Automatic Performance & Energy Tuning
with the Periscope Tuning Framework

Renato Miceli
renato.miceli@fieb.org.br
SENAI CIMATEC Supercomputing Center for Industrial Innovation

Michael Gerndt
gerndt@in.tum.de
Technische Universität München
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup
• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH
• AutoTune Demonstration Center
Background

• Given
  – Multicore, Accelerators, HPC Cluster
  – Many programming models and applications

• Problem
  – How to tune application for a given architecture & application?

• Targeted Users
  – Improve performance: HPC Application Developers
  – Faster tuning: HPC Application Users
  – Reduce energy cost: Supercomputing Centers
Motivation

• Why tune applications?
  – Increasing complexity of HPC architectures
  – Frequently changing HPC hardware
  – Compute time is a valuable resource

• Manual tuning
  – Large set of tuning parameters with intricate dependencies
  – Diverse set of performance analysis tools
  – Several iterations between analysis and improvements
Tuning Stages

• **Measure**
  – Application tuning runs
  – Performance data collection
  – Identify metrics

• **Analyze**
  – Paradigm and programming model
  – Search space strategies

• **Optimize**
  – Apply identified optimizations
  – User knowledge

• **Test**
  – Re-evaluate
An Ideal Solution

• **Productivity**
  – Removes burden of tuning from developers

• **Portability**
  – Portable tuning techniques across different environments

• **Reusability**
  – Same techniques applied across different applications

• **Flexibility**
  – Re-evaluate optimizations for different scenarios

• **Performance**
  – Produces at least average level of performance
Periscope Tuning Framework

• Objective - Single tool for performance analysis and tuning
• Extends Periscope with a tuning framework
• Tuning plugins for performance and energy efficiency tuning
• Online tuning
• Combine multiple tuning techniques
AutoTune Consortium

Technische Universität München, Germany

Universität Wien, Austria

CAPS Entreprise, France

Universitat Autònoma de Barcelona, Spain

Leibniz Computing Centre, Germany

Irish Centre for High-End Computing, Ireland
AutoTune Goals

• Automatic application tuning
  – Tune performance and energy
  – Create a scalable and distributed framework
  – Evaluate the alternatives online
  – Tune for Multicore and GPU accelerated systems

• Variety of parallel paradigms
  – MPI, OpenMP, OpenCL, Parallel pattern, HMPP
Recapping...

• Tune application
  – Automatic tuning is necessary because manual tuning is a time consuming and cumbersome process

• Ideal tool
  – Should be able to perform tuning automatically with improved productivity, flexibility, reusability and performance.

• AutoTune Project
  – AutoTune project is addressing
Overview

- Motivation
- The AutoTune Project
- Periscope Tuning Framework (PTF)
  - Architecture
  - Plugins
  - Installation & Setup
- Case Study & Best Practices
  - NPB
  - SeisSol
  - LULESH
- AutoTune Demonstration Center
Periscope Performance Analysis Toolkit

• Online
  – no need to store trace files
• Distributed
  – reduced network utilization
• Scalable
  – Up to 100,000s of CPUs
• Multi-scenario analysis
  – Single-node Performance
  – MPI Communication
  – OpenMP
• Portable
  – Fortran, C with MPI & OMP
  – Intel Itanium2, x86 based systems
  – IBM Power6, BlueGene P, Cray
AutoTune Approach

• AutoTune will follow Periscope principles
  – Predefined tuning strategies combining performance analysis and tuning, online search, distributed processing.

• Plugins (online and semi-online)
  – Compiler based optimization
  – HMPP tuning for GPUs
  – Parallel pattern tuning
  – MPI tuning
  – Energy efficiency tuning
  – User defined plugins - info in developer session
Periscope Framework

- Extension of Periscope
- Online tuning process
  - Application phase-based
- Extensible via tuning plugins
  - Single tuning aspect
  - Combining multiple tuning aspects
- Rich framework for plugin implementation
- Automatic and parallel experiment execution
Plugin Lifecycle

1. Static Analysis and Instrumentation
2. Analysis using Periscope Frontend
3. Tuning Strategy
4. Analysis Strategy
5. Plugin Strategy
6. Generate Tuning Report
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup
• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH
• AutoTune Demonstration Center
## Plugin Overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Target</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler Flag Selection (CFS)</td>
<td>Compiler flag combinations</td>
<td>Optimize performance</td>
</tr>
<tr>
<td>Dynamic Voltage and Frequency Scaling (DVFS)</td>
<td>CPU Frequency and power states</td>
<td>Reduce energy consumption with minimal impact</td>
</tr>
<tr>
<td>High Level Pipeline Patterns</td>
<td>Pipeline stages</td>
<td>Optimize throughput</td>
</tr>
<tr>
<td>MPI Runtime</td>
<td>MPI parameters</td>
<td>Optimize SPMD MPI application performance</td>
</tr>
<tr>
<td>Master – Worker MPI</td>
<td>Workload imbalance in MPI applications</td>
<td>Balance application load by optimizing communication</td>
</tr>
<tr>
<td>Hybrid Manycore HMPP Codelets</td>
<td>Codelet and many core runtime parameters</td>
<td>Improve utilization of many core architecture with codelets</td>
</tr>
</tbody>
</table>
Compiler Flag Selection (CFS)

• **Goal**
  – Optimize application performance by guiding machine code generation using compiler flags

• **Tuning Technique**
  – Selection of compiler flags and corresponding values
  • -O1, O2, -floop-unroll
  – Search through combination of flags
Dynamic Voltage and Frequency Scaling (DVFS)

• **Goal**
  – Reduce energy consumption with the objective of optimal performance to energy ratio

• **Tuning Technique**
  – Select CPU governor and frequency, processor specific power states
  – Energy prediction model avoids evaluating all frequency governor combinations
High Level Pipeline Patterns

• Goal
  – Optimizing throughput of pipeline patterns by exploiting CPU and GPU effectively

• Tuning Technique
  – Selection of stage replication factor
  – Sizes of intermediate buffers
  – Choice of hardware for executing a task
MPI Runtime

• Goal
  – Tweaking MPI Parameters to improve SPMD code performance

• Tuning Technique
  – MPI runtime parameters – application mapping and buffer/protocol
  – MPI communication parameters
    - Eager limit, buffer limit
Master-Worker MPI

• **Goal**
  – Optimize execution time by balancing application load and decreasing communication between workers

• **Tuning Technique**
  – Impact of varying partition size per worker
  – Number of workers
  – Size of partition per worker
Hybrid Manycore HMPP Codelets

• Goal
  – Optimization of HMPP Codelet computations on accelerators to improve many-core performance

• Tuning Technique
  – Selection of static codelet variant
  – Selection of runtime parameters
    • GPU grid size, ...
  – HMPP callsite locations
Recapping...

• Periscope Tool
  – Periscope performs the performance analysis

• Periscope Tuning Framework (PTF)
  – PTF extends the Periscope to use the performance data and allows developers to write tuning plugins

• Tuning Plugins
  – Tuning plugins uses PTF infrastructure and automate the tuning of applications using on or more tuning techniques
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup
• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH
• AutoTune Demonstration Center
Setting up PTF

- **http://periscope.in.tum.de/**
- **Version:** 1.1 (latest stable from April 2015)
- **Supported systems:**
  - x86-based clusters
  - Bluegene
  - IBM Power
- **License:** New BSD
Setting up PTF

• Download PTF
  – http://periscope.in.tum.de/releases/latest/tar/PTF-latest.tar.bz2

• Installation of PTF
  – Uses GNU make
  – Simple steps to install

• Third party library requirements
  – ACE (version >= 6.0)
  – Boost (version >= 1.47)
  – Xerces (version >= 2.7)
  – Papi (version >= 5.1)
Setting up PTF

• Plugin Specific libraries
  – Enopt library for the DVFS plugin
  – Vpattern library for the Patterns plugin

• Doxygen documented code

• Plugin Specific user guides

• Sample applications repository
Using PTF

- **Identifying Phase Regions**
  - User regions are the computationally intensive part
  - PTF measurements focus on phase region
  - Optional step
  - Entire application is default phase region

- **Instrumenting the application**
  - Use `psc_instrument` command

- **Executing application**
  - Use periscope frontend for execution
  - Use `psc_frontend` command

- **Interpreting the results**
Finding Phase Region

Identify phase regions using the output of gprof or equivalent tool.

e.g. Serial NPB BT-MZ benchmark

```
Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27.35</td>
<td>237.34</td>
<td>237.34</td>
</tr>
<tr>
<td>15.08</td>
<td>368.18</td>
<td>130.83</td>
</tr>
<tr>
<td>14.49</td>
<td>493.93</td>
<td>125.75</td>
</tr>
<tr>
<td>13.55</td>
<td>611.48</td>
<td>117.55</td>
</tr>
<tr>
<td>11.89</td>
<td>714.63</td>
<td>103.15</td>
</tr>
<tr>
<td>10.89</td>
<td>809.12</td>
<td>94.49</td>
</tr>
<tr>
<td>3.70</td>
<td>841.24</td>
<td>32.11</td>
</tr>
</tbody>
</table>

[2] 100.0 0.00 867.87 main [2]
    0.02 867.85 1/1 MAIN__ [1]

[3] 99.0 0.02 859.19 51456/51456
    0.02 859.19 51456
    130.83 135.30 51456/51456
    103.15 139.31 51456/51456
    94.49 134.66 51456/51456
    116.97 0.00 51456/51712
    4.48 0.00 51456/51456

<spontaneous>
```

Setting up PTF

Find Phase Region

Instrumenting Application

Executing Application

Interpreting Results

Finish
Finding Phase Region

Annotating the code within loop using pragmas

```c
 c-- start the benchmark time step loop
 c-----------------------------------------------

do step = 1, niter

   if (mod(step, 20) .eq. 0 .or. step .eq. 1) then
     write(*, 200) step
     format( ' Time step ', i4)
   endif

1$MON user region
   call exch_gbc(u, qbc, nx, nxmax, ny, nz)

   do zone = 1, num_zones
     call adi(rho i(start1(zone)), us(start1(zone)),
        vs(start1(zone)), ws(start1(zone)),
        qsl(start1(zone)), square(start1(zone)),
        rhs(start5(zone)), forcing(start5(zone)),
        u(start5(zone)),
        nx(zone), nxmax(zone), ny(zone), nz(zone))
   end do
1$MON end user region

end do
```

BT-MZ/bt.f

1. Setting up PTF
2. Find Phase Region
3. Instrumenting Application
4. Executing Application
5. Interpreting Results
6. Finish
Adding Instrumentation

Changing the makefile to add instrumentation information

```
# This is the fortran compiler used for fortran programs
#F77 = gfortran -p
F77 = psc_instrument -v -d -s ../bin/bt-mz.C.x.sir -t user mpi90 -g
# This links fortran programs; usually the same as $(F77)
FLINK   = $(F77)
```

---

**Flowchart: Instrumenting Application**

1. **Start**
2. Setting up PTF
3. Find Phase Region
4. Instrumenting Application
5. Executing Application
6. Interpreting Results
7. Finish
Executing Application

Prepare a config file

```c
// application related settings
makefile_path="../";
makefile_flags_var="FLAGS";
makefile_args="BT-MZ CLASS=C TARGET=BT-MZ";
application_src_path="../BT-MZ";

// plugin related settings
selective_file_list="x_solve.f y_solve.f z_solve.f";
make_selective="true";
search_algorithm="exhaustive";
compil_opt = "-O" ["01", "02", "03"];
```

Start

- Setting up PTF
- Find Phase Region
- Instrumenting Application
- Executing Application
- Interpreting Results
- Finish
Executing Application

- The `psc_frontend` command is used to execute PTF.
- The `--tune=compilerflags` option is used to select the plugin.

```
$ psc_frontend --apprun="./bt-mz.C.x"
  --starter=FastInteractive --delay=2
  --mpinumprocs=1 --tune=compilerflags
  --force-localhost
  --debug=2 --selective-debug=AutotuneAll
  --sir=bt-mz.C.x.sir
```
Interpreting the results

Combination executing in minimal time is reported as the optimal scenario

Optimum Scenario: 1

Compiler Flags tested:
Scenario 0 flags: "-01"
Scenario 1 flags: "-02"
Scenario 2 flags: "-03"

All Results:
Scenario | Severity
0  | 4.67491
1  | 4.49332
2  | 4.66382

[psc_frontend][INFO:fe] Plugin advice stored in: advice_15827.xml

----------
End Periscope run! Search took 77.0425 seconds ( 10.3693 seconds for startup )

----------
[psc_frontend][INFO:fe] Experiment completed!
Recapping...

- Installing Periscope Tuning Framework
- Using Periscope Tuning Framework
- Instrumenting and executing a sample application
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup

• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH

• AutoTune Demonstration Center
Tuning Results

Comparison of manual tuning against automatic tuning

<table>
<thead>
<tr>
<th></th>
<th>Time (Sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (With -O2)</td>
<td>872.4</td>
</tr>
<tr>
<td>Manual Tuning</td>
<td>797.8</td>
</tr>
<tr>
<td>PTF Suggested Flags</td>
<td>797.5</td>
</tr>
</tbody>
</table>
Search strategy influencing the number of scenarios executed and total execution time of the plugin.
Improvement in the PTF execution time to find the optimal flags for BT-MZ class C

Uninstrumented: 42299.7
Instrumented: 2790.63
Selective Make: 2070.3
Individual Search: 285.6
Fast Starter: 182.6
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup
• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH
• AutoTune Demonstration Center
SeisSol Use Case

• Developed by the Department of Earth and Environmental Sciences at the Ludwig-Maximilian University.

• Application simulates realistic earthquakes, propagation of seismic waves e.g. viscoelastic attenuation, strong material heterogeneities and anisotropy.

• Code Details
  – FORTRAN Code
  – MPI
SeisSol Use Case

Execution of SeisSol with the DVFS plugin

<table>
<thead>
<tr>
<th>Found Optimum Scenario: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Path:</td>
</tr>
<tr>
<td>Scenario</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

- Energy Prediction Model predicted the optimal frequency as 1.2GHz
- Plugin tests +/- 100MHz step for the plugin
- Here 1.2GHz is the minimum frequency supported by hardware
- ~26% energy is saved at the cost of ~10% increased execution time
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup
• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH
• AutoTune Demonstration Center
LULESH Use Case

• Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)
• Modelling hydrodynamics
• DOE proxy applications

• Code Details
  – C++ Code
  – Serial, OMP, MPI, MPI+OMP
LULESH Use Case

CFS Tuning Results

<table>
<thead>
<tr>
<th>Time (Sec)</th>
<th>Baseline</th>
<th>PTF</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50.6</td>
<td>45.8</td>
</tr>
</tbody>
</table>

Baseline vs. PTF
LULESH Use Case

Execution Results with the DVFS plugin

- 1 iteration

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Governor</th>
<th>Freq (MHz)</th>
<th>Energy (J)</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Userspace</td>
<td>1200</td>
<td>4.000</td>
<td>0.327</td>
</tr>
<tr>
<td>1</td>
<td>Userspace</td>
<td>1300</td>
<td>4.000</td>
<td>0.307</td>
</tr>
</tbody>
</table>

- 1000 iterations

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Governor</th>
<th>Freq (MHz)</th>
<th>Energy (J)</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Userspace</td>
<td>1200</td>
<td>2709.000</td>
<td>115.537</td>
</tr>
<tr>
<td>1</td>
<td>Userspace</td>
<td>1300</td>
<td>2740.000</td>
<td>106.674</td>
</tr>
</tbody>
</table>
Meta-plugins

• Fixed Sequence Plugin
  – Executes selected set of plugins in the given order

• Adaptive Sequence Plugin
  – Uses the optimal scenario found by the previous plugin
Overview

• Motivation
• The AutoTune Project
• Periscope Tuning Framework (PTF)
  – Architecture
  – Plugins
  – Installation & Setup
• Case Study & Best Practices
  – NPB
  – SeisSol
  – LULESH
• AutoTune Demonstration Center
AutoTune Demonstration Center

- The AutoTune Demonstration Centre (ADC)
  - Established by AutoTune partners & supporters

- ADC objectives:
  - Spread the use of AutoTune software
  - Further exploit energy save and power steering
  - Disseminate Best Practice Guides
  - Provide developers with a platform to test/validate
  - Become a platform for exchange of information
  - To extend the PTF developer community

- ADC services:
  - Online documentations, best practice guides
  - Discussion forums
  - AutoTune training events
  - Individual support for applications
  - Education and training of students
  - AutoTune website as central hub
  - Service Desk and Issue Tracking

Structure of the ADC
Conclusions

• Automatic Tuning - Motivation and AutoTune
• Periscope Tuning Framework (PTF)
• Tuning Plugins
• Instrumenting and executing sample application
• Demo : NPB, SeisSol, LULESH
• Best Practices
• AutoTune Demonstration Center
PTF Tutorial for Plugin Developers

Michael Gerndt
gerndt@in.tum.de
Overview

• PTF Architecture
• Tuning Concepts
• OpenMP Scalability Plugin
• Design Your Own Plugin
• Phase Region Tutorial
• Analysis & Region Selection Plugin
• Advanced Concepts and Tutorial Plugins
PTF Components

- **Tuning Parameters**
  - Aspects that can be changed and affect performance
  - Can be plugin specific
  - Semantics are strictly defined in each plugin

- **Agents**
  - Collects performance metrics from the MRI Monitor
  - Generate performance properties
  - Propagate properties to the frontend and finally to the plugin

- **ScoreP Monitor**
  - Collect the performance metrics
  - Support for different runtime systems
PTF Components

• Search algorithm generates the combinations of tuning parameters to form a tuning space to be evaluated
• There are several search algorithms available
  – Exhaustive search strategy
  – Individual search strategy
  – Random search strategy
• One can also write the custom search strategy
Tuning Plugin

- Codifies the expert knowledge to automate the tuning process
- Interfaces with PTF and uses its infrastructure to collect performance data for tuning the application
- To interact with PTF it is required that the plugins implement the **Tuning Plugin Interface** (TPI)
Tuning Plugin Interface

- Abstract class that defines the TPI operations
- Has a method for each step in the workflow
- Each plugin has to implement the members of the IPlugin interface class
- The PTF runtime system calls the implemented methods while executing a plugin
  - The order of the operations is predetermined by a state machine specification
  - A plugin dictates which valid paths to take in the tuning flow
class IPlugin {
protected:
    DriverContext *context;
    ScenarioPoolSet *pool_set;
public:
    virtual ~IPlugin() = 0;
    virtual void initialize(DriverContext *context, ScenarioPoolSet *pool_set) = 0;
    virtual void startTuningStep(void) = 0;
    virtual bool analysisRequired(StrategyRequest** strategy) = 0;
    virtual void createScenarios(void) = 0;
    virtual void prepareScenarios(void) = 0;
    virtual void defineExperiment(int numprocs, bool& analysisRequired, StrategyRequest** strategy) = 0;
    virtual bool restartRequired(std::string& env, int& numprocs, std::string& cmd, bool& instrumented) = 0;
    virtual bool searchFinished(void) = 0;
    virtual void finishTuningStep(void) = 0;
    virtual bool tuningFinished(void) = 0;
    virtual Advice *getAdvice(void) = 0;
    virtual void finalize(void) = 0;
    virtual void terminate(void) = 0;
};
PTF Tuning Concepts
Important Concepts

• Search space
• Objectives
• Tuning scenarios
• Experiment
Search Space

• **Variant Space**
  – Multidimensional space of tuning parameters

• **Search Space**
  – Variant space plus regions where to apply the tuning parameters
Objectives

• PTF supports max/min of severity of a single performance property for a scenario.
  – Performance property
    • Formalizes a performance characteristic
    • Consists of condition, confidence, and severity
  – Objectives can be based on any performance property.

• Support for user defined objectives
  – E.g. Mean severity of the properties found in different processes.
Tuning Scenario

• It specifies the evaluation of objectives and the tuning configuration.

• Consist of
  – **Tuning configuration**: list of tuning parameter settings, regions, and processes.
  – **Objectives**: tuned region, property list, and processes.
Scenario Pools

Created Scenario Pool

Prepared Scenario Pool

Experiment Scenario Pool

Finished Scenario Pool
Experiment

• Evaluation of scenarios through measurements.
• Configured as
  – List of scenarios
  – Scenario analysis if required
OpenMP Scalability Plugin
Scalability Tuning Plugin

OpenMP

- Allows the parallelization of serial code
- Easy to annotate code with OpenMP pragmas
- Uses multiple threads for execution
- Number of threads can be changed

Motivation

- Execution time differs for different number of threads
- Scaling is not linear (more is not always better; energy)
- Need to find the optimal number of threads
Using Developers Mode

• Install PTF with developers mode enabled
  --enable-developer-mode
• Developer's mode provides
  – Skeleton plugin generator script
  – Small utilities to ease development
    • Such as a source code locator
Generating Plugin Skeleton

• Generate a plugin skeleton using skeleton generator script

  
  psc_generate_bare_plugin_from_skeleton \ 
  -c ExamplePlugin -l ExamplePlugin

• Creates a skeleton in

  <PTF-source>/autotune/plugins/ExamplePlugin

• Generates source (src/ExamplePlugin.cc) and header file (include/ExamplePlugin.h)

• Compiles into a library (libptExamplePlugin.la)

• To run a plugin using PTF use --tune=ExamplePlugin with psc_frontend command

  
  psc_frontend --apprun="./appToTune" --sir=appToTune.sir \ 
  --mpinumprocs=8 --tune=ExamplePlugin --force-localhost
Plugin Management Info

- `IPlugin* getPluginInstance(void);`
  Returns the instance of the plugin

- `int getVersionMajor(void);`
  Returns the major version number of the plugin

- `int getVersionMinor(void);`
  Returns the minor version number of the plugin

- `string getName(void);`
  Returns the name of the plugin

- `string getShortSummary(void);`
  Returns the short description about the plugin
Plugin Initialization

- Create local references for the context and pool_set
  - Context initiated by PTF to offer services to the plugins
  - pool_set contains the scenario pools
- Define the tuning parameters
- Define the search algorithm
void ExamplePlugin::initialize(DriverContext *context, ScenarioPoolSet *pool_set) {
    this->context = context;
    this->pool_set = pool_set;
    TuningParameter *numberOfThreadsTP = new TuningParameter();
    numberOfThreadsTP->setId(0);
    numberOfThreadsTP->setName("NUMTHREADS");
    numberOfThreadsTP->setPluginType(ExamplePlugin);
    numberOfThreadsTP->setRange(1, context->getOmpnumthreads(), 1);
    numberOfThreadsTP->setRuntimeActionType(FUNCTION_POINTER);
    tuningParameters.push_back(numberOfThreadsTP);
    string results = numberOfThreadsTP->toString();

    int major, minor;   string name, description;
    context->loadSearchAlgorithm("exhaustive", &major, &minor, &name, &description);
    searchAlgorithm = context->getSearchAlgorithmInstance("exhaustive");
    if (searchAlgorithm){
        searchAlgorithm->initialize(context, pool_set);
    }
}
Start Tuning Step

• Generates a search space as a cross-product of tuning parameters
• Adds the region to which the tuning parameters are applied
• Decides which tuning parameters to explore in the current step
void ExamplePlugin::startTuningStep(void)
{
    VariantSpace *variantSpace=new VariantSpace();
    SearchSpace *searchSpace=new SearchSpace();

    for (int i = 0; i < tuningParameters.size(); i++)
    {
        variantSpace->addTuningParameter(tuningParameters[i]);
    }

    searchSpace->setVariantSpace(variantSpace);
    searchSpace->addRegion(appl->get_phase_region());
    searchAlgorithm->addSearchSpace(searchSpace);
}
Analysis Required

- In this step a performance analysis strategy could be triggered to get performance info
- Can be used to decide the further execution of plugin
  - e.g. Used to predict the optimal frequency for the DVFS plugin

```cpp
Bool ExamplePlugin::analysisRequired( StrategyRequest** strategy )
{
    return false;
}
```
Creating Scenarios

- Create scenarios from all possible combinations of the tuning parameters in the search space.

```c++
void ExamplePlugin::createScenarios(void)
{
    searchAlgorithm->createScenarios();
}
```
Prepare Scenarios

- Used to do preparation for scenario
- e.g. Recompiling application, setting up environment

```c
void ExamplePlugin::prepareScenarios(void) {
    while(!pool_set->csp->empty()) {
        pool_set->psp->push(pool_set->csp->pop());
    }
}
```
Define Experiment

- Scenarios are mapped to the runtime environment (Processes, threads)
- Execute the scenario and measure the performance properties
- The PTF runtime system then loops until all scenarios were evaluated through experiments

```c
void ExamplePlugin::defineExperiment(...) {
    Scenario *scenario = pool_set->psp->pop();

    Scenario->setSingleTunedRegionWithPropertyRank(appl->get_phase_region(), EXECTIME, 0);

    pool_set->esp->push(scenario);
}
```
Restart Required

- Indicates to the PTF runtime system whether the scenario requires an application restart to modify tuning parameters

```cpp
bool ExamplePlugin::restartRequired(
    std::string& env, int& numprocs,
    std::string& command,
    bool& is_instrumented)
{
    return false;
}
```
Search Finished

• Step determines if the search is finished
• It can be delegated to search algorithm to decide whether the search is finished

```cpp
bool ExamplePlugin::searchFinished(void) {
    return searchAlgorithm->searchFinished();
}
```
Finish Tuning Step

- Perform any post-processing related to the current tuning step
  - Clean up temporal or tuning step specific data structures
  - Process current results and update aggregated metrics
  - Set-up data structures for the next tuning step
- Empty implementations valid
  - no post-processing required
  - single tuning step plugins

```cpp
void ExamplePlugin::finishTuningStep(void) {
    psc_dbgmsg(
        PSC_SELECTIVE_DEBUG_LEVEL(AutotunePlugins),
        "ExamplePlugin: call to processResults()\n" );
}
```
Finish Tuning Step

• The plugin indicates that it will continue with an additional tuning step, or that its tuning loop is finished
• Used in multi-step tuning strategy to search for other scenarios

```cpp
bool ExamplePlugin::tuningFinished(void) {
    return true;
}
```
Finish Tuning Step

- Generates XML output of tuning result
- Displays the best scenario based on search algorithm
- Passes an Advice object to runtime for further processing
Get Advice

- Generates XML output of tuning result
- Displays the best scenario based on search algorithm
- Passes an Advice object to runtime for further processing
Get Advice

Advice *ExamplePlugin::getAdvice(void) {  

    map<int, double> timeForScenario = searchAlgorithm->getSearchPath();  
    int optimum = searchAlgorithm->getOptimum();  

    for (int scenario_id = 0; scenario_id < pool_set->fsp->size(); scenario_id++) {  
        Scenario *sc = (*pool_set->fsp->getScenarios())[scenario_id];  
        list<TuningSpecification>* tpSpec = sc->getTuningSpecifications();  
        map<TuningParameter*, int> tpValues = tpSpec->front()->getVarName()->getValue();  
        int threads = tpValues[tuningParameters[0]];  
        double time = timeForScenario[scenario_id];  
        cout << scenario_id << "|" << threads << "|" << time << "|" << serialTime/time << endl;  
        for (scenario_iter = pool_set->fsp->getScenarios()->begin();  
            scenario_iter != pool_set->fsp->getScenarios()->end();  
            scenario_iter++) {  
            Scenario *sc = scenario_iter->second;  
            sc->addResult("Time", timeForScenario[sc->getID()]);  
        }  
    }  

    Scenario *bestScenario = (*pool_set->fsp->getScenarios())[optimum];  
    return new Advice(getName(), bestScenario,  
        timeForScenario, "Time", pool_set->fsp->getScenarios());  
}
Design Your Own Plugin
Overall Approach

• What to tune?
• How to tune?
• How to speedup the tuning?
What to search?

• Tuning parameters
  – Integer Ranges
  – Vector restrictions providing a vector of values
  – Tuning action with name of the parameter

• Tuning actions
  – Runtime tuning actions: implemented in monitor
    • Variable tuning action: assign value to variable's address
    • Function tuning action: function call with TP value
  – Other tuning actions
    • Implemented by plugin
What to search?

• Modified regions
  – Regions to which the tuning parameters are applied.
  – Single or multiple regions in the code
    • E.g. setting the number of threads for the phase region or setting it for individual parallel regions

• Tuned region
  – Region, for which the objective is evaluated.
What is the objective?

• Single or multiple objectives?
  – Single: energy delay product
  – Multiple: e.g. execution time and energy consumption

• Measurement point?
  – Tuned region and process to measure the objective
  – Typically for phase region in a single or all processes
How to search?

• Number of tuning steps
• Pre-analysis
• Search strategy
How to search?

- Number of tuning steps
  - Single tuning step as default
  - Split in a sequence of tuning steps to structure the search
    - Use different search techniques in the tuning steps
    - Enforce an order in which the tuning parameters are searched.
How to search?

• Pre-analysis
  – Use pre-analysis in a tuning step to guide the search
  – Example: Select certain tuning parameters
    • e.g. use message lengths to guide range of eager limit tuning in MPI
  – Example: Select modified regions
    • e.g. regions with significant execution time
How to search?

• Select search strategy
  – Search strategy walks the space of tuning parameters
  – Dependent tuning parameters (cross product)
    • exhaustive
    • genetic
    • random
    • Random enhanced by machine learning
  – Independent tuning parameters
    • individual
How to speed up the search?

• Use phase region
  – Assumeshes similar behavior of application phases

• Scenario analysis to gather effect for multiple regions
  – Measure the effect of a tuning parameters for individual regions

• Genetic or random search instead of exhaustive
  – Enhanced via machine learning

• Combine scenarios in a single experiment

• Distribute scenarios across processes
Phase Region Tutorial
Agenda

• Avoid executing the entire run for a tuning experiment
  – Very time consuming
  – Frequently not necessary since iterations of progress loop have similar characteristic

• Mark body of the simulation progress loop as phase region
  – User Region directive
  – Each execution of the user region is a phase

• Experiment will be performed in a single phase
  – Multiple iterations used to go through different scenarios
  – Significant speedup of search
Adding User Region

```plaintext
!---------------------------------------------------------------------!
c      start the benchmark time step loop
!---------------------------------------------------------------------!

do  step = 1, niter

   if (mod(step, 20) .eq. 0 .or. step .eq. 1) then
      write(*, 200) step
200        format(' Time step ', i4)
   endif

!$MON user region
   call exch_qbc(u, qbc, nx, nxmax, ny, nz)
   do zone = 1, num_zones
      call adi(rho_i(start1(zone)), us(start1(zone)),
               vs(start1(zone)), ws(start1(zone)),
               qs(start1(zone)), square(start1(zone)),
               rhs(start5(zone)), forcing(start5(zone)),
               u(start5(zone)),
               nx(zone), nxmax(zone), ny(zone), nz(zone))
   end do
!$MON end user region

end do
```
Analysis & Region Selection Plugin
Agenda

• Using PTF analysis to detect areas of interest
  – Locating areas of interest in the code
  – Understanding use of regions to accelerate the search

• Handling multiple Regions
  – Acquiring & processing region information
  – Using PTF pre-analysis feature
  – Filtering performance dominant regions
Tuning Problem

• Tuning parameters can be applied to multiple regions
  – These regions are included in the phase region
  – Important are regions with high impact

• Approach
  – Select high impact regions via the pre-analysis functionality of PTF
Implementation

- **Step 1**: Get list of available regions while starting tuning step

- **Step 2**: Request pre-analysis

- **Step 3**: PTF will gather the performance data

- **Step 4**: Create scenarios for the maximum time consuming region
Find Regions

- `get_regions()` method of an application object returns the list of regions.
- Type of each region is checked `get_type()` function.
- Regions with a type `PARALLEL_REGION` are selected to be analyzed.
void TutMultipleRegions::startTuningStep(void) {

std::list<Region*> code_regions;
code_regions = appl->get_regions();
std::list<Region*>::iterator region;

// iterating over all regions
int count = 0, parallel_regions = 0;

for (region = code_regions.begin(); region != code_regions.end(); region++) {
    if ((*region)->get_type() == PARALLEL_REGION) {
        parallel_regions++;
        code_region_candidates[(*region)->getIdForPropertyMatching()] = *region;
    }
}

psc_dbgmsg(PSC_SELECTIVE_DEBUG_LEVEL(AutotunePlugins),
            "TutMultipleRegions: found %d parallel regions (out of %d total).\n",
            parallel_regions, count);
}
Requesting Pre-analysis

- The pre-analysis requires a `StrategyRequest` object formed using
  - A `StrategyRequestGeneralInfo` object with information about the analysis strategy to be used
  - A `PropertyRequest` object that determines which performance properties to be requested. E.g. Execution Time.
- Return `true` from `analysisRequired()` in order to request an analysis
bool TutMultipleRegions::analysisRequired(StrategyRequest** strategy) {
    std::map<string, Region*>::iterator region;

    StrategyRequestGeneralInfo* analysisStrategyInfo = new StrategyRequestGeneralInfo;
    analysisStrategyInfo->strategy_name = "ConfigAnalysis";
    analysisStrategyInfo->pedantic = 0;
    analysisStrategyInfo->delay_phases = 0;
    analysisStrategyInfo->delay_seconds = 0;

    PropertyRequest *req = new PropertyRequest();
    req->addPropertyID(EXECTIME);

    for (region = code_region_candidates.begin(); region != code_region_candidates.end();
         region++) {
        req->addRegion(region->second);
    }

    req->addAllProcesses();

    list<PropertyRequest*>* reqList = new list<PropertyRequest*>;
    reqList->push_back(req);
    *strategy = new StrategyRequest(reqList, analysisStrategyInfo);
    return true;
}
Performing Analysis

- This step is done by the PTF runtime system
- It gathers the performance data requested using the `StrategyRequest` object
- The data is later processed by the plugin and/or search strategies
Create Scenarios

- Select the longest running region using data collected in pre-analysis
- Set up a SearchSpace to pass to the search algorithm
- Create a search space with a tuning parameter that represents OpenMP threads
- Delegate scenario creation and evaluation to the search strategy
void TutMultipleRegions::createScenarios(void) {
  ...
  double severity = 0;
  SearchSpace *searchspace = new SearchSpace();
  VariantSpace *variantSpace = new VariantSpace();
  for (int i = 0; i < tuningParameters.size(); i++) {
    variantSpace->addTuningParameter(tuningParameters[i]);
  }
  ...

  selected_region =
    code_region_candidates[longest_running_property.getRegionId()];
  searchspace->addRegion(selected_region);
  searchAlgorithm->addSearchSpace(searchspace);
  searchAlgorithm->createScenarios();
}
void TutMultipleRegions::createScenarios(void) {
    list<MetaProperty>::iterator property;
    MetaProperty longest_running_property;
    list<MetaProperty> found_properties =
        pool_set->arp->getPreAnalysisProperties(0);
    ...

    for (property = found_properties.begin(); property != found_properties.end(); property++){
        if (property->getSeverity() > severity){
            severity = property->getSeverity();
            longest_property = *property;
        }
    }

    selected_region = code_region_candidates[longest_property.getRegionId()];
    searchspace->addRegion(selected_region);
    ....
}
Define Experiment

- Select each scenario, set its objective property to `EXECTIME` in the previously selected region at rank 0, and then move it to the experiment scenario pool.

```c
void TutMultipleRegions::defineExperiment(int numprocs, bool& analysisRequired, StrategyRequest** strategy) {
    Scenario *scenario = pool_set->psp->pop();
    scenario->setSingleTunedRegionWithPropertyRank(selected_region, EXECTIME, 0);
    pool_set->esp->push(scenario);
}
```
Get Advice

• Present the results of the tuning to the user
  – Optimal number of threads
  – Number of threads evaluated and their scaling factor

• Generate output to the screen as well as in XML format
  – The XML conforms to a distributed schema
    • Can be used for processing with external tools
    • Contains complete and detailed results
Advanced Concepts and Tutorial Plugins
Additional Tutorial Plugins

The following tutorials are available:

- OpenMP Scalability Basic Tutorial (covered today)
- Analysis and Region Selection Tutorial (covered today)
- Vector Range Restriction Tutorial
- Basic Regions Tutorial
- Scenario Execution Tutorial
- Search and Objective Functions Tutorial
- Tuning Parameter Cross-Product Tutorial
- Genetic Search Tutorial
- Scenario Analysis Tutorial
- Multiple Tuning Steps Tutorial
- Multiple Objective Tutorial
- Combined Scenarios Tutorial
Available with PTF Sources

• Documentation for the tutorial plugins
  – Install PTF with developers mode enabled
    --enable-developer-mode
  – Build and install documentation
    • make doc
    • make doc-install

• Access documentation at periscope.in.tum.de
  – http://periscope.in.tum.de/releases/latest/doxygen
  – For the tutorials, follow the “Tutorial” link at the top of the main doxygen page.
Advanced Tuning Parameters

- Tuning Parameters
- Vector restrictions providing a vector of values
  - Vector Range Restriction Tutorial
  - Uses restriction for power of 2 values.
Own Objective Functions

• Default objective function
  – Max or min on the severity of a single property returned for the scenario.
  – Any property can be used.

• Plugin developer can provide own objective function
  – It can for example combine the severities of multiple returned properties.
  – For example: max/min for energy delay product
  – Search and Objective Functions Tutorial
Multiple Objectives

• PTF provides support for multiple tuning objectives
• Combine multiple tuning objectives in a plugin
  – Request multiple properties for the tuned region
  – Determine the best scenario at advice generation
    – Multiple Objectives Tutorial
• Perform multi-objective tuning
  – Choose genetic search with multiple objectives
    – Genetic Search Tutorial (only genetic search not multi-objective)
Multiple Tuning Steps

• Plugins can make use of multiple tuning steps with pre-analysis, creation of scenarios ....
• Each tuning step could optimize a single tuning parameter, for example, e.g., the Master Worker Plugin
• Multiple Tuning Steps Tutorial: The selected parallel regions are optimized in individual tuning steps.
Speeding up Tuning

• Scenario analysis
• Heuristic search algorithms
• Evaluation of multiple scenarios in a single experiment
• Parallel evaluation of scenarios
Scenario Analysis

• PTF allows to apply a performance analysis strategy for a tuning scenario, e.g., MPI analysis

• This is used frequently to measure the effect of tuning decisions on different regions, e.g., execution time with a number of threads.

• Scenario Analysis Tutorial illustrates, how to measure the execution time of multiple parallel regions with the configurable analysis strategy.
Heuristic Search Algorithms

• For big search spaces use heuristics.
• For example, the cross-product of parallel regions and number of threads. The Tuning Parameter Cross-Product Tutorial illustrates this.

• Heuristic search algorithms
  – Genetic search and random search

• Genetic Search Tutorial illustrates genetic search for the cross-product tuning of parallel regions.
Evaluate Multiple Scenarios in Single Experiment

• If tuning scenarios are independent, they can be combined into a single experiment.

• The Combined Scenarios Tutorial illustrates this.
Parallel Evaluation of Scenarios

• Scenarios can be assigned to different processes and thus be evaluated in parallel in a single experiment.

• The Scenario Execution Tutorial illustrates this for assigning scenarios with different number of threads to multiple processes.
  – Assumption: no MPI in a parallel regions
Tutorial Material

Public Git repository:
• http://periscope.in.tum.de/git-releases

Doxygen documentation:
• http://periscope.in.tum.de/releases/latest/doxygen
  • For the tutorials, follow the “Tutorial” link at the top of the main doxygen page.

PTF’s documentation:
• http://periscope.in.tum.de/releases/latest/pdf

Today’s presentations:
• http://periscope.in.tum.de/tutorials/sc2015

Source code in tar packages:
• http://periscope.in.tum.de/releases/latest/tar
Detailed Information about PTF

• Book describing the PTF approach, architecture and tuning plugins
  
  **Automatic Tuning of HPC Applications: The Periscope Tuning Framework**
  Editors: Michael Gerndt, Eduardo César, Siegfried Benkner
  Shaker Verlag, 2015
  ISBN 978-3-8440-3517-9

• Available online at periscope.in.tum.de
Focusing on runtime tuning for energy efficiency.

Starting date: September 1st 2015

Duration: 3 years
Thank you!
More Questions?