D3.6.1 –
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:	M12
Submission date:	30/09/2012
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	KTH
Version:	1.0
Status	Final
Author(s):	Harvey Richardson (CRAY)
Reviewer(s)	Dmitry Khabi (USTUTT), Xavi Aguilar (KTH)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	31/7/2012	First version	Harvey Richardson (CRAY)
0.2	7/8/2012	Small revisions, added product	Harvey Richardson (CRAY)
0.3	15/8/2012	Configuration parameters, more detail on library and compiler aspects	Andrew Turner (UEDIN)
0.4	29/8/2012	Reverted to template structure	Harvey Richardson (CRAY)
0.5	30/8/2012	References, examples, review version	Harvey Richardson (CRAY)
1.0	17/9/2012	Final draft with incorporated review comments	Harvey Richardson (CRAY)

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.

Subsequently we will refine the DSL based on experience with the CRESTA co-design applications and publish a final specification for project month 30.

The following sections introduce the 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.

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 initial specification for the DSL.

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 new unrelated work under CRESTA to research more general library tuning and whole-application tuning of parallel applications.

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.

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 characterization parameters and may, for example, map to input parameters relating to problem size. The other tuning parameters 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 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
    o Stencils
    o Data movement primitives
    o Process placement
- Build information and control
- Runtime information and control
- Interfaces to tools/components
    o Parameter optimization
    o Machine-learning (ML) matcher interface
    o Library tuners
    o 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 characterization parameters and process results with an ML system in order to build a tuned library or application that can cater for a range of characterization parameters.

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

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 make it into 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.

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

We also need to describe 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.

Parameter space exploration

> This is a variation on the above where some parameters are characterizations 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 an exploration characterization space 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 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 characterization parameters into a model that predicts the best set of tuning parameters for a given choice of 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 characterization parameters and the tuning parameters at the call site. The framework explores the characterization space and passes the result to the ML system. The ML system then creates an interface routine that maps the characterization 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 will be 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 will outline initial syntax in the form of directives/pragmas.

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). (We will mock-up an application to do this by the end of the project using a scripting language.)
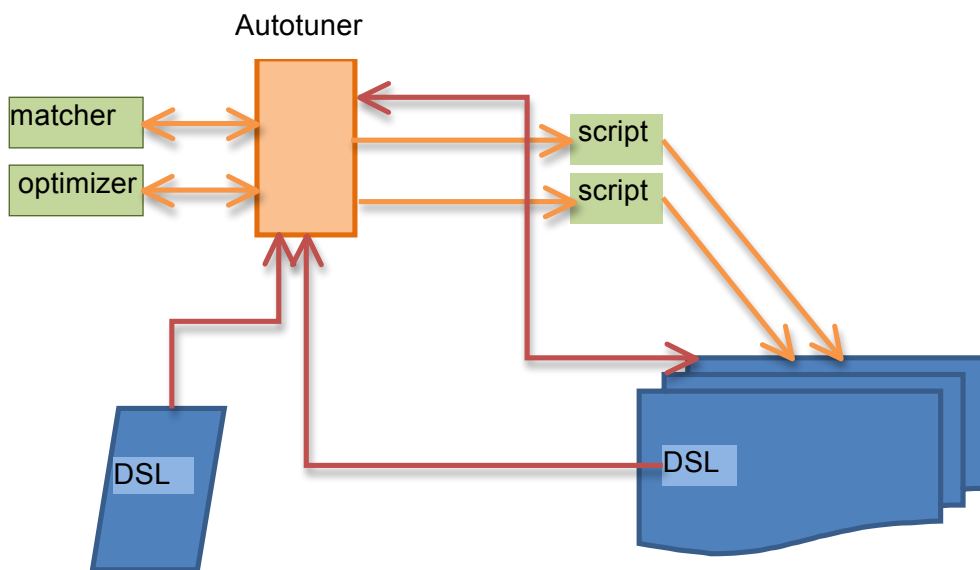

Figure 1: Tuning Architecture – high level view

The components shown in Figure 1 implement 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 characterization space 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 characterization run is undertaken then the

matcher is called to build a model that maps characterization 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 expect to be able to provide a specimen mock-up implementation that we can use to test the architecture by the end of the project. The optimizer can be implemented as an exhaustive search and the matcher can generate a closest match.

Our intention would be to use Ruby for the mock-up.

NOTE: Although we may define a compiler-parsed DSL by project completion we are not providing compiler infrastructure to compile it.

4.2 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 (text):

```
begin configuration
 <configuration entity>
end configuration
```

Allowable configuration entries are tuning target, tuning scope, previous statefile and logfiles.

4.2.1 Tuning target and scope

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

```
begin tune
 <tune entity>
…
end tune
```

where the tune entity is:

```
mode: tune | characterize
characterize-params: param-list / collection
characterize-param-source: parameter [ file | export ]
characterize-param-file: filename
characterize-param-sets: number
characterize-maxruns: number
target: min|max
scope: param-list / collection
metric-source: file | stdout | runtime
metric-placement: lastline | lastnumber| validation
```

If the mode is set to "tune" then we will optimize over the tune-scope parameters. If the mode is set to characterize then we will do a characterization run. For a characterization run the set of characterization parameters to use can come from various places as defined by the characterize-xxx settings. The characterization parameters could be just defined as normal parameters with ranges. They could come in as a set of values from a file of keyword-value pairs, one set per line or be exported from a run of the application. If the characterization values come from the application then the number of runs needs to be defined. The matcher feedback options define if the matcher is supposed to provide a wrapper for library tuning or produce a script to produce whole-program tuning.

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 metric can be obtained from the last line of a file, be the last recognizable number in a file or

can be output from a validation script executed after a single tuning run. 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 4.6).

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

4.2.2 Previous state

In order to cater for incorporation of previous state the following commands are defined:

```
use-previous-state: true | false
previous-state-file: filename
```

Optionally a filename can be given which will be used to store previous state. This would allow the continuation of a previous tuning session.

4.2.3 Logfiles

A log statement directs progress and state information to specified locations.

```
progress-log: filename
```

4.3 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.

4.3.1 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-entry>
 …
 end typing
 begin constraints
  <constraint-entry>
  …
 end constraints
 begin collections
  <collection-entry>
   …
 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-entry is  type 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 supports specific sets of values, ranges, parameter relationships and legality constraints.

```
Constraint-entry 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:

```
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 we used OPTa.c which would be understood by a build script to define the current compile options (or additional options) to be used when compiling the file a.c.

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.

4.3.2 DSL tuning parameter definition

Tuning parameters can also be declared in source DSL. With directive syntax we use a tune sentinel as follows:

```
!dir$ tune
#pragma tune …
```

The syntax can mirror the syntax described above but with the tune sentinel prepended. As an alternative we allow the all-in-one syntax:

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

For scoping, the default situation would be to assume that parameters have a scope of the next use of the variable and hence the autotuner (preprocessor) will insert an assignment for the specific value of the parameter just after the directive.

4.4 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.

4.4.1 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.

4.4.2 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 a 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
#pragme 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 will 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).

4.4.3 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.

One aspect we will look at in the next stage of the project is rank reordering where we remap ranks on launch. To do this intelligently requires information about application topology.

4.5 Build description and control

This would be 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
 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.

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 in the command. 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. This could be used for example as part of the executable name. The same tag is available to the run script.

4.6 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
  command-param-file: filename
  validation: stdout | script
  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 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 validation metric may be defined here. Note that because the validation 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.

The $build_tag is 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.

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.

4.7 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.

4.7.1 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.

4.7.2 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. In the mockup we may implement a simple closest match model and apply it to a library routine tuning example.

4.7.3 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 characterization run where the 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 characterization 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 characterization-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 characterization-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.

4.7.4 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 expect the compiler to create a companion file with the name "file.ctune" which contains the following DSL:

```
#pragma tune compiler-export
#pragme 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 Sample Configurations

Here we provide a couple of examples one with pure parameter tuning and one using source DSL.

5.1 Build and runtime parameter examples

In this example consider an application that has one tunable 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 (command-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 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.

```
    !dir$tune parameter bfac type int range 20,40,80
    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
```

The global configuration would name bfac as a tuning parameter.

5.3 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

Annex A. Deferred Work

This document describes the initial DSL specification which will be refined based on experience gained from the CRESTA applications. The following areas are deferred for future study:

Table 1: Future work

DSL constructs	In-place DSL parsing, Scoping and null-case (ignore tuning) templates (based on application use)
Runtime aspects	Rank reordering, template runtime
Mockup implementation	Basic parameter exploration scripts Basic optimizer Compiler flags
DSL XML Format	
Other project interactions	What can we learn from other projects? What software (ML, optimizers etc.) can we interoperate with and how?
Project interactions	Compiler support, Static autotuning or runtime planning?
CRESTA co-design	Nek5000 and parallel FFT perhaps?
Mock-up	