D3.6.2 – Domain Specific Language (DSL) for expressing parallel auto-tuning

WP3: Development environment

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M30
Submission date:	31/03/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organization	KTH
Version:	1.0
Status	Final
Author(s):	Harvey Richardson (CRAY UK)
Reviewer(s)	Stephano Markidis (KTH), Bastian Koller (USTUTT)

Dissemination level	
PU	*PU – Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	26/2/2014	Base version from D3.6.1	Harvey Richardson (CRAY UK)
0.2	1/3/2014	Incorporate mockup developments	Harvey Richardson (CRAY UK)
0.3	7/3/2014	Scenarios and DSL source updates	Harvey Richardson (CRAY UK)
0.31	7/3/2014	Version for internal WP review	Harvey Richardson (CRAY UK)
0.4	20/3/2014	Address comments from reviews	Harvey Richardson (CRAY UK)
1.0	31/03/2014	Final version of the deliverable	Harvey Richardson (CRAY UK)

Table of Contents

Index of Figures

Index of Tables

1 Executive Summary

This document describes a domain-specific language (DSL) that serves as the central component of an autotuning framework for the tuning of parallel applications. We describe what features the DSL was designed to provide, how it fits within a wider autotuning framework and outline the initial implementation.

Our initial approach was to start from scratch without detailed reference to, or consideration of, existing autotuning technology but starting from the basis of a specific set of requirements we considered important. One reason for this is that the remit of the CRESTA project is to define a distinct European approach. We also need to be sure that we can be in control of (or define) an environment that will support particular aspects of tuning parallel applications.

The following sections introduce and outline the scope and objectives of WP3 task 3.2.1 giving an overview of what the actual deliverable addresses. We then consider the objectives for the DSL and move on to describing the specification and how it would be implemented within an autotuning framework.

This document is an update of the initial DSL specification (D3.6.1). Since that work we have incorporated changes based on experience in two areas:

1) We implemented a mockup implementation as a platform for testing the practicality of the DSL and made it available to CRESTA partners
2) The mockup script was used in the tuning of NekBone (D3.5.2) and new feature and implementation requests came from this work.

2 Introduction

The CRESTA WP3 work package contains a task (3.2) on Compiler and Runtime Environments of which subtask (3.2.1) deals with autotuning, a technology that can address the inherent complexity of the latest and future computer architectures. In the context of this project, autotuning is the process by which an application may be optimised for a target platform by making automated optimal choices of how the application is built and deployed. Tuning choices can be made that target algorithms, source code (optional branches, data flow, loop transformations etc.), compilation and application launch. We can express both the tuning choices and controls for an autotuning framework via the use of a domain specific language (DSL) and this is the focus of this specification.

The DSL we are developing can expose these choices within an application and primarily concerns source mark-up; in particular we aim to address parallel tuning aspects and interoperability with existing and future autotuning technology.

In subsequent sections we note how the work on the DSL fits in the wider context of the project and note in more detail the requirements and implementation choices we have made.

2.1 Purpose

The purpose of this document is to provide an update to the initial DSL specification.

2.2 Related work

One of the project partners (Cray) has extensive experience of production autotuning, specifically for the generation of optimized scientific subroutine libraries[1]. This was, in part, motivation for the new unrelated work under CRESTA to research more general library tuning and whole-application tuning of parallel applications which we present here.

The DSL described here would exist in a more general autotuning framework, some components of which would be studied or provided within the CRESTA project. Work on adaptive runtime and compiler autotuning are separate tasks in CRESTA. The DSL will be subsequently developed in conjunction with the CRESTA co-design applications and be informed by those application requirements in addition to other project activities. This will happen later in the project.

2.3 Scope and objectives

The DSL is primarily concerned with markup of tuning choices. These may be either exposed by the programmer/user (for example algorithm choices, source optimization choices, library choices, runtime choices) or may be implementation choices of higher level programming "constructs" (for example stencils, communication patterns etc.).

The DSL principally targets the application developer and possibly those concerned with application optimization. Runtime features may be of interest to the application user. Use of the DSL should facilitate exploration of the application tuning space to make it easier to produce an application optimized for a particular platform.

DSL statements can appear in various places: in source files or files associated with source or in a configuration file describing the overall tuning process. The particular aspects which would warrant placement in a global configuration file are the following:

1. Runtime choices (for example how many threads for a mixed-mode application)
2. Dependency information between tuning parameters
3. Convenience grouping of tuning parameters
4. Linking or disambiguating tuning parameters defined in multiple places in order to fix their scope
5. Compiler options and build choices

6. Use of external tools/components for example compiler autotuners, parameter optimization, machine learning matcher
7. Interfaces to above

Note that there is some overlap and you could embed DSL in application code that could alternatively be placed in a configuration file. In addition, a global configuration could augment DSL in source files, for example by adding additional scoping. For subsequent sections we will use the term XDSL to indicate DSL that would more naturally be placed in an external configuration file.

We will also consider control information that would be required in order to perform an end-to-end autotuning session, this allows us to understand how the DSL would fit in a wider context and would give us something that we can mock up at the end of the project. We have also defined some components as external modules (for example a parameter optimizer) because this gives us flexibility to build simple reduced-capability implementations or interface to other software that can provide the required functionality.

Note that a basic autotuning infrastructure could be useful in general code development and testing, allowing a simple mechanism for the developer to explore choices even if an intelligent tuning framework is not required.

One of our major goals is to provide something that is easy to use with minimal application changes. In practice this means that a new user should be able to use the framework very quickly without installing a complicated software stack or making significant changes to how the application is built and run. As a result we started by defining how a tuning session is accomplished and how an application is built, run and optimized. Only after doing this did we consider the precise details of DSL embedded in source.

Our aim is to support, at a minimum, applications that are written in C, C++ and Fortran and use OpenMP and MPI in addition to selected single-sided or PGAS programming models.

Autotuning is a wide area of research with many aspects[2], in particular a lot of work has been done on autotuning compiler infrastructure; our focus is more on parallel application tuning in the context of a general framework.

2.4 Glossary of Acronyms

D	Deliverable
DSL	Domain Specific Language
EC	European Commission
ML	Machine Learning
PGAS	Partitioned Global Address Space
PM	Project Manager/ Project Month
WP	Work Package
XDSL	External DSL
XML	Extensible Markup Language

3 Requirements

In this section we outline the requirements and objectives we had in designing the DSL and begin by considering the overall tuning framework.

The DSL is a component of an autotuning framework and at the highest level we assume that this framework can optimize an application over a set of tuning parameters. Some parameters we term *scenario characterization parameters* and these may for example, map to input parameters relating to problem size. This is illustrated in Figure 1 below where we have two scenario parameters S1 and S2 and show five scenarios.

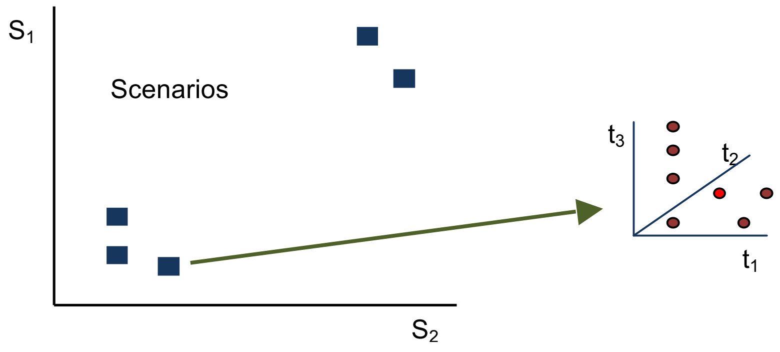

Figure 1: Scenario and Tuning spaces

For each scenario, we aim to pick the best values for a set of tuning parameters (in the figure: t_1, t_2, and t_3). The tuning parameters will relate to build and runtime optimization choices which we can choose to give for example the best runtime. At its simplest, the autotuner framework can optimize over the tuning parameters, at the most complex it can build routines and applications choosing the best tuning parameters for a set of scenario characterization parameters.

In this section we consider which features the DSL needs to support and we categorize them as follows:

- Overall tuning configuration information
- Details of tuning parameters and relationships
- Parallel autotuning features
  - Stencils
  - Data movement primitives
  - Process placement
- Build information and control
- Runtime information and control
- Interfaces to tools/components
  - Parameter optimization
  - Machine-learning (ML) matcher interface
  - Library tuners
  - Serial compilation and Compiler-based tools (profile feedback and autotuning)

For now we will not consider the syntax and placement details of DSL constructs.

3.1 Overall tuning configuration

We need a way to control an overall tuning session, describe the objectives for a tuning run, the parameter space and build and run details. It is an implementation decision as to which aspects of the tuning configuration are described centrally or in application files. The overall configuration should describe the following:

3.1.1 Tuning Control

This is where we describe what an autotuning run should do. There are likely two scenarios:

1. Only tune across tuning parameters picking the best
2. Run tuning (over tuning parameters) for a set of scenario characterization parameters and optionally post-process results with an ML system in order to build a tuned library or application that can cater for a range of scenario parameters.

For option 2 we need to define or obtain the set of scenario parameters.

3.1.2 Tuning target

We need to be able to describe optimization for a target performance metric which could be minimum execution time or be related to some output from the application. This should be flexible enough to support something like minimizing power consumption.

3.1.3 Tuning scope

It is very likely that we would want to tune for some subset of tuning parameters or code base so some parameter grouping method should be provided to enable this.

3.1.4 Previous state

State from previous tuning sessions may be available; this section would describe how this could be done. In order to interface to an external optimizer it would most likely be a requirement to manage state.

3.1.5 Logfiles

The tuning process should produce a log of progress to specified output streams/files.

3.2 Tuning Parameters and relationships

Various aspects of autotuning require that we set parameters within some range. This enables us to optimize over fixed choices of code paths, parallel decompositions, optimizations (for example blocking and unrolling) etc. To do so we need to define that a parameter is a tuning parameter and describe any bounds and constraints.

The parameter definition should support

- Typing, to include integer, real, character and Boolean
- Definition of a range of values or a specified set of values
- Any constraints between parameters (for example we may have parameters N and M which have to satisfy some relationship M*N = P)
- A way to indicate that particular parameter choices are not allowed; this could happen if machine-learning software was generating parameter values.

Another requirement which was raised by users of the mockup implementation was the ability to have parameters which were only valid for particular values of another parameter. For example if a tuning parameter is used to choose an OpenACC accelerated loop then other parameters choosing loop clauses become valid.

At a higher level we need the ability to group parameters for the following reasons:

1. So we can describe which parameters should be treated as dependent for tuning purposes. All parameters are likely to be dependent to some degree but to cut down the search space and to aid understanding it will be useful to have the ability to declare parameters as independent.
2. So we can tune over a subset of parameters.

3.3 Parallel autotuning features

To some extent traditional serial autotuning techniques can be applied to the parallel domain. We can use any features we have to choose amongst implementation choices at the routine or block level to choose different ways to implement parallel operations. However there is scope to move beyond this and more directly address parallel aspects of an application, for example to target standard patterns (stencils for example) or address data movement. These two aspects are where the DSL should have most utility as we expect to move beyond mere tuning choice parameterization. The other topic is that of decomposition and runtime process and thread distribution choices. These topics are considered in this subsection.

3.3.1 Stencil computation

A stencil computation is a core component of some algorithms and comprises a distributed calculation on a grid. On a local level this typically equates to an iteration encompassing data movement (to move data from other processors) and a local computation partly involving data that has been moved. The classic examples are simple iterative schemes to solve for example the Laplace equation.

Stencils are described at a very high level and require an appropriate infrastructure to manage the decomposition and communication. Our approach will be to tackle stencils via aggregate data movement primitives described below. It would be a bonus to integrate somehow with software that can optimize stencil computations.

3.3.2 Data movement primitives

The idea here is to think about a particular pattern of computation that we think is lower-level than for example the stencil but will cover more real application usage. These primitives could, for example, address the data movement (halo exchange) that we see in stencil computation but be more general. One approach we want to investigate is to describe locations in the code where data could be communicated and locations where that data needs to be available. The autotuning infrastructure can use various choices on how that movement can take place to optimize the communication (likely to be the most expensive part). Choices that would need to be made would be the use of non-blocking communications, use of buffers and synchronization.

3.3.3 Process placement

By process placement we mean the ability to vary, for example, the number of processes and threads in a multi-process programming model. Also the mapping of processes to the hardware is something that can be varied. For GPU models we can vary the decomposition to the GPU (grid dimensions, number of threads etc.).

This is more of a runtime concern for some programming models so would be part of a runtime configuration.

3.4 Build description and control

We need to allow an autotuner to control the build process. There are various aspects to this.

We need to be able to describe how parameters defined in the global configuration can made available to the build process. For example one scenario is that they are passed into a Makefile as variables that appear as –D options in compilations. There should be enough flexibility that a script can be provided to enable this integration for a more general build infrastructure.

There should be a way to name or tag any output binary with a tuning build for a set of parameters. Note also that we need to define which parameters relate to a new build of a binary as we will have to rebuild if we vary those parameters. Requiring a rebuild for any change in parameters is not acceptable.

We should describe how to call a clean script provided for the application build.

We also need to be able to support compiler flag optimization; this is where we explore a set of compiler options applied to many or some source files.

In order to build a version of the application for a specific set of tuning parameters defined in DSL embedded in source we need to either parse the DSL with a DSL-capable compiler or process the source files appropriately for a specific set of tuning parameters and pass the resulting source into the build.

Our approach needs to be flexible and not mandate new and invasive build procedures.

3.5 Runtime information and control

We need a way to describe how to run the application.

Some parameters may map to an input file for the application. We should provide a standard format file for this and also allow a script to create the expected input for the application.

One part of this is a correctness check where we can optionally determine that a run was successful; there would be no point in optimizing for the fastest incorrect run!

3.6 Interfaces to tools and components

In order to provide a general autotuning framework there are some components that are essential or which it would be useful to interoperate with. This means we can describe a modular structure which should be more attractive to potential users.

3.6.1 Parameter Optimization

This is a core component of an autotuner and can be as simple as an exhaustive search of the parameter space, or as complex as a full machine-learning environment.

There are two running scenarios that should be supported as outlined below:

Parameter space optimization

This is where the tuning process finds the optimal parameters from the parameter space.

Scenario exploration

This is a variation on the above where some parameters are characteristics of the application run, for example problem sizes. The tuner would explore the remaining parameter space and a machine learning system could then build a model to find the optimal set of tunable parameters from the characterization parameters. Note that there are compilation systems that work like this (the features are program fragments).

To support these two scenarios we need the ability to run across the whole parameter space or to optimize all or part of the parameter space.

Provision should be made to call a plugin tuner which would accept the parameter definitions and control the tuning process. This would have to be some sort of "delegated control interface" such that the tuner passes back information on which set of parameters to use for the next run as it explores the parameter space to find the minimum. Some state will have to be maintained by the tuner and possibly by the overall autotuner.

We also need provision to send the results of scenario exploration to an ML system and integrate the results back into the build so that we can either choose an appropriate binary for a set of scenario characterization parameters or optimize individual routines for relevant characterization parameters as noted below in section 3.6.2.

The core configuration should describe the primary metric for optimization and what that optimization is. Additional secondary metrics (for example performance counter data) should be optionally provided to the optimizer.

We should be able to incorporate compiler option tuning into a generic optimizer by having labels as independent tuning parameters.

This process can and will involve many compilations and for some applications this will be time consuming. The design should minimize the number of builds required to explore the tuning space.

3.6.2 Machine learning matcher

In order to produce a library routine or application that is tuned for a range of scenarios this is an important component. The purpose of the "matcher" as we have termed it is to build a mapping from scenario characterization parameters into a model that predicts the best set of tuning parameters for a given choice of scenario characterization parameters. An interface is required that sends the tuning experiment results to the matcher and accepts back the model in a form that can be incorporated into a library interface or some sort of runtime launch. At its most complex, the matcher could be a machine learning[3][4] system using decision trees or some other technique.

3.6.3 Library tuners

One technology that is relatively mature is the library tuner that can produce optimal code for a library routine for a given architecture and set of input parameters. We should support this aspect of tuning in the following ways:

1. Allow grouping of parameters and their independent tuning for some source subset (the library routine).

Library tuning fits into the overall architecture as follows. The DSL describes the tuning location, the scenario characterization parameters and the tuning parameters at the call site. The framework explores the scenario parameter space and passes the result to the ML system. The ML system then creates an interface routine that maps the scenario parameters to the optimal choice of optimization parameters. Note that for this to work at runtime the optimization parameters must be runtime parameters or we need a mechanism to create multiple versions of the library routine (which might be possible).

It is likely that library tuning will be a separate component of whole-application tuning and could potentially be used to tune implementation of parallel data movement routines (like MPI collectives).

3.6.4 Serial compilation and compiler-based tools

A crucial aspect of application performance is the optimal compilation of source code into machine instructions. Performance-critical code sections typically involve loop nests and require the compiler to apply transformations such as loop unrolling and blocking for cache along with decisions concerning register use and instruction scheduling. Some specific tunable aspects are unroll length, prefetch length, prefetch depth, loop order and blocking factors.

We need to decide to what extent we want to support this aspect of tuning and we can consider various capabilities:

1. We support the programmer in manually implementing tuning by using the general framework to tune for parameters that control loop transformations (our framework should at least provide this but it puts the onus very much on the programmer to do all the work.
2. We provide DSL to allow the programmer to control loop transformations and arrange for these to be mapped to implementation-specific compiler directives or we generate source to manually implement the options.

3. We have a compilation system that can operate at the IR level and accept transformation instructions in terms of the IR or alternatively can output its tuning choices in such a way that we can mandate it picks one of those choices.
4. We treat compiler-based tuning as something separate.

Our initial approach is to support options 1, 3b and 4 and work towards supporting option 2 in the course of the project if we can add any value above existing compiler autotuning projects.

4 Implementation and DSL specification

In this section we consider which choices we have to make or have already made (or have deferred) in implementing a DSL/XDSL that meets the requirements previously outlined.

The most high-level choices to be made are the style of the DSL and the implementation of the global configuration; various choices are possible[5]. The most flexible approach is going to be a DSL that is a first class addition to the language in use (primarily C and Fortran for Exascale applications). An alternative is to use directives within application source.

Our initial approach was to define a DSL with a view to an initial mockup implementation at the end of the project that would parse source and generate compilable source that did not require extra compiler infrastructure beyond standard language support. As a result, this document outlines initial syntax in the form of directives/pragmas. We also designed the DSL to be minimally invasive such that as far as possible only DSL directives need be added to an application and the source is untouched and remains valid compilable source.

4.1 Software Architecture

The most appropriate architecture would seem to be to have an overall controlling application which reads the global configuration and controls the whole tuning process. This application would be responsible for building and running the application and interfacing with the optimizer (a separate component).

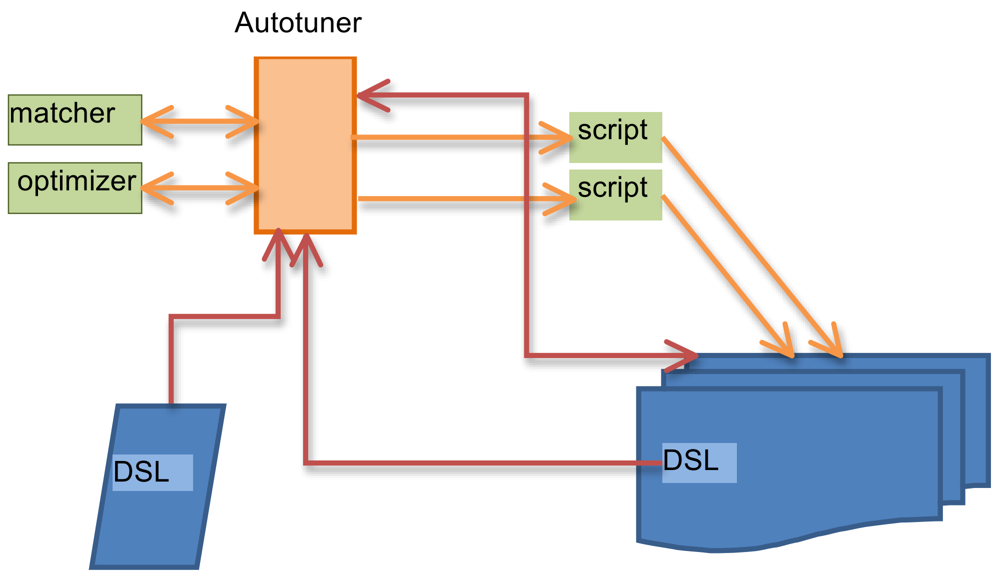

Figure 2: Tuning Architecture – high level view

The components shown in Figure 2 implements the scenarios outlined in the requirements section 3.1.1 as follows. The autotuner component controls the whole process and starts by reading DSL in application source files and a global configuration file. The autotuner will either scan all application source or call a special script which would cause a supplied preprocess script to be run over the application source. The autotuner would then decide if a simple tuning run or scenario exploration run was required. To accomplish a tuning run the source is appropriately preprocessed (or just compiled) and an optimization process organized. Build and run scripts manage the build and run process and the optimization component can help streamline the search for the best tuning parameters. If a scenario exploration run is undertaken then the matcher is called to build a model that maps scenario parameters to the appropriate tuning parameters. The output from this could be used to either build optimal versions of specific routines or to choose from alternative application binaries. How this process is controlled and implemented is outlined in subsequent sections.

We have implemented a mockup of the autotuner and optimizer components which implements some of the DSL features. It can parse the global configuration and run an exhaustive-search tuning session. We expect to extend this mockup further in the hope that we can apply it in practice to more application scenarios during the rest of the project. NOTE: Although we may define a compiler-parsed DSL by project completion we are not providing compiler infrastructure to compile it.

4.2 An introduction to the source DSL

In this section we explain the most basic features of DSL that can appear in application source code. We outline here only the features that are required to inject parameters from a tuning run into application source. Note that it is possible to tune without ever using DSL in the application source. Subsequent sections introduce more specific DSL syntax for other purposes.

Our DSL is implemented as compiler directives and the two forms accepted are

```
!tune$ directive [clause] [clause]…
#pragma tune directive [clause] [clause] …
```

The basic form of the directives is

```
[if-clause] [begin | end] tune-directive [clause…]
```

The basic set of directives is:

```
import param-name-list
inject:text
inject_r:text
inject_r$:text
skip [fill]
replace
```

The import directive makes a tuning parameter available for use in other directives. The inject directives cause the supplied text to be inserted into the program; the inject_r version replaces any occurrence of the parameter in the text with the value at the point the file was parsed. The inject_r$ version only replaces text of the form $param or ${param}, this is the replacement style used in the tuning configuration file.

Examples might be:

```
!tune$ import NB
!tune$ inject_r:  blocksize=NB
```

The replace directive is used to inject parameters into enclosed source and the use of begin and end is required. The skip directive also requires begin and end to define a section of code which will be removed (fill is optional and if present the source will be replaced by blank lines which may be useful to preserve line numbers for comparison purposes).

Note that replacements should not take place inside quoted strings.

The if clause takes the form

```
if (lexpr)
```

where lexpr is a logical expression which may reference any imported tuning parameters. An implementation should support at least the following operators (!=,==, <,>,>=,<=,%,+,-,*,/) where % is modulus. Should the expression evaluate to false then the tuning directive is ignored.

4.3 Overall tuning configuration

The overall tuning configuration is described in a configuration file with sections for the various parts of the configuration (build, run etc.). This file will be defined in two styles, text and XML. The XML form will be described at a later stage but would closely follow the text syntax described here.

Syntax (case-sensitive text):

```
begin configuration
 configuration-entity
end configuration
```

Allowable configuration entries are available to set the tuning target, tuning scope, previous statefile and logfile locations.

The configuration file may contain comments which are lines that are either empty, all whitespace or start with a "!" character as the first non-whitespace character.

In general whitespace within the configuration file is not significant, it would be sensible to use indentation to increase readability as per some of the examples provided later.

4.3.1 Tuning target and scope

This section describes the tuning objective for the optimization of parameters and how target metrics are obtained from tan application run.

```
begin tune
 tune-entity
…
end tune
```

where the tune entity can be:

```
mode: tune | scenarios
target: min|max
scope: param-list | collection
metric-source: file | stdout | runtime
metric-placement: lastregexp| validation
```

If the mode is set to "tune" then the autotuner will optimize over the tune-scope parameters. If the mode is set to scenarios then the autotuner will perform a scenario exploration run as described in the next subsection.

The target entry defines how the optimizer should optimize the runtime metric. The source of that metric can be from a named file, standard out or the runtime. The following additional statement defines the location of the metric file.

```
postrun-metric-file: filename
```

The metric can also be obtained from the validation output (see section 4.8).

It the metric is not obtained from the validation output then a regular expression must be supplied as follows:

```
metric-regexp: regexp
```

The supplied (ruby) regular expression extracts the metric from a match within braces "()". The default regular expression used if this is not defined is to match any number format you might obtain from program output (including exponential formats).

The scope statement allows us to restrict the tuning session to part of the parameter space.

4.3.1.1 Scenario Exploration

If the tuning mode is set to scenarios then the tuner will explore a set of scenarios and perform a tuning run over the tuning parameters for each scenario. The following tune entities relate to scenario exploration

```
scenario-params: param-list | collection
scenario-params-combiner: combinations | tuples
scenario-params-source: parameters | file
scenario-params-file: filename
matcher-feedback: script | code(C) | code (Fortran)
```

Once the mode is set to scenarios the default is to explore scenarios from all combinations of the values of any parameters mentioned in the scenario-params line. Note that the scope setting for parameters does not apply to scenario parameters as defining them as such puts them in scope. Optionally the parameters can come from a file if scenario-params-source is set to file and the filename is defined. If the parameter combiner is set to "tuple" then each scenario is formed as follows: Scenario 1 takes the first value of each parameter, Scenario 2 takes the second value and so on.

4.3.1.2 Repeated Runs

In order to allow for some performance variability the metric can be aggregated from multiple repeated runs. The DSL to control this is

```
run-repeats: nrepeats
metric-aggregation: min | max | average
```

If the metric aggregation method is not set then it is set to the same as the tuning target.

4.3.2 Previous state

In order to cater for incorporation of previous state we can provide a filename for a file which will be used to store the state of a tuning run::

```
state-file: filename
```

We also need to describe how this setting interacts with the autotuner implementation, the requirement being that by default the autotuner will record the state to the file if the state-file is set in the configuration file. The autotuner should provide a control for a user to continue a previous run.

The mockup script we implemented supports an argument (-continue *n*) which will continue a previous run from the step n or from the whole previous run if n is greater than or equal to the number of steps in the previous run.

4.3.3 Logfiles

An autotuner implementing the DSL should provide summary information to the standard output stream and more detailed information on tuning progress to a log file.

The following statement directs progress and state information to the specified location.

```
progress-log: filename
```

4.4 DSL parameters relating to application source

This section defines parameters relating to the location of DSL source and how that should be handled. A sample set of parameters follows:

```
begin sources
dsl-filelist: filelist
dsl-filenames-file: filename
dsl-map-input: input-map
dsl-map-output: output-map
end sources
```

The list of files to be processed is set by either supplying a list of filenames or a file which contains the filenames. If either of these is present then the tuner enables DSL parsing from source. The map definitions allow a mapping to be defined between the file containing DSL and the processed version after DSL parsing. This is best illustrated in the following example:

```
begin sources
dsl-filelist: a.f90.dsl b.c.dsl
dsl-map-input: %.dsl
dsl-map-output: %
end sources
```

The set of files is `a.f90.dsl` and `b.c.dsl`. These would be parsed to the files `a.f90` and `b.c` respectively. If a filenames file is provided then it can either contain one filename per line (in which case the mapping is used to obtain the output filename) or it can contain two filenames per line (the input and output filenames)

Once DSL parsing is enabled, the parsing process is started before any tuning runs commence (to optionally pick up new parameter definitions) and then each time a new build is run. The initial preprocess stage will parse the files assuming default values of tuning parameters.

4.5 Tuning parameters and relationships

The properties of tuning parameters can be described as part of the overall configuration (appropriate for runtime parameters for example) or can be declared in source DSL.

4.5.1 XDSL tuning parameter definition

The configuration contains a section where we define tuning parameters, their ranges constraints and aggregation. The parameters section includes various parts:

```
begin parameters
 begin typing
  type-entity
 …
 end typing
 begin constraints
  constraint-entity
  …
 end constraints
 begin values
  values-file: values-filename
 end values
 begin collections
  collection-entity
   …
 end collections
 begin dependencies
  depend: depend-list
 end dependencies
end parameters
```

The typing section allows parameters to be typed as int, real or label, more specifically:

```
type-entity is  type param-name
type        is  int | real | label
```

For example:

```
int np
int m
int Q
label method
```

The set of allowed values of a parameter are defined in the constraints section. This section supports specific sets of values, ranges, parameter relationships and legality constraints.

```
constraint-entity is range | product | constraint | default
range is value-list | value-range [ default value ]
default is value
value-range is value-list | value-triplet
constraint is logical expression | assignment
```

Some examples are:

```
range N 1-100
range M 20,40,60
range M2 20:60:20
range NB 100,110,120,130,140,150 default 120
range opt1 –O1,-O2
range threads 1,4
range nppn 1,2,4
range method buffer, nobuffer
product OPTa.c opt1 {-m32,-m64}
constraint M*N < NP
constraint Q = P / N
```

The default value is used when the parameter in question is not being varied by the tuning (otherwise the first value in the list or range is chosen). This default value can be overridden in the tuning section. Note that a constraint as an assignment means that the parameter that is the target of the assignment should be generated from the expression.

Parameters may be grouped into collections, for example:

```
Blocksizes: M N
decomposition: P Q
runtime: np, threads, pagesize
```

Note that collections may also be defined in the build and runtime configuration and some have special meanings.

The dependency section allows us to say which parameters should be treated as dependent: depend-list is either a list of parameters or a list of collections.

Note particularly the product definition which defines a product of the list of possibilities. Along with some (user-defined) naming convention understood by the build script this can be used to associate compile options with filenames by using the filename as part of the parameter name. In the example above we used OPTa.c which could be understood by a build script to define the current compile options (or additional options) to be used when compiling the file a.c.

The optional values section of the parameters section is used to define a file to import values for parameters. Normally parameters would be set from the constraints section but in the case of configuration parameters used for library routine tuning it is likely there could be a large number of parameters or they may come from an application run. The values file contains sections per parameter as follows

```
param param-name [ default default-value]
v0 v1 v2 …
```

Runtime and build parameters are naturally defined in the global configuration but we provide a related syntax to define parameter types and ranges in the source DSL as outlined below.

4.5.2 A node on tuning parameter validity

As mentioned previously in section 4.3.1 it is possible to define which parameters are in scope for a tuning session. As a result parameters can be fixed at their default values for a given run with minimal change to the configuration.

A feature request was to support a more dynamic mechanism so that parameters could have validity based on the values of other parameters *during* the tuning session. The following configuration parameters introduce new constraints to address this requirement:

```
constraint param inscope forscenario lexpr
constraint param inscope if lexpr
```

These definitions are valid in the constraints section of the configuration. The first form introduces a constraint that applies to scenario exploration. For each scenario this constraint is checked for each tuning parameter to see if it should be in scope for the tuning run. The second form applies more generally to tuning parameters and instructs the optimizer to avoid varying the parameter if the expression if false. To what extent this will reduce the search space depends on the ability of the optimizer to organize the search appropriately.

The second feature is somewhat experimental. A better (and much more complicated) approach which we may consider in the future would be to maintain a graph of parameter validity constraints.

An example of using such constraints is found in section 5.4.

4.5.3 DSL tuning parameter definition

Tuning parameters can also be declared in source DSL.

The syntax mirrors the syntax described above but with the tune sentinel prepended and definitions combined into an all-in-one syntax

```
#pragma tune param define p type t range r [default d]
```

which defines a new parameter *p* of type *t* with values from the range *r* and with an optionally defined default value *d*.

An example could be:

```
#pragma tune parameter n type int range 10,20
```

The parameters have global scope and can be used in DSL in source as described previously. An import directive is not required for later use of the parameter in the same file where the parameter was defined.

Tuning parameter constraints can also be defined in source DSL in the following variants

```
#pragma tune param define p type t constraint p=expr
#pragma tune constraint lexpr
```

The first variant constrains the new parameter to be generated from other parameters from the expression expr. The second form supplies a logical expression which determines if a set of tuning parameters is valid.

4.6 Parallel Autotuning Features

This section addresses tuning specific to a parallel application. Note that the generic framework can of course be used to do this by parameterization of control flow.

The following subsections address specific aspects of parallel tuning.

4.6.1 Stencil computation

This is the highest level aspect of parallel tuning that we hope to address.

A stencil is an operation on a grid expressed as updates to grid values as functions of nearby grid values. Once this is distributed in parallel we can decompose into local computation and a communication phase to move edge data that is required for computation on other processes. There are various ways to organize this and a stencil approach defines the problem generally leaving the details to the framework. We support stencils by using the pattern feature of data movement primitives.

4.6.2 Data movement primitives

Data movement primitives allow us to express parallel data movement and have the autotuner explore the best way to do this. This is implemented as follows:

1. We use DSL to optionally define patterns of data movement in an aggregate way.
2. We place DSL in source at points that data is available to be communicated and where it needs to be available.

Data movement patterns could be for example an alltoall pattern or a halo-swap communication pattern.

The DSL to accomplish this looks like the following:

```
#pragma tune pattern label mylabel ptype M N pmodel

#pragma tune label dlabel var available
#pragma tune label dlabel var available

… compute …

#pragma tune label dlabel var required
```

The pattern type (ptype) specifies one of predefined (or user supplied) patterns and this particular pattern is labelled with mylabel by the programmer. The pattern data dependencies are defined by the dlabel clauses as appropriate for the pattern. So to give an example we assume the availability of a 2D HALO pattern and the DSL would look like:

```
#pragma tune pattern label myhalo type HALO_2D M N MPI
myrank
#pragma tune myhalo left A(1:n) available
#pragma tune myhalo right B(1:n) available

… compute

#pragma tune myhalo left B(1:n) required
```

A range of available patterns should be predefined (or potentially user-supplied).

This idea can be used at a simpler level to just move data between locations. We will need to experiment with these ideas and work on the restrictions on what restrictions will be required in the source for this to work. Note that a pattern may require information from the program, in this case the global decomposition (M,N) and the variable containing the local rank (because we will use MPI).

4.6.3 Process Placement

A parallel application brings a new level of complexity at launch beyond a serial application. We can for example decide how many processes to use, where those processes are located and how they are mapped to the hardware. For a hybrid application (for example MPI application with OpenMP threading) we can trade processes for threads within the same total thread count and have various options for thread affinity. The framework of parameters can be used to explore this tuning space as these are all runtime parameters and just need mapped to the right environment variables or application launch options by the run script. The support for constraints was added to particularly address this scenario where for example threads multiplied by processes would be constant for a tuning run. Similarly an application may use an internal process decomposition (say P_x * P_y) which we may wish to vary.

How processes are placed on nodes can have a performance impact, for example the correct arrangement can yield an optimal on-node and off-node communication pattern for an application with particular data topologies. MPI implementations address this by either supporting a hostfile/mapfile for rank placement or a similar file or parameter to perform rank reordering. Our tuning infrastructure can accommodate the choice of such a file controlled by a runtime parameter. A possible rank order could be generated by a tool for a given topology or from MPI profiling data (the Cray software stack contains such tools). In addition it would be possible to reorder communicators within an MPI application per application phase and a reorder file could be an input to such a scheme, also controlled by a runtime parameter.

4.7 Build description and control

This is controlled by a section of the global configuration file where we define the interface to the build process:

```
begin build
 prescan-type: directory | script
 build-preprocess: directory | implicit
 command: shell-command
 param-file: filename

  begin collection
   BUILD: N M
  end collection

end build
```

We need to build the application taking account of the current set of tuning parameters. Because the source can contain DSL and we start by implementing this as source directives we need a mechanism to parse the DSL source. The prescan-type setting gives a choice of scanning a whole directory tree looking for source with DSL or calling a script that will cause the source to be scanned (this script could be the build command). Note that this script is called with an argument that provides a script which converts source to compilable source.

The initial scan is just looking for definitions of tuning parameters.

When the actual build is done we also need to preprocess, and the build-preprocess setting defines again if a scan is undertaken or if the normal build will use the preprocess script.

Note that we can alternatively define a specific set of files to scan in the sources section of the configuration. That method is likely to be more useful and less invasive.

The build progresses by running a shell-command which should exit with 0 exit code to indicate a successful build. The parameter list can be provided as a keyword list or as a file containing names and values for the parameter set. The parameters can be referenced as $*param* or ${param} in the command, the latter providing separation within a string. Here is an example:

```
begin build
 command: make N=$N P=$P
end build
```

Note that the collection BUILD has a special meaning and defines the set of parameters that would require a new build. This collection can be defined here or in the global configuration. If this is not defined then it is assumed that any change of parameters will require a new build.

Each build has an associated unique tag generated by the autotuner and this is available to the build command as $build_tag (or as ${build_tag}). This could be used for example as part of the executable name. The same tag is available to the run script.

4.8 Runtime information and control

This section of the global configuration describes the run process and is similar to that for the build process:

```
begin run
 command: shell-command
 param-file: filename
 validation-source: stdout | command
 validation-command: shell-command
```

```
  independence: none | n
end run
```

The application is run by the command script (which should return a successful exit status) and the current parameter values will be provided (optionally) in the command parameter file. The run can be validated by supplying data at the end of the standard output or providing the validation data as the output of a script. In particular the syntax of this output should be:

```
tune run status validation-status [ metric value ]
validation-status is validated | failed
```

Only a validation status of validated indicates a successful run. Optionally the tuning metric may be defined here. Note that because the tuning metric can be obtained from a script this allows extra flexibility, for example to obtain metrics that relate to power consumption, something that is not likely to be available to the application. Note that if validation is used the devault is to abort if a run is not validated. This behaviour can be changed by setting the failure mode:

```
Validation-failure-mode: abort | warning
```

Tuning parameter values and $build_tag are available to the run command. If more than one instance of the application can be run at the same time then the extent of execution parallelism can be declared via the independence setting. Two more variables (run_id) and (repeat_id) are available to the run command. The former is unique for each run and the latter would be defined for repeating runs and would be unique for each repeat. (Note that these variables are also available when defining the metric and validation filenames as these relate to a run.)

Environment variables can be provided to the run by including parameters in a collection called RUN_ENVARS. If this is done then at runtime the parameters will be mapped to environment variables of the same name, if the parameter is set to the value unset then the environment variable will not be defined.

4.9 Interfaces to tools and components

This section of the global configuration describes how we interact with external components. This is part of the global configuration.

4.9.1 Parameter optimization

This is where we describe the interface to the optimizer that explores the parameter space searching for the optimal set of parameters.

We do this via a “delegated control” interface where we setup the optimizer and then act on its responses by running the application and returning the resultant metric to the optimizer. Using this technique means that we only need the autotuner to understand how to optimize a set of parameters and not understand how to run the application.

The control section is as follows:

```
begin optimizer
  command: shell-cmd
  cycles: <integer>
end optimizer
```

The interaction with the optimizer proceeds as follows:

It is sent a start command and the parameter configuration (types, ranges, constraints, objective and a pointer to a file containing previous history of optimizer runs).

The optimizer should respond asking for a tuning run of the application for a given set of parameters. The framework sends back the results of the tuning run by returning the primary metric along with any secondary metrics. The process continues until the parameter space has been explored. The cycles parameter limits how many times the optimizer will be called, this would be used with a complex (intractable) search space and an optimizer that does not do exhaustive search in order to limit the computation.

4.9.2 Machine learning matcher

It is outside the scope of this project to implement a full machine learning system for the matcher component but our intention is describe the interface to this component for the full DSL specification. With the mockup we may be able to implement a simple closest match model and apply it to a library routine tuning example.

4.9.3 Library tuners

In this section we are concerned with the capability of specifically tuning a subroutine/function, something we would do if producing a library or optimized routines.

Our implementation uses the scenario exploration run where the scenario characterization parameters map to input parameters to the routine in question. So for example consider that we wish to tune a routine NORMALS which accepts arguments M,N. This function includes DSL to expose tuning choices with parameters B,L,O.

We perform a scenario run over values of M,N (assume for now these can be program input). The matcher produces a model mapping any M,N to the optimal choices or B,L,O and we instantiate that logic into a wrapper to call NORMALS appropriately. So in DSL this would look like:

```
#pragma tune library NORMALS scenario-params M, N tune
B,L,O wrapper NORMALS_wrap
```

This declares the parameters and names a wrapper routine that can be inserted after the matcher has run.

Additional features allow capturing the values of variables from within the program and supporting timing:

```
#pragma tune library NORMALS scenario-params M,N tune B,L,O
wrapper NORMALS EXPORT M,N timer
```

This would cause the program to be instrumented to export the M,N values and implement a timer which could be used as the tune metric.

4.9.4 Compiler-based tools

As discussed in section 3.6.4 this is a complex area and initially we will only support this by parameterized control flow and describing an interface whereby we could interact with a compiler that can expose its tuning choices.

For the latter we would hope that the compiler could create a companion file with the name “file.ctune” which contains the following DSL:

```
#pragma tune compiler-export
#pragma tune…
```

where the second and subsequent lines define tuning parameters.

The compiler should accept as input a file “file.ctune.in” in the same directory that sets those parameters.

5 Tuning Configuration Examples

Here we provide a set of examples with full configuration files to illustrate various features of the DSL.

5.1 Build and runtime parameter examples

In this example consider an application that has one tuneable parameter, some blocksize (NB) and that this can be set on the command line.

A possible autotuner configuration to tune for NB is as follows:

```
begin configuration
 begin tune
  mode: tune
  scope: NB
  target: min
  metric-source: runtime
 end tune
end configuration
begin parameters
 begin typing
  int NB
 end typing
 begin constraints
  range NB 80,90,100,120,140
 end constraints
end parameters
begin build
 command: make
end build
begin run
 command: ./solver NB=$NB
end run
```

We could have used a run script and picked up the value of NB from a provided input file (param-file) or from an environment variable (the RUN_ENVAR collection).

In the next example we assume that we have an additional tuning choice which is controlled at compile time by a preprocessor variable USE_EXTRA_BUFFER:

```
begin configuration
 begin tune
  mode: tune
  scope: NB EXTRA_BUFFERING
  target: min
  metric-source: runtime
 end tune
end configuration
begin parameters
 begin typing
  int NB
  label EXTRA_BUFFERING
 end typing
 begin collections
  BUILD: EXTRA_BUFFERING
 end collections
 begin constraints
  range NB 80,90,100,120,140
  range EXTRA_BUFFERING “YES”,”NO”
  depends NB EXTRA_BUFFERING
 end constraints
end parameters
begin build
 command: make EXTRA_BUFFERING=$EXTRA_BUFFERING
end build
```

```
begin run
 command: ./solver NB=$NB
end run
```

The differences are that we added the new parameter and passed it into the build. We included it in the BUILD collection to make sure that any changes cause a new build and we marked NB and EXTRA_BUFFERING as dependent so that the optimizer would not treat them independently in tuning.

5.2 Scenario Exploration

This example extends the previous example to add three scenarios SMALL, MEDIUM and LARGE.

```
begin configuration
 begin tune
  mode: scenarios
  scenario-params: SIZE
  scope: NB EXTRA_BUFFERING
  target: min
  metric-source: runtime
 end tune
end configuration
begin parameters
 begin typing
  int NB
  label EXTRA_BUFFERING
  label SIZE
 end typing
 begin collections
  BUILD: EXTRA_BUFFERING
 end collections
 begin constraints
  range NB 80,90,100,120,140
  range EXTRA_BUFFERING “YES”,”NO”
  range SIZE X100,X500,X10000
  depends NB EXTRA_BUFFERING
 end constraints
end parameters
begin build
 command: make EXTRA_BUFFERING=$EXTRA_BUFFERING
end build
begin run
 command: ./solver NB=$NB < input.$SIZE
end run
```

The additions define SIZE to have values of either X100, X500 or X10000 and these values will be used in turn to tune for the optimum values of EXTRA_BUFFERING and NB. In this case the SIZE parameter was used to choose the input file for the run command. It would be easy to keep the binary corresponding to the optimum parameters obtained for each scenario.

5.3 Example of DSL in source

This is a simple example to show how a source file may have DSL to control values of a variable.

```
!tune$ param define bfac type int range 20,40,80
!tune$ inject_r$:bfac = $bfac
do jb=1,n,bfac
 do ib=1,m,bfac
  do j=jb, min(jb+bfac,n)
  do i=ib, min(ib+bfac,m)
   …
   end do
   end do
 end do
```

```
    end do
```

An alternative would be for the global configuration to name `bfac` as a tuning parameter and to use an import directive instead of the parameter definition.

5.4 Scenario exploration with constrained parameters

This example shows how we can make some tuning parameters invalid for particular scenario runs. This shows what is likely to be a particular usage pattern for scenarios: where scenarios are used to explore different algorithms.

Consider this configuration:

```
begin configuration
 begin tune
  mode: scenarios
  scenario-params: LOOP
  scenario-params-combiner: combinations
  target: min
  metric-source: runtime
 end tune
end configuration
begin parameters
 begin typing
  int NB
  int block
  int unroll
  label LOOP
 end typing
 begin constraints
  range NB 80,90,100
  range block 2,4,8
  range unroll 1,2,4
  range LOOP blocking,unrolling
  constraint unroll inscope forscenario LOOP=="unrolling"
  constraint block inscope forscenario LOOP=="blocking"
 end constraints
end parameters
begin build
 command: make
end build
begin run
 command: ./solver NB=$NB
end run
```

This file defines a scenario run where the parameter LOOP takes the values “blocking” and “unrolling”, for each scenario a tuning run is started to find the optimum values of NB, block and unroll. But the constraints make sure that block is only varied for the scenario with LOOP=”blocking” and unroll is only varied for the scenario with LOOP=”unrolling”.

5.5 Compiler flag tuning

This example shows how tuning of compiler flags can be achieved with the framework. Assume that there are various source files main.c, solver.c, stats.c and that we wish to explore the use of certain options in the build. A sample configuration is shown below:

```
begin configuration
 begin tune
  mode: tune
  scope: copts
  target: min
  metric-source: runtime
 end tune
end configuration
begin parameters
 begin typing
```

```
   label OPT_strength
   label OPT_fp
   label OPT_default
   label OPT_num
   label FOPT_main.c
   label FOPT_solver.c
   label FOPT_stats.c
  end typing
  begin constraints
   range OPT_base -m64
   range OPT_fp -fp_model=strict,-fp_model=precise
   range OPT_strength -O2,-O3
   product OPT_num OPT_fp OPT_strength
   range FOPT_main.c OPT_num
   range FOPT_solver.c OPT_num
   range FOPT_stats.c OPT_num
  end constraints
  begin collections
   BUILD: OPT_base FOPT_main.c FOPT_solver.c FOPT_stats.c
  end collections
  begin dependencies
   depend: OPT_fp OPT_strength
  end dependencies
 end parameters
 begin build
  command: make
 end build
 begin run
  command: ./program
 end run
```

In this case the assumption is that the Makefile is expecting to use $OPT_base and the FOPT_xxx parameters to set the compilation options for each file. Note that in this example the same options are applied to the files, but if we split OPT_num to OPT_num1 and OPT_num2 and made those independent then each file would get distinct choices of OPT_fp and OPT_strength when the parameter search was done.

6 References

[1] Adrian Tate, “Industrial Auto-tuning with CrayATF”, iWAPT, Tokyo, Oct 2009, (abstract, presentation)

[2] Eds. Ken Naono, Kerita Teranishi, John Cavazons and Riji Suda, *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, Springer 2010.

[3] Trevor Hastie, Robert Tibshirani, Jerome Friedman, *The Elements of Statistical Learning: Data Mining, Inference and Prediction*, 2nd ed. Springer 2009.

[4] Sholom M. Weiss, Casimir A. Kulikowski, *Computer Systems that Learn: classification and prediction methods from statistics, neural nets, machine learning and expert systems*. M. Kaufmann Publishers, 1991

[5] Marjan Mernik, Jan Heering, and Anthony M. Sloane. *When and how to develop domain-specific languages*. ACM Comput. Surv. 37, 4 (December 2005), 316-344. DOI=10.1145/1118890.1118892

[6] David Henty, Luis Cebamanos, Jing Gong, Stefano Markidis and Alistair Hart, CRESTA project deliverable *D3.5.2 – Compiler Support for Exascale*,

[7] Luis Cebamanos, David Henty, Harvey Richardson and Alistair Hart, “*Auto-tuning an OpenACC accelerated version of Nek5000*”, EASC 2014, (to be presented).

Annex A. Autotuning Mockup

A.1 Introduction

In parallel with development of the DSL described here we produced a simple mockup autotuner implementation that could be used to inform the development of the DSL, give a view on practicability of features and provide something that CRESTA partners could use. The initial version was made available as an alpha prototype (this met a project milestone). Note that this is not production software but is an implementation internal to the CRESTA project that serves as motivator and demonstrator for the autotuning DSL specification (which is the actual deliverable of the project.)

A.2 Mockup status

The autotuner prototype is a ruby script that is capable of a rudimentary autotuning session controlled by a configuration script. Development is using ruby 1.9.1 with some very limited testing on 1.8.7.

Currently the following aspects are implemented:

- configuration script parsing
- sources section
- tune parameters
    - mode, scope, target, metric source
    - scenario setup (from configuration only)
    - metric-placement
- parameters section
    - typing section
- collections definition
- constraints
    - ranges, depends
    - product (but not the inline value syntax)
    - expression and assignment constraint types
    - scenario tune parameter validity/scope constraints
- build section
    - command definition with embedded parameters
    - param-file
    - build dependency from BUILD collection
    - executing build command
- run section
    - command definition with embedded parameters
    - param-file
    - executing run commands
    - validation behavior
- DSL source parsing (simple parameter definition, constraints, import, skip, replace and the three inject variants)
- determination of the order of executions and of best parameters by exhaustive search of parameter space.

Specific features not currently supported are: value input from file, parallel runs, plugins and the more advanced source DSL.

As the mockup developed, some new features were added and the syntax for existing features altered or extended. Those additions were incorporated into the DSL description elsewhere in this document so are not described here.

A.3 Mockup execution

The mockup autotuner is a command line application which is executed as follows:

```
% tune linpack.conf
```

This will start a tuning session controlled by the configuration file specified. The default log file will be tune.log. The autotuner accepts the following additional arguments:

```
-help          Print this usage information
-nobuild       Progress without executing the build command (for testing)
-norun         Progress without executing the run command (for testing
-noopt         Do not run the optimizer
-stripdsl      Parse DSL files removing DSL, do not run optimizer.
-continue seq  Continue a previous run from the seq'th run
-colour        Colorize standard output (at present just red for warnings)
-csv    file   Sent summary output to file in csv format
cfile          Name of the file that holds the tuning configuration
```

A.4 Acknowledgements, feature additions and suggestions

The mockup was used to tune OpenACC implementations in the Nek5000 application[6].

The main feature requests coming from that work were

- The ability to have better control over scope and validity of parameters. For example some parameters were only valid when a particular algorithm implementation was chosen.
- Required implementation of the scenarios
- Better summary of the results

The author received complimentary feedback about the usability of the software, in particular that is was not difficult to interface to the build and run of an application and extract the tuning metric.

The author acknowledges useful feedback on the DSL and mockup from Alistair Hart(Cray UK), David Henty(EPCC) and Luis Cebamanos(EPCC).

A.5 Availability

The mockup is currently only available internally to the project partners from the author or from the CRESTA SVN.

The SVN location is:

https://svn.ecdf.ed.ac.uk/repo/ph/cresta/wp3/autotuning/

The mockup files can be found in `trunk/mockup`

Annex B. Future Work

We expect to look at additional aspects of the DSL and tuning between publication of this deliverable and the end of the project. The following areas are of specific interest.

Table 1: Future work

DSL constructs	Expand support from current set
Matcher	Can we implement a simple closest-match or decision tree matcher for scenario runs?
DSL XML Format	Reconsider. Perhaps something like JSON is more appropriate.
Other project interactions	What can we learn from other projects? What software (ML, optimizers etc.) can we interoperate with and how?
Project interactions	Use tuner with an additional co-design application.
Parallel Data Mover Runtime	Experiment with data mover aspects