package Requirements and Suggested Practices.

Although there are several requirements listed in Table 2, we have structured the integration process so that packages can be incorporated into Trilinos in a gradual

manner. Listed below are four levels of requirement compliance. It is common for new packages to address these steps one at a time, and not necessarily in the listed order.

## 4.1 Add Package to Trilinos Repository

Except for rare instances, placing a package in the Trilinos CVS repository is a minimum requirement for any package to become part of Trilinos. Other than receiving approval from the Trilinos Project Leader to add a new package to Trilinos, there are no prerequisites for adding a package to the Trilinos repository. At this stage, it does not matter if the package is finished. In fact, we encourage developers to keep source files in the repository from package inception so that source code is backed up and properly managed. Our primary restrictions are:

1. A package must be buildable on one or more platforms in order to be added to the Trilinos level configure and build structure.
2. A package must be portable to all supported platform in order to be built by default using the top-level Trilinos configure process.

A package can remain in a predistribution state indefinitely. Any package that is not ready or approved for release can easily be omitted from a distribution.

<b>Key Point:</b> A package can remain in a predistribution state indefinitely.
---

## 4.2 Port Package to All Supported Platforms

Although use of Trilinos Autotools is the easiest and most robust way to ensure portability across all supported platforms, a package is not required to use them. At the same time, a package must provide some mechanism to build across all platform that Trilinos supports. Typically, if not using autotools, this support would be in the form of platform-specific makefiles that the installer could invoke for a given platform.

## 4.3 Respond to All Relevant Configure Options

The Trilinos top-level configure script accepts numerous configuration options as described in Section 2.3. To the extent that each option is appropriate, a package should respond to each option. For example, if a package can be built with MPI support, it should respond to the `--enable-mpi` option.

Note that this does not mean the package must use Trilinos Autotools, but must simply be sensitive to certain defined parameters that are generated when the Trilinos autotools scripts are invoked.

## 4.4 Respond to Issue Reports in a Timely Manner

The Trilinos Team does not have any specific requirements concerning how bugs should be submitted and processed. However, packages should have a process in place that deals with issue tracking. Packages developer teams that are looking for an efficient and useful issue tracking tool are encouraged to consider using Bugzilla, which is discussed in Section 3.6.

# 5 Suggested Software Engineering Practices

There are many ways to define an effective software engineering process. As a result, the Trilinos project specifies very few *requirements*. At the same time, many software packages do not have well-defined practices to support good software engineering. In this section, we discuss suggested practices based on our experience with some common tools and processes for software engineering. We want to strongly emphasize that these are *suggested* practices only and we discuss them here in order to facilitate adoption of practices for packages that have few existing practices in place.

<p><b>Key Point:</b> ... the Trilinos project specifies very few <i>requirements</i> ... we discuss <i>suggested practices</i> here in order to facilitate adoption of practices for packages that have few existing practices in place ...</p>
---

## 5.1 Preliminary Steps

Prior to anything else, a new Trilinos package should have the following infrastructure established. Visit the Trilinos home page [13] for information on who to contact for these preliminary steps.

The preliminary steps are:

1. Set up user accounts for each package developer on `software.sandia.gov`.
2. Establish Bugzilla Product and Component Definitions for the new package, identifying who will be default owner of each component.
3. Establish Email Lists for the package. Five lists will be defined:
  - (a) `PackageName-Announce@software.sandia.gov` : Announcements such as new releases, feature lists and any other newsworthy items will be sent to this list. Any person interested in any aspect of the package should subscribe to this list.
  - (b) `PackageName-Checkins@software.sandia.gov` List to which all CVS commit log message for the package are sent. Developers with an interest in the day-to-day activity of package development can subscribe to this list.
  - (c) `PackageName-Developers@software.sandia.gov` List by which all development discussions are conducted, or to which notes from development discussions are sent and archived. This is also the list to which detailed design documentation is sent for review and archiving.
  - (d) `PackageName-Regression@software.sandia.gov` List to which all automated regression test results are sent for archival purposes.
  - (e) `PackageName-Users@software.sandia.gov` User forum where package users can communicate with each other. Developers should monitor this list and interject comments as necessary.

These preliminary steps can generally be completed in a few hours. Once complete, the new package has a set of tools in place that address a large fraction of software engineering practices.

## 5.2 Practices to Support the Software Lifecycle

One common view of software engineering processes breaks the process down into seven phases:

1. Requirements.
2. Specification.
3. Design.

4. Implementation.
5. Integration.
6. Maintenance.
7. Retirement/Replacement.

In this section we discuss suggested practices to address most of these phases. The value of adopting these specific practices is that they are commonly used or planned for use in a number of existing Trilinos packages. It is worth noting that testing is not a phase, but should be done at each of the above phases in the process as appropriate for that phase.

## 5.3 Requirements

The majority of requirements for Trilinos packages come either directly or indirectly from funded research proposals and plans. Although these requirements are sometimes difficult to elicit from the proposals and plans, we assume that a package is satisfying requirements by virtue of being funded. Therefore we suggest that package developers track their requirements as part of the communication with funding sponsors. Regardless of the source of requirements, the appropriate documents should be kept under source control.

## 5.4 Specification/Design

Package specifications can be done in many ways. An effective way for object-oriented, e.g., C++ packages is to use documented header files and a documentation tool such as Doxygen [19], and then communicate the generated HTML output to the package development team via the `package-developers@software.sandia.gov` email list. Package specifications created using another method should also be communicated to the development team. If appropriate, the clients for this feature should also be included on this correspondence. This approach satisfies both the specification and the design requirements in the case of object-oriented engineering of mathematical software.

Also worth noting in this section is that the end of the design and early part of the implementation phases is the ideal time to write the first set of unit tests. These tests can be used to confirm the interface structure and prepare for incremental implementation testing.

## 5.5 Implementation

Assuming that the above approach is used to define a documented header file, implementation involves implementing the methods as specified and developing test code to verify the correctness of the implementation. Implementing new capabilities should never take place in a release branch. Changes to release branches should be limited to fixing broken code and related activities. For example, clarifying vague or incorrect documentation and making changes necessary to port to a new platform.

## 5.6 Integration

Prior to checking any new code into the Trilinos CVS repository, all regression tests for any affected package should be run by the developer. Also, the developer should make a special point of confirming that nightly automatic regression tests ran successfully. Confirmation is easily done by visiting the archives for the `package-regression@software.sandia.gov` mail lists. The archives contain the results of the regression test runs for all Trilinos packages. A developer will also see the results of the regression tests run by a particular script if their email address is explicitly listed in that test script.

## 5.7 Maintenance

Trilinos provides a number of tools to facilitate the ongoing development and support of packages. CVS, Bugzilla, Mailman and the regression test harness are the most important ones.

1. **CVS:** The Trilinos CVS repository is the most important tool for proper maintenance. With each Trilinos release, a release branch of the CVS repository is created that allows simultaneous, independent development on the main CVS branch and incremental feature development and bugfixes on the release branch. Prior to a release, each package is encouraged to stabilize its source on the main development branch, or create a tagged version of the package that is stabilized. At that point, the main Trilinos development branch will be tagged and branched using the versions of all packages as specified by the package leaders. After the Trilinos tag and branch is complete, package developers are encouraged to continue large scale active development on the main development branch, respond to bugfixes in the release branch and merge bugfixes from the release branch into the development branch.

Further discussion on these topics is in Appendix A. For a full discussion of advanced CVS topics, we recommend the book by Fogel and Bar [5].

2. **Bugzilla:** The Trilinos Bugzilla site allows users and developers to submit issues against a package. Issues may be submitted against the following components of a package:
  - (a) Configuration and Building.
  - (b) Documentation and Web Pages.
  - (c) Examples.
  - (d) Source code.
  - (e) Tests.

Issues may range from a critical source code bug to a new feature request. When an issue is submitted, the owner, submitter and any party that was explicitly listed will be notified upon submission of the issue, and when any subsequent update is made to the issue.

3. **Mail lists:** Trilinos mail lists also support ongoing maintenance by allowing developers to subscribe to the package checkins list. When subscribed to this list, all CVS commits made for the package will be sent in email form to the checkins list, and subscribers will see exactly what has changed. The other package lists mentioned in Section 3.7 also facilitate ongoing communication between developers, users and clients.
4. **Test Harness:** The Trilinos test harness simplifies code maintenance in two ways. First, code is tested on a nightly basis on various platforms to help maintain portability. Second, developers can execute a suite of tests via the “runtests” make targets before committing changes. Developers can easily contribute to the coverage of the test harness. For more information about the test harness see Section 3.3.

## 5.8 Retirement/Replacement

To the extent possible, checkins to a release branch should not force interface changes for users. Even on the development branch, users should be notified (via the package-users mail list) that checkins are about to happen that would require an interface change to user code.

In general, we will be very slow to omit a package, or version of a package, that is in use, unless there is equivalent interface and functionality support from a new package.

## 6 Petra, Thyra, and Teuchos: Special Trilinos Libraries

In order to understand what Trilinos provides beyond the software engineering tools and the contributions of each Trilinos package, we briefly discuss two special parts of Trilinos: Petra and Thyra. Thyra provides a common abstract application programmer interface (API) for other Trilinos packages and Petra provides a common concrete implementation of basic classes used by most Trilinos packages. In addition, we will introduce Teuchos, the Trilinos package that provides a number of useful tools to many Trilinos packages.

### 6.1 Petra

Matrices, vectors and graphs are basic objects used in most solver algorithms. Most Trilinos packages interact with these kinds of objects via abstract interfaces that allow a package to define what services and behaviors are expected from the objects, without enforcing a specific implementation. This facilitates integration of a Trilinos package into almost any existing application.

However, in order to use these packages, some concrete implementation of matrix and vector classes must be selected. Petra is an object model for parallel, distributed-memory, object-oriented matrix and vector classes. Presently there are three Petra libraries: Epetra, Jpetra and Tpetra. Of the three, Epetra is the most mature and the one presently used in production computing settings. Epetra is a collection of concrete classes that supports the construction and use of vectors, sparse graphs, and dense and sparse matrices. It provides serial, parallel and distributed memory capabilities. It uses the BLAS and LAPACK where possible, and as a result has good performance characteristics.

In addition to providing easy construction and use of matrices, vectors and graphs in a parallel distributed memory environment, one of the most important aspects of Epetra is that every other Trilinos package can accept user data as Epetra objects. This facilitates the use of multiple Trilinos packages in combination. For example, Ipack objects can be used as preconditioners for AztecOO, as can ML or Amesos objects. Users can also use Trilinos packages in sequence such as solving linear and eigen problems with the same matrix.

In summary, Epetra provides a common foundation for all other Trilinos packages while retaining an open architecture that allows any package to be used independently. Epetra also supports light-weight copying of user data, allowing easy inter-

operability with other package such as PETSc.

## 6.2 Thyra

Many different algorithms are available to solve any given numerical problem. For example, there are many algorithms for solving a system of linear equations, and many solver packages are available to solve linear systems. Which package is appropriate is a function of many details about the problem being solved and the platform on which application is being run. However, even though there are many different solvers, conceptually, from an abstract view, these solvers are providing a similar capability, and it is advantageous to utilize this abstract view. Thyra is a collection of abstract classes that provides an application programmer interface (API) to perform the most common solver operations. It can provide a single interface to many different solvers and has powerful compositional mechanisms that support the light-weight construction of composite objects from a set of existing objects. As a result, Thyra users gain easy access to many solvers and can bring multiple solvers to bear on a single problem.

Thyra contains a set of core classes that are considered essential to almost any abstract linear algebra interface including Vector, MultiVector, LinearOp and VectorSpace. Capabilities build on top of the Thyra core classes include overloaded, block and composite operators, all of which support powerful abstraction capabilities.

Thyra is important because it allows interfacing and sophisticated use of numerical linear algebra objects without requiring the user or application to commit to any particular concrete linear algebra library. This approach allows us to leverage the investment in sophisticated abstract numerical algorithms across many concrete linear algebra libraries and gives application developers a single API that provides access to many solver packages.

## 6.3 Teuchos

Teuchos provides a suite of common tools for Trilinos for developers to use. These tools include select BLAS and LAPACK wrappers, smart pointers, parameter lists, and XML parsers. Most Trilinos packages have a dependence on the Teuchos library.

## 7 Integrating a Package into Trilinos

Before beginning to add a new package to Trilinos, permission must be granted by the Trilinos Project leader. Sections 2.8, 4, and 5 discuss different aspects of adding a package to Trilinos. These sections cover how to add files to a package, what is required of a package, and how these requirements could be met, respectively. This section will address the steps that can be taken to integrate a new package into the Trilinos configure and build system. It is assumed throughout that a process like the one in Section 2.8 has already been used to add all of the source files for the new package to the CVS repository. This section also assumes that the recommended directory structure for Trilinos packages (src, test, example, and doc subdirectories) is being used.

Some useful terminology for this section is listed in Table 3.

Term:	Definition:
autotool'ed package	A package that can be configured and built using Autotools.
new package	A package to be added to the Trilinos Autotools configure and build process.
new_package	A Trilinos package found in <code>Trilinos/packages/new_package</code> . Serves as a guide for adding new packages to the Trilinos Autotools configure and build processes.
np	The generic name for a new package that is used throughout this section. The source code for np is assumed to be located in <code>Trilinos/packages/np</code> .

**Table 3.** Useful Terminology for Section 7.

There are seven general steps that need to be followed to add a new package to the Trilinos Autotools configure and build system. Note that these steps do not have to be completed in the exact order listed below, nor does a step necessarily have to be completed in its entirety before moving onto the next step. (For example, a portion of a library can be autotool'ed and tested before work begins on the rest of the library.) The seven steps are listed below.

1. Request services that are provided to Trilinos packages.

See Section 3 for more information about services available to Trilinos packages such as mail lists and issue tracking. To request these services, contact the Trilinos Team Leader. It is helpful to complete this step early on because issue tracking can begin and mail lists can preserve initial commit comments.

2. Write the non-generated files necessary for Autoconf and Automake.

Examples of all of the new, non-generated Autoconf and Automake files required to add a packages to the Trilinos Autotools configure and build processes are located within the `new_package` directory structure. Most of these files will need to be customized for each new package. See the individual example files for more details. Instructions for customizing are listed behind

`#np#`

in the following files:

```
Trilinos/configure.ac
Trilinos/packages/Makefile.am
Trilinos/packages/new_package/Makefile.am
Trilinos/packages/new_package/configure.ac
Trilinos/packages/new_package/src/Makefile.am
Trilinos/packages/new_package/example/Makefile.am
Trilinos/packages/new_package/example/example1/Makefile.am
Trilinos/packages/new_package/test/Makefile.am
Trilinos/packages/new_package/test/test1/Makefile.am
```

Note that not all of these files are used in building `new_package`, as it is a very simple package.

### 3. Create generated Autoconf and Automake files.

**IMPORTANT:** Before starting this phase, please see `Trilinos/config/AutotoolsVersionInfo` , and obtain the correct version of both Autoconf and Automake.

If `Trilinos/configure.ac` or `Trilinos/packages/Makefile.am` have been changed (these files will have changed if `np` is being added to the Trilinos level configure and build system), run

**Command:** `./bootstrap`

in the `Trilinos` directory. If any Autotools files at the `Trilinos/packages/np` level or lower have been changed, run

**Command:** `./bootstrap`

in the

**Command:** `Trilinos/packages/np`

directory.

The bootstrap commands should complete without any errors. Note that while bootstrapping at the package level often completes in less than thirty seconds, bootstrapping at the Trilinos level can take more than an hour.

#### 4. Test and debug Autoconf and Automake files.

Run `configure` with the appropriate options in a clean build directory. Once the configure process completes successfully, type

**Command:** `make everything`

to build the configured packages, and the associated tests and examples. For information about configuring and building Trilinos, see Section 2. Testing and debugging can become a long iterative process. Below are some tips for improving efficiency in this step.

- Consider disabling all of the packages that do not need to be built to complete the current task. When debugging the `configure.ac` script for `np`, consider configuring at the `np` level, as libraries that `np` is dependent on are not needed at the configure stage. (To do this, make sure to point at the `np` configure script from the build directory.)
- The `echo` command can be used in `configure.ac` to print out the value of variables for debugging purposes.
- The `AC_CHECK_FILE` macro can be used in `configure.ac` to check for the existence of a particular file or directory.
- Do not run `configure` in the source tree, always use a separate build tree.

#### 5. Add all Autoconf and Automake files to the Trilinos CVS repository.

The `bootstrap` command will generate files necessary for the configure and build processes. These files must be added to the repository because users are not required to have Autoconf or Automake. See Section 2.8 for information regarding how to add files to the repository.

When the new files have been committed, a good sanity check is to checkout a new copy of Trilinos and attempt to configure and build. This will catch any files that have not been added.

#### 6. Add regression tests to the Trilinos test harness.

For more information about this step, see Section 3.3. It is not uncommon for packages to contribute tests to the test harness some time after the package has been added to Trilinos. However, it is important that all packages can be tested thoroughly via the test harness.

## 7. Perform tests.

Build with and without package options and run all tests associated with np on a variety of platforms. Make sure that dependencies have been properly established in the Autotools system so that users cannot disable packages that np is dependent on. Finally, use the “runtests” make targets to ensure that all Trilinos packages still build properly. For more information about how to use the runttests make targets, see the end of Section 2.8.

## 7.1 Improving Portability

Trilinos packages build on a wide variety of platforms. Below are a few tips for improving portability using mechanisms associated with Autoconf.

- Make sure that np\_config.h is included in all your source files, directly or indirectly. The value of any package-specific options are pulled from this file. It is usually best to include it through a ConfigDefs file; read on for more information.
- Include np\_config.h before (usually indirectly) including (AnyOtherPackage)\_ConfigDefs.h, or the Autotools PACKAGE macros will be improperly defined.
- Consider creating a file analogous to Epetra\_ConfigDefs.h. This file takes care of various issues such as setting the value of package-specific options and defining macros used in the package. The ConfigDefs file is usually included in all other source files. Don't forget to include np\_config.h in the ConfigDefs file because that is the file that contains the results of the checks performed during the configure stage.

## 8 Interoperability Status for Existing Trilinos Packages

Most Trilinos packages depend on one or more other Trilinos packages. For example, a large number of packages depend on Teuchos, the Trilinos tools package. Many packages also “can use” one or more other Trilinos packages, meaning that additional features can be conditionally enabled that depend on the other packages.

A table illustrating package interoperability can be found on the Trilinos website at <http://trilinos.sandia.gov/packages/interoperability.html> .

# References

- [1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM Pub., Philadelphia, PA, second edition, 1995.
- [2] David Day and Michael A. Heroux. Solving complex-valued linear systems via equivalent real formulations. *SIAM J. Sci. Comput.*, 23(2):480–498, 2001.
- [3] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.
- [4] J.J. Dongarra, J. DuCroz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14, 1988.
- [5] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Coriolis Technology Press, Scottsdale, Arizona, 2nd edition, 2001.
- [6] Free Software Foundation. Autoconf Home Page. <http://www.gnu.org/software/autoconf>, 2004.
- [7] Free Software Foundation. Automake Home Page. <http://www.gnu.org/software/automake>, 2004.
- [8] Free Software Foundation. Gnu CVS Home Page. <http://www.gnu.org/software/cvs>, 2004.
- [9] Free Software Foundation. Gnu m4 home page. <http://www.gnu.org/software/m4>, 2004.
- [10] Free Software Foundation. Gnu mailman home page. <http://www.gnu.org/software/mailman/mailman.html>, 2004.
- [11] Free Software Foundation. Libtool Home Page. <http://www.gnu.org/software/libtool>, 2004.
- [12] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.
- [13] Michael A. Heroux. Trilinos home page. <http://trilinos.sandia.gov>, 2004.

- [14] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5, 1979.
- [15] Marzio Sala, Michael A. Heroux, and David M. Day. Trilinos Tutorial. Technical Report SAND2004-2189, Sandia National Laboratories, 2004.
- [16] The Mozilla Organization. Mozilla Bonsai Home Page. <http://www.mozilla.org/bonsai.html>, 2004.
- [17] The Mozilla Organization. Mozilla Bugzilla Home Page. <http://www.mozilla.org/projects/bugzilla>, 2004.
- [18] Ray S. Tuminaro, Michael A. Heroux, Scott. A. Hutchinson, and J. N. Shadid. *Official Aztec User's Guide, Version 2.1*. Sandia National Laboratories, Albuquerque, NM 87185, 1999.
- [19] Dimitri van Heesch. Doxygen home page. <http://www.doxygen.org>, 2004.
- [20] G. Vaughan, B. Elliston, T. Tromeey, and I. Taylor. *Gnu Autoconf, Automake, and Libtool*. New Riders, 2000.

# A Commonly Used CVS Commands

To access the Trilinos CVS repository, an account on software.sandia.gov is required. To request an account, send a note to `trilinos-help@software.sandia.gov`. The following two environment variables must be set to access the repository:

**Command:** `CVSROOT :ext:your_user_name@software.sandia.gov:/space/ CVS`

**Command:** `CVS_RSH ssh`

(Replace “your\_user\_name” with your user name on software.sandia.gov.)

Below is a brief description of the most commonly used CVS commands. For a more complete listing of CVS commands, see the GNU CVS Home Page [8].

- **Checking Out a Working Copy:** To checkout a working copy of the development branch of Trilinos in the current directory from the CVS repository, type

**Command:** `cvs checkout Trilinos`

To checkout a working copy of only one package of Trilinos in the current directory, type

**Command:** `cvs checkout <package_name>`

(Replace “package\_name” with the name of the package.)

To checkout a different branch or a tagged version of Trilinos, type

**Command:** `cvs checkout -r <name_of_branch_or_tag> Trilinos`

- **Updating a Working Copy:** To update after a version has been obtained use the `cvs update` command. First, `cd` to the directory that is to be updated (often the Trilinos root directory). Then type:

**Command:** `cvs -q update -dP`

The “-q” option means “be somewhat quiet”. Try an update without the “-q” to see exactly what the option does.

The “-d” option means to get any new directories. For example, if a new package is added to the repository, but the “-d” option is not used, that new

package will never appear in the working copy. However, the first time that the “-d” option is used, all of the new package directories will be downloaded, and from that time on, all CVS updates will update the directories that were downloaded. It is good practice to include this option for every CVS update.

Finally the “-P” option “prunes” empty directories. This helps to keep the directory structure from getting more cluttered than it needs to. For example, the old “petra” and “tsf” packages were removed from the repository, but the directory structures will remain if this option is not specified. If an empty directory is needed, simply issue one update command without the “-P” and the empty directories will be restored.

- **Viewing Local Changes:** After saving changes to a working copy of a branch of the Trilinos repository, the differences between the most recently obtained version of the edited file(s) and the current local version of the file(s) can be viewed using the following command:

**Command:** `cvsv -q diff`

The “-q” option again means “be somewhat quiet”. Try a diff without the “-q” to see exactly what the option does.

The diff command works recursively, but optionally accepts options that specify specific files and/or directories. For example, to see the diff’s associated only with a file in the current directory named `abc.cpp`, as well as all files located (recursively) in the relative directory `examples`, type

**Command:** `cvsv -q diff abc.cpp examples`

- **Adding Files to and Removing Files From the Repository:**

To add new files `abc.cpp` and `abc.h` to the Trilinos repository, type

**Command:** `cvsv add abc.cpp abc.h`

in the directory where the files are located (in a checked out version of the Trilinos repository). To remove the same files, type

**Command:** `cvsv remove abc.cpp abc.h`

The above commands do not actually add the files to or remove the files from the repository, but simply prepare for the addition or removal of the files. The initial version of the file will be written to the repository using `cvsv commit`.

The `cvsv add` command can also be used to add new directories to the repository. When adding a directory, no subsequent `cvsv commit` is necessary. Directories cannot be removed from the repository using `cvsv remove`.

- **Committing Changes:** Note that the `CVSEEDITOR` environment variable denotes which text editor will be used to edit CVS commit logs. The default value for `CVSEEDITOR` is `vi` on most machines.

Before committing changes, be sure to perform a CVS update. Any conflicts must be resolved before a commit will complete properly. Changes are committed (saved) to the repository using the following command:

**Command:** `cvsc commit`

The above command will recursively check the current directory and all directories that are direct descendants of the current directory for changes. To commit only specific files or directories (specified directories will be checked recursively for changes), append the files and directories to be committed as additional arguments to the commit command. For example, to commit the changes to the file called `file1`, and the directory called `directory1`, type

**Command:** `cvsc commit file1 directory1`

At this point, an editor window will appear with a CVS commit form to fill in and a list of files that are to be added, removed, or modified. The form was created to help developers avoid some of the most common mistakes that are made when committing to the repository, like not bootstrapping after modifying the build system for a package, not using `cvsc add` to add a new `Makefile.in` that was generated when bootstrapping, or not running tests before committing a major change. It is important to enter a description of the changes that are being made to make it easier for people to understand what happened without the need to look at line by line diffs. Also, including any related Bugzilla issue numbers makes it easier to find additional information about the change if necessary.

Completing all the steps in the CVS commit form is not required, but doing so is a good practice to follow, because it significantly reduces the frequency of common mistakes that occur when committing code. Completing the form is very highly recommended for changes made to release branches. For all commits, it is a good practice to make sure that the list of files shown under the checklist makes sense for the changes that are to be made. It is easy to inadvertently commit files or to forget to remove files.

Here are a few helpful tips to use when verifying the list of files to be committed.

- Make sure that all directly modified files appear in the list.
- Make sure that files generated from files that were modified appear in the list. When modifying a `Makefile.am`, the corresponding `Makefile.in` will change also. When modifying `configure.ac`, `configure` will also change,

but some Makefile.in files will also change when using macros such as AC\_SUBST.

- If unsure about whether or not a file should be in the commit list, abort the commit by closing the editor without saving, and choosing `abort` . Then, if `abc.cpp` is in question, type

**Command:** `cvs diff abc.cpp`

Next, look at the diff output and see if it makes sense for the changes that were made.

After completing the CVS commit form, verifying that all of the files that are in the list belong there, and verifying that all files that belong in the list are listed, save the file and exit the CVS editor to commit the changes.

## B The Trilinos Release Process

From a developer's perspective, the Trilinos Release Process consists of six phases.

- A target release date is announced in an email to the Trilinos-Developers mail list.

This message also describes other details of the release such as the (tentative) release version number, which packages are to be released (if this is known at the time the message is sent) and whether the release will be an internal release or a public release. It will be noted in the message that any package development teams that would like a version of their code other than what is in the development branch at the time of the release to be included in the distribution should send the name of a tag that contains the correct version to the Trilinos team member who will be creating the release branch. The details of upcoming releases are usually also discussed during one or more monthly Trilinos Leaders meetings.

- A release branch is created and tagged.

The creation of a release branch is discussed in Section C.

- All package teams participating in an external release (and in some cases, an internal release) complete the appropriate package level release process checklists. During this time, acceptance testing is conducted by some of the largest Trilinos customers, and the release branch is tested on the nightly test harness platforms. Often tests are run on additional platforms as well. Any issues that arise during the testing of the new branch are resolved (either fixed or documented as a failure that is "acceptable" to the associated package development team).

- The Trilinos project leader approves the release.

- The release is announced and made available.

An announcement of the release is sent to the Trilinos-Announce mail list. For outside releases, the release is made available via a compressed tar file from the Trilinos download site.

- Support for the release continues at least until the next release.

More information about the Trilinos release process, and the process checklists used during the release process, can be found on the website:

<http://software.sandia.gov/trilinos/developer/sqp/release/index.html>

<http://software.sandia.gov/trilinos/developer/sqp/checklists/index.html>

## C Creating a New Trilinos (Release) Branch with CVS

A CVS branch can be used to maintain a version of the code that is different than the primary development branch. This section describes how to create a Trilinos release branch, but the steps can easily be generalized to apply to other types of branches. For example, developers commonly want to attempt an experimental reworking of a piece of code. If there is a good chance that this code will not be stable for some time or if the code might not ever become a part of the development branch (ie the experiment fails), the developer might want to create a separate branch. If the experiment is a success, the changes can always be merged back into the development branch.

There are eight steps in creating a Trilinos release branch:

### 1. Checkout Trilinos

It is a good idea to start with a fresh copy of Trilinos. If an old copy is used, various problems can occur. Most of these problems are beyond the scope of this brief tutorial, but be advised that it will not work to make changes to a local copy of Trilinos and commit those changes directly to a branch. To create a branch for changes that have already been made to a local copy, A developer must checkout a new working copy, create the branch, copy the changes over, and then commit the changes to the branch.

To checkout a copy of the development branch, type

**Command:** `cvs checkout Trilinos`

To checkout a different branch or a tagged version of Trilinos, type

**Command:** `cvs checkout -r <name_of_branch_or_tag> Trilinos`

### 2. Update packages with specific tags, if necessary

Package developers have the opportunity to include a special tagged version of their package in a release instead of the current version in the development or release branch that is being branched off of for the release. For any packages that have such a tag to use, cd to Trilinos/packages/(package\_name) and type

**Command:** `cvs update -dr <name_of_tag_submitted>`

### 3. Tag the working copy

The existing branch is now tagged to mark the point where the new branch will diverge. To do this, cd to the top Trilinos directory and type

**Command:** `cvs tag <tag_name>`

For a release branch, `tag_name` should be `root-of-trilinos-release-XYZ-branch`, where XYZ is the release number with the component numbers separated by hyphens instead of periods. For example for release 3.1.13, XYZ should be 3-1-13.

### 4. Create a branch

To create a branch, remain in the top Trilinos directory and type

**Command:** `cvs tag -b <branch_name>`

For the Trilinos release version XYZ mentioned above, the `branch_name` should be `trilinos-release-XYZ-branch`.

### 5. Update to convert the working copy to the new branch

The act of creating a new branch does not convert the existing working copy to a copy of that branch. To make this conversion, remain in the top Trilinos directory and type

**Command:** `cvs update -r <branch_name>`

To verify that the preceding command was successful, type

**Command:** `cvs status configure.ac`

The value of the “Sticky Tag:” field should begin with “branch\_name”.

### 6. Update the Trilinos release version number

When creating a new Trilinos release branch, the Trilinos version number needs to be incremented in the `Trilinos/configure.ac` file. The version number of individual packages can also be updated at this time. Note that not all packages will have the same version number as Trilinos. Package version numbers do not have to be updated at this time. Permission must be obtained from package developers before the version number for a package can be updated. To update a version number, open the `configure.ac` file and search and replace the old version number with the new version number. The number will need to be updated in one or two places. Next, remain in the top level Trilinos directory (or the top level directory of the package) and type

**Command:** `./bootstrap`

to update the generated Autotools files to reflect the new version number. Finally, follow the commit process listed in Section A to commit the changes. Remember to update before committing so that generated files are not committed just because the timestamp changed. In the commit message, note that `branch_name` is listed.

#### 7. Tag the new branch

It is a good idea to tag the branch at this point so that the initial state of the branch is easily retrievable. It is required that the initial state of a release branch is tagged. To perform this step, type

**Command:** `cvstag <initial_tag_name>`

For the Trilinos release version XYZ mentioned above, the `initial_tag_name` should be `initial-trilinos-release-XYZ-branch`.

#### 8. Test the new branch

It is a good idea to test the new branch to help make sure that the above process has been completed properly. As a part of the general release process, many tests need to be run. This step simply refers to running a few simple sanity checks (for example, configure and build Trilinos and execute one of the “runtests” make targets or a couple of individual regression tests) to work out any obvious problems before the branch is subjected to the more complete test suite.