

Active Harmony Tutorial

July 8, 2009

Active Harmony is based on a client-server model. The client is the “harmonized” application, which sends performance numbers to the server. The server makes application tuning and adaptation decisions based on the numbers reported by the client.

The Harmony system consists of three main components: the Harmony parameter API (implemented in C++), the Resource Specification Language (RSL) (Tcl implementation) and parameter tuning/optimization algorithms (Tcl implementation). Harmonization of an application is done using the Harmony API. The RSL is used to communicate between the tunable programs (e.g., application or library) and the Harmony tuning server. The parameter tuning algorithms explore the optimization space of the application and adjust the parameter values based on observed performance.

This document will provide build instructions, steps needed to “harmonize” an application and will also describe key components of the tuning system.

1 Build Instructions

Listing 1 provides instructions on how to build and run the software.

Listing 1 Build Steps

```
$ tar -zxvf harmony.tgz
$ cd build
$ make
```

To run the server:

```
$ cd ../bin
$ ./hserver
```

To run the example client you have to setup the environment variables:
with bash:

```
$ export HARMONY_S_HOST=localhost
$ export HARMONY_S_PORT=1977
```

with csh/tcsh:

```
$ setenv HARMONY_S_HOST localhost
$ setenv HARMONY_S_PORT 1977
```

localhost is put just for demo purposes. You should set that variable to whatever hostname the server is running on. However, the port has to be 1977. If there is port number conflict, change it in hserver.h and recompile.

To build and run the h_server example client:

```
$ cd ../example/h_server
$ make
$ example
```

2 Using the C++ API

Listing 2 shows the changes made in the main program of a typical harmonized application. First the application registers with the harmony server using the `harmony_startup` function. Next, it sends the description of the application, which in the example above is read from a file. The application description consists of parameter space definition. An example of

Listing 2 Steps involved in harmonizing a code

```
/* initialize */
harmony_startup(0);
/* void harmony_application_setup_file(char *filename); OR
   void harmony_application_setup(char *description); where description is the TCL
   specification.
   A sample setup.tcl file is provided later in this document.
*/
harmony_application_setup_file('setup.tcl');

/* register tunable parameters */
/* void *harmony_add_variable(char *appName, char *bundleName, int type);
   Note that the appName has to be the name of the application used in the Tcl RSL
   specification and the bundleName have to the name of the bundle in the
   same specification.
*/
x = (int *) harmony_add_variable('test_tuning', 'x', VAR_INT);
y = (int *) harmony_add_variable('test_tuning', 'y', VAR_INT);

/* program main loop: These loops are application 'hotspots' (for example 3D
   stencil loop, matrix multiplication loop etc.). Loop-level performance
   measurements (time to execute the loop, L1/L2 cache miss/hit data etc.) are
   taken here and reported to the Harmony server.
*/

/* report the measurements: integer value is the parameter for
   harmony_performance_update function.
*/

harmony_performance_update(performance_result);

/* update values for the tunable parameters
*/
harmony_request_all();

/* end of program main loop
*/

/* finalize */
harmony_end();
```

the setup file is provided later in this document. Next, the parameters specified by the application in its description have to be bound with variables in the main program. The `harmony_add_variable` function takes care of this. The function binds a harmony variable to an application variable. The application can then use this bound variable, which will be periodically by the Harmony server. Periodically (typically on a per time step or per query basis), the application sends a value of the performance function to the harmony server by calling `harmony_performance_update`. The application then requests new values for the bound variables from the Harmony server invoking `harmony_request_all`. Finally, the application calls the `harmony_end` to un-register with the server.

In addition to the most relevant functions discussed above, the API provides other utility functions - `harmony_set_variable` (used to send the current value of the parameter to the server), `harmony_request_variable` (used to request the server-side value of a particular tunable parameter) and `harmony_set_all` (used to update the values of the parameter at the server-side).

3 Resource Specification Language

The RSL is implemented using the TCL scripting language. TCL was chosen because it provides support for arbitrary expression and function evaluations. Listing 3 shows an example (along with embedded comments).

The `harmonyApp` keyword precedes the description of an application. The application description contains the parameter space definition for the tunable parameters. A tunable parameter of the application is defined using the `harmonyBundle` tag and is characterized by type and range of allowable values. In Listing 3, `x` variable can take any even integer value between 8 and 128 inclusive. `obsGoodness` tag is used to describe a range for an application-defined metric that is used by the tuning algorithm. For example, a scientific simulation might be described by a metric that indicates the time required to process a timestep of data. Since a single value of an `obsGoodness` might not be indicative of the overall performance of the application, an optional `numValues` attribute can be defined that indicates the

number of values to be collected, aggregated and reported to the optimization algorithm. `predGoodness` tag describes a range of expected values for the performance function.

`global` tag is used for tunable parameters and for the `goodness` tags. Different instances of the same application (i.e. MPI processes of a SPMD program) can define a global bundle, which is used to simultaneously tune values of local (processor-specific) variables.

Listing 3 Harmony Resource Specification (aka Search Space Definition): `setup.tcl`

```
harmonyApp test_tuning {
  { harmonyBundle x {
    # <type> {<min> <max> <step>} global?
    int {8 128 2}
  }
  { harmonyBundle y {
    int {8 128 2}
  }

  # observed performance and predicted performance
  # control the dimension of display window and
  # have no impact on the tuning process.
  { obsGoodness 500 1100 }
  { predGoodness 300 800 }
}
```

For more information on both the C++ API and the RSL, please look into their respective manuals in `docs` directory. In addition, examples provided in `examples` directory should help as well.

4 Initial Simplex Construction Options

Initial simplex plays an important role on how the search progresses. By default, the initial simplex is constructed by considering the extreme values. For example, the default initial $N + 1$ simplex for the example (with parameters `x` and `y`) provided in 3, will be:

$$(8, 8), (128, 8), (128, 128)$$

End-users can change this behavior by specifying where to start the search from and the initial simplex size (more specifically, the distance between points in the simplex). To activate, users need to change the value of `init_simplex_method` variable in the `hconfig.tcl` file, which resides in the `HARMONY_HOME/build` directory. By default, this variable is set to `max`. One can change this to `user_defined`. User-defined mode of simplex construction requires specification of two additional parameters to the `harmonyBundle` construct in the `.tcl` file. An example is provided below.

Let's consider two parameters, `x` and `y`. The definition of `harmonyBundle` for `x` and `y` can be as follows:

```
{ harmonyBundle x {
    # <type> {<min> <max> <step> <initial_value> <initial_scale>}
    #       where <initial_value> is the start value for x and
    #       <initial_scale> determines the initial scaling
    #       distance for this parameter.
    int {8 128 2 50 10}
}
{ harmonyBundle y {
    int {8 128 2 40 20}
}
}
```

The initial $N + 1$ simplex will be:

$$(50, 40), (60, 40), (50, 60)$$

5 Parameter Tuning Algorithm

Currently, two tuning algorithms make the optimization kernel of the Harmony Server: Brute-Force (explores every allowable configuration in the search space) and Nelder-Mead

Simplex algorithm. The choice of what optimization algorithm can be made at server launch-time. All algorithms are implemented on top of Tcl. A call to `harmony_performance_update` function from the client application automatically invokes the search process. The updated values of the parameters can then be accessed by the client via Harmony API call to update the variables.

Nelder-Mead simplex is an optimization algorithm based on the original simplex algorithm. The algorithm maintains a $N+1$ simplex and at each iteration of the algorithm the simplex is moved to the minimum by considering the worst point in the simplex and forming its symmetrical image through the center of the opposite (hyper) face.

A more advanced Parallel Rank Ordering algorithm is in its development phase. A new release will be available at the project webpage.

6 Further Reading

Harmony software as well as the documentation of relevant APIs can be accessed at <http://www.dyninst.org/harmony>.