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

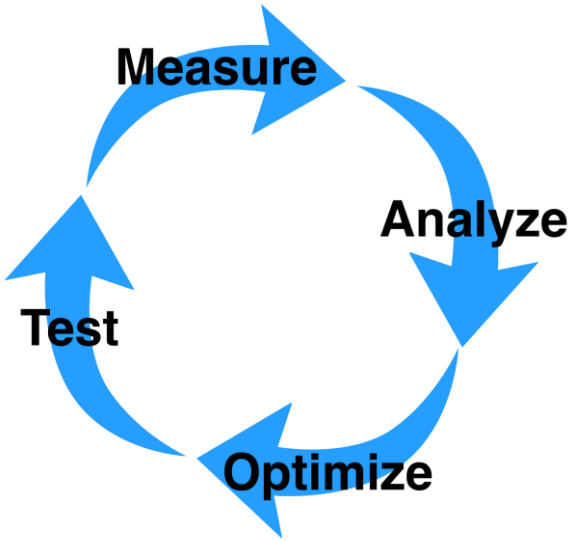
Technische Universität München

Overview

- Periscope Tuning Framework (PTF)
  - Architecture
  - Plugins
  - Installation & Setup
- Plugin Development

Tuning Stages

- **Measure**
  - Application tuning runs
  - Performance data collection
  - Identify metrics
- **Analyze**
  - Find inefficiencies
- **Optimize**
  - Try optimization
  - User knowledge
- **Test**
  - Re-evaluate

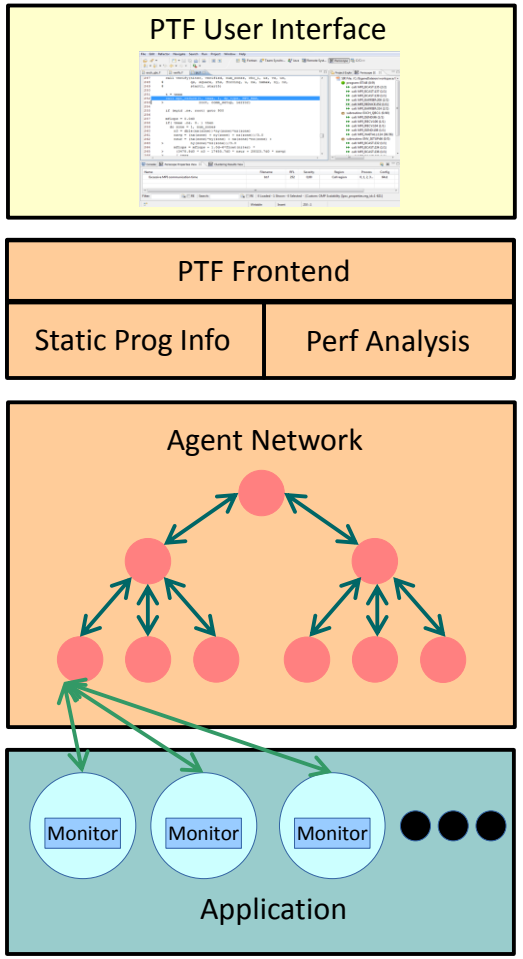


Periscope Tuning Framework

- Objective - Single tool for **performance analysis** and **tuning**
- Extends Periscope with a tuning framework
- **Tuning plugins** for performance and energy efficiency tuning
- **Design time** tuning
- Combine multiple tuning techniques

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



Performance Analysis

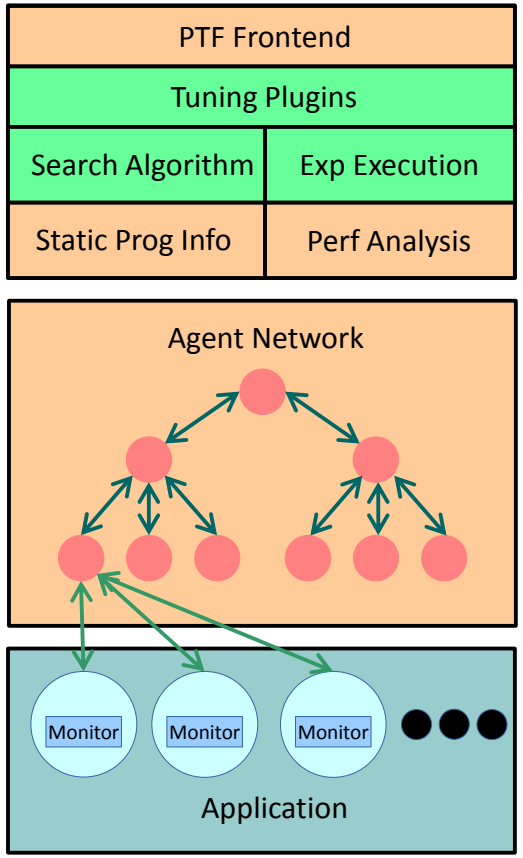
- Performance properties
  - Formalize what a performance problem is based on measurable metrics.
  - Return severity
- Analysis strategies
  - Define how to go through the search space of properties and regions.
- Analysis agent
  - Executes analysis strategy and returns found performance properties
  - Independent of other agents
  - Configures the monitor linked to the application and retrieves measurements.

PTF Tuning Plugins

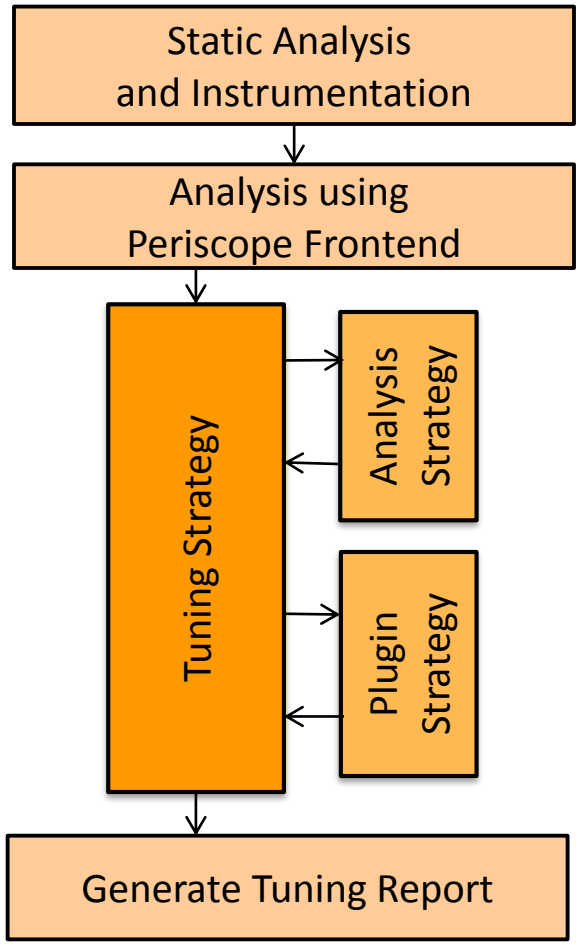
- Plugins (online and semi-online)
  - Compiler based optimization
  - HMPP tuning for GPUs
  - Parallel pattern tuning
  - MPI tuning
  - Energy efficiency tuning
  - User defined plugins - info in developer session

Periscope Tuning 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



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

Name	Tuning Parameters	Objective
Compiler Flag Selection (CFS)	Compiler flag combinations	Optimize performance
Dynamic Voltage and Frequency Scaling (DVFS)	CPU Frequency and power states	Reduce energy consumption with minimal impact
MPI Runtime	MPI parameters	Optimize SPMD MPI application performance
PCAP	Number of threads	Reduce energy, improve performance

Compiler Flag Selection (CFS)

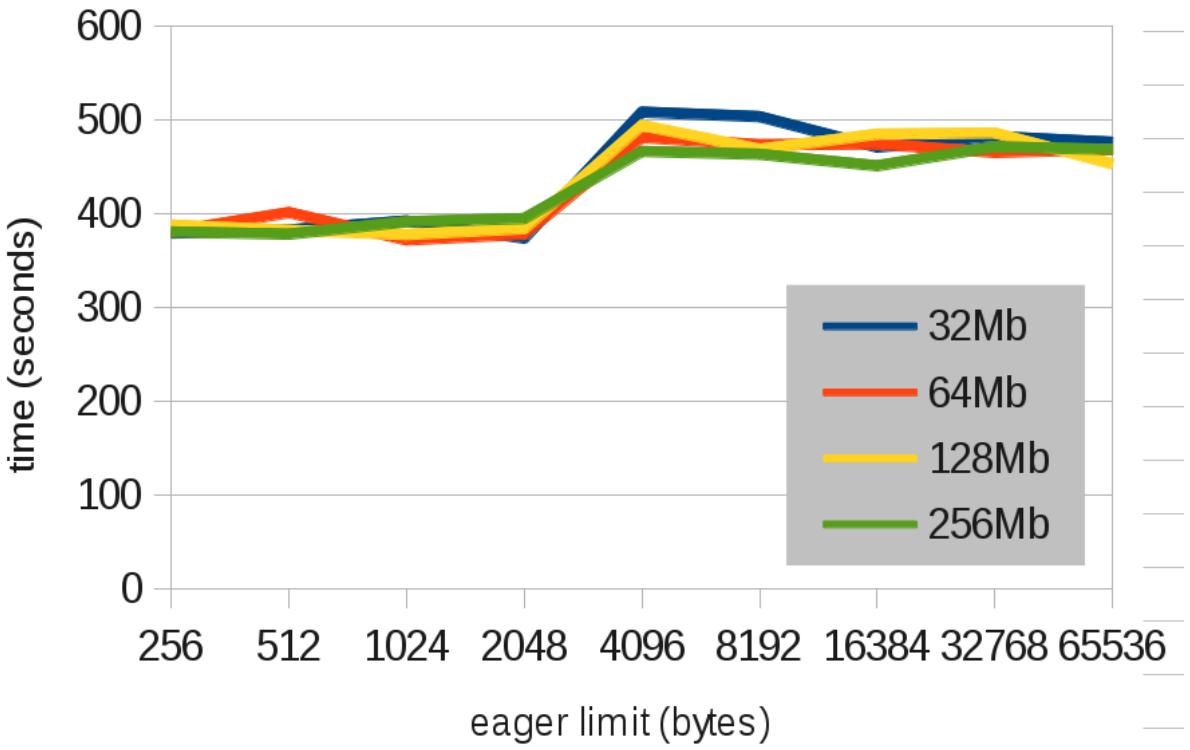
- **Goal**
  - Optimize application performance by guiding machine code generation using compiler flags
- **Tuning Technique**
  - Selection of compiler flags and corresponding values
    - -O1, O2, -floop-unroll
  - Search through combination of flags

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

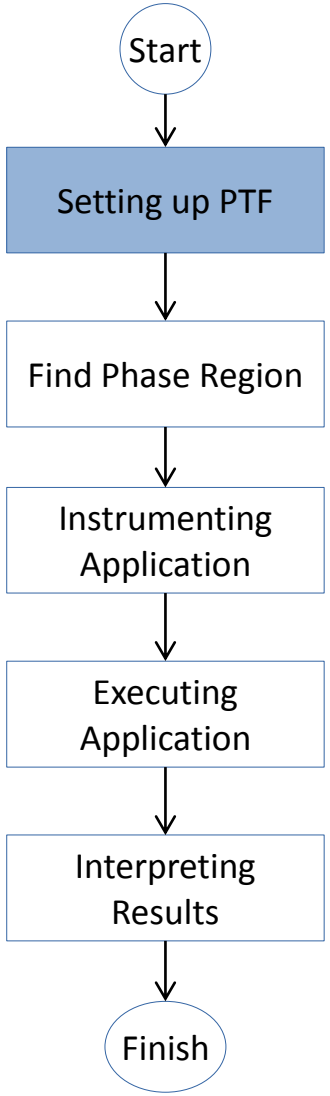
MPI Runtime

- **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



Using PTF

- Identifying Phase Regions
  - Phase region marked as user region
  - Entire application is default phase region
- Instrumenting the application
  - Use `score_p`
- Executing application
  - Use periscope frontend for execution
  - Use `psc_frontend` command
- Interpreting the results

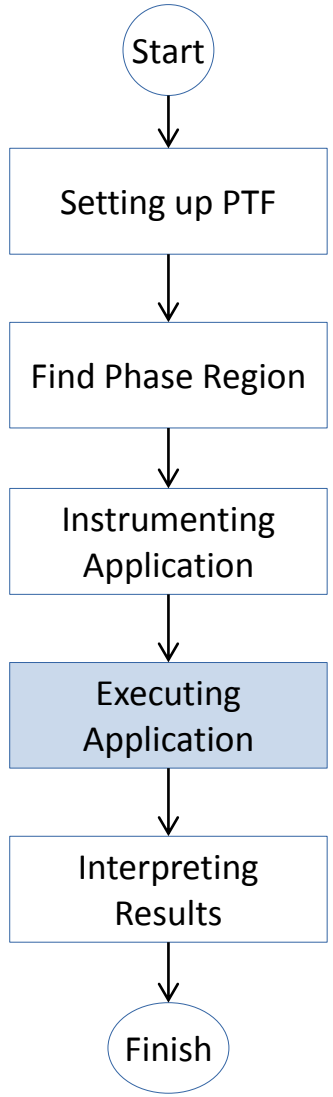


Executing Application

Prepare a config file

cfs_config.cfg

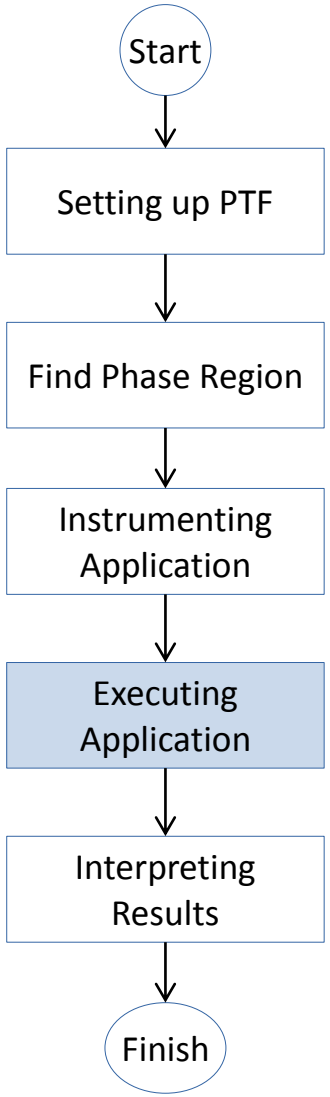
```
// ********* application related settings **********
// the path to the Makefile
makefile_path="../";
// the variable containing the build flags
makefile_flags_var="FFLAGS";
// arguments for the make command
makefile_args="BT-MZ CLASS=C TARGET=BT-MZ";
// path to the source files of the application
application_src_path="../BT-MZ";
// ***********************************************

// ********* plugin related settings ***************
//Details about the selective make
selective_file_list="x_solve.f y_solve.f z_solve.f";
make_selective="true";
// the desired search algorithm: exhaustive or individual
search_algorithm="exhaustive";
// the compiler flags to be considered in the search
tp "Opt" = "-" ["O1", "O2", "O3"];
// ***********************************************
```



Executing Application

- psc_frontend command is used to execute PTF
- --tune=compilerflags to select the plugin

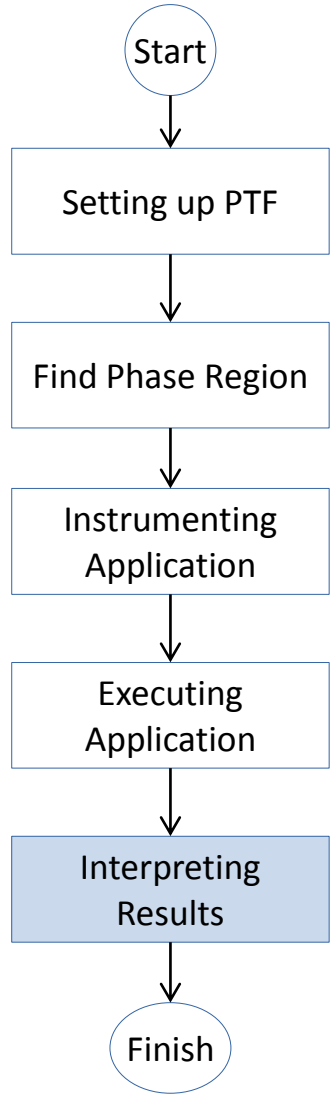
```
$ psc_frontend --apprun="./bt-mz.C.x"     \
 --starter=FastInteractive --delay=2      \
 --mpinumprocs=1 --tune=compilerflags     \
 --force-localhost                        \
 --debug=2 --selective-debug=AutotuneAll  \
 --sir=bt-mz.C.x.sir
```



Interpreting the results

Combination executing in minimal time is reported as the optimal scenario

```
Optimum Scenario: 1

Compiler Flags tested:
Scenario 0 flags: "-O1 "
Scenario 1 flags: "-O2 "
Scenario 2 flags: "-O3 "


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!
```



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

PTF Tutorial for Plugin Developers

Michael Gerndt

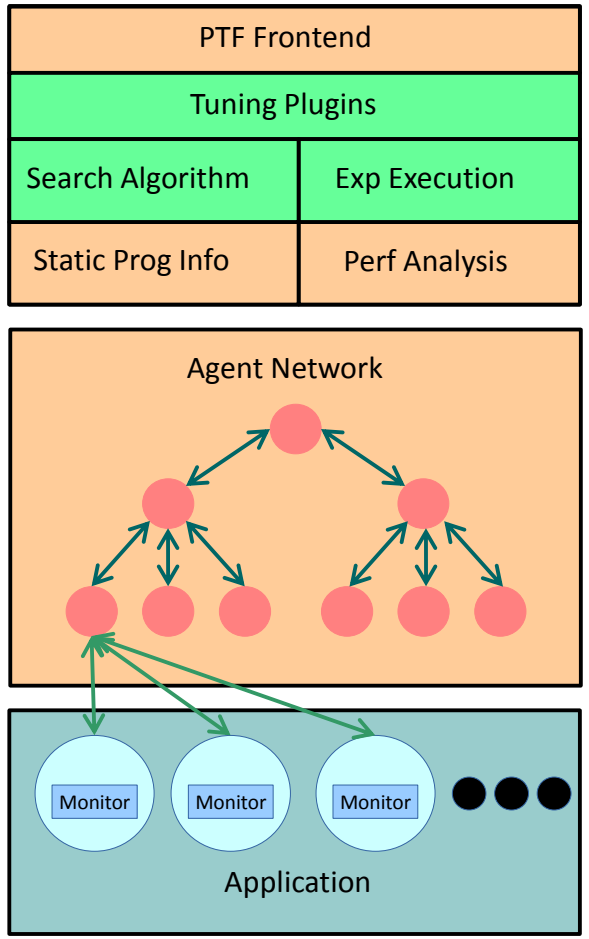
gerndt@in.tum.de

Overview

- PTF Architecture
- PTF Components
  - Tuning Parameters
  - Search Strategies
- PTF Plugin workflow
- Plugin Tutorial
  - OpenMP Scalability Plugin Tutorial
  - Analysis & Region Selection Tutorial

PTF Components

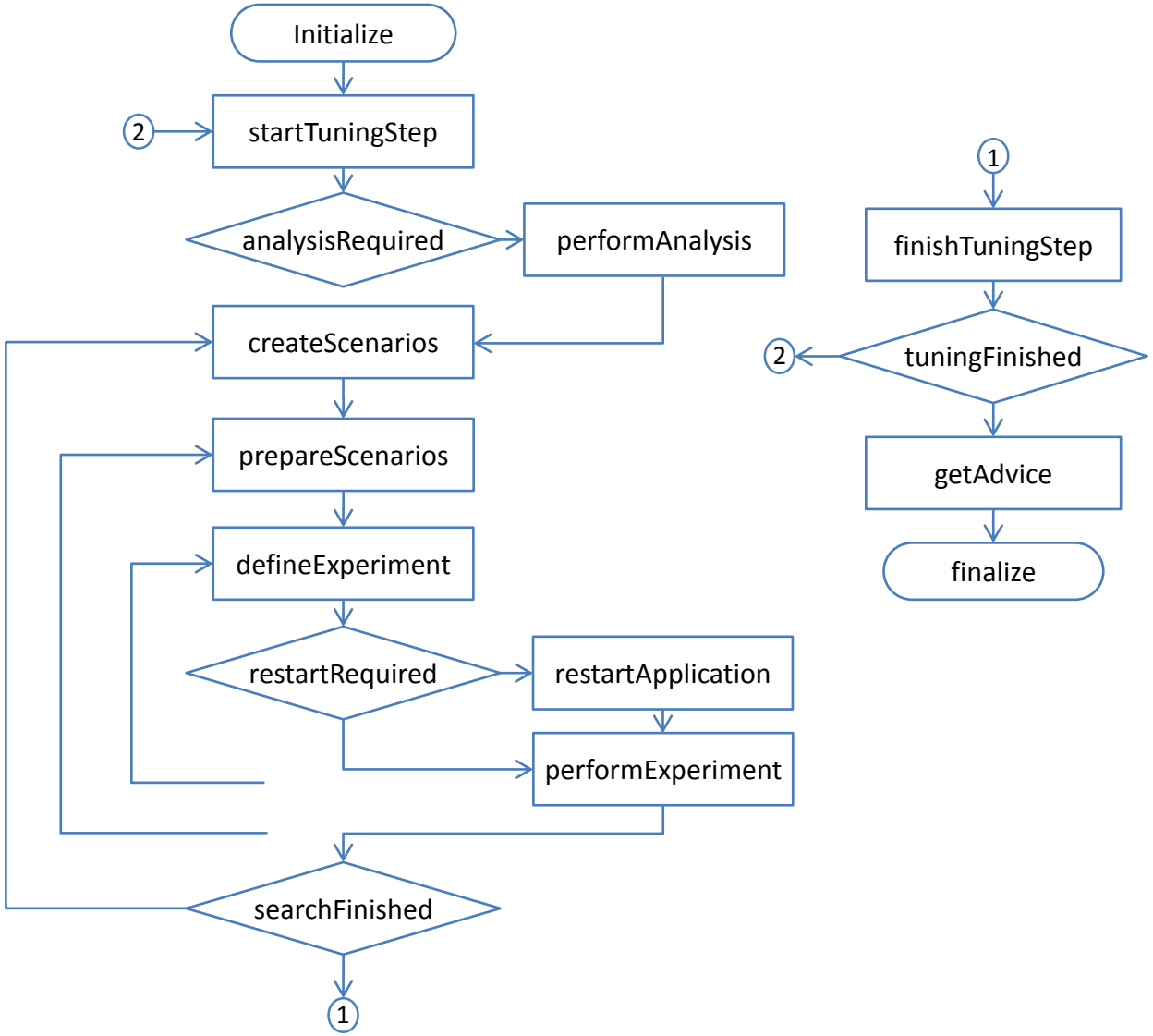
- **Tuning Parameters**
  - Aspects that can be changed and affect performance
  - Can be plugin specific
  - Semantics are strictly defined in each plugin
- **Agents**
  - Collects performance metrics from the MRI Monitor
  - Generate performance properties
  - Propagate properties to the frontend and finally to the plugin
- **MRI Monitor**
  - Collect the performance metrics
  - Support for different runtime systems



Tuning Plugin

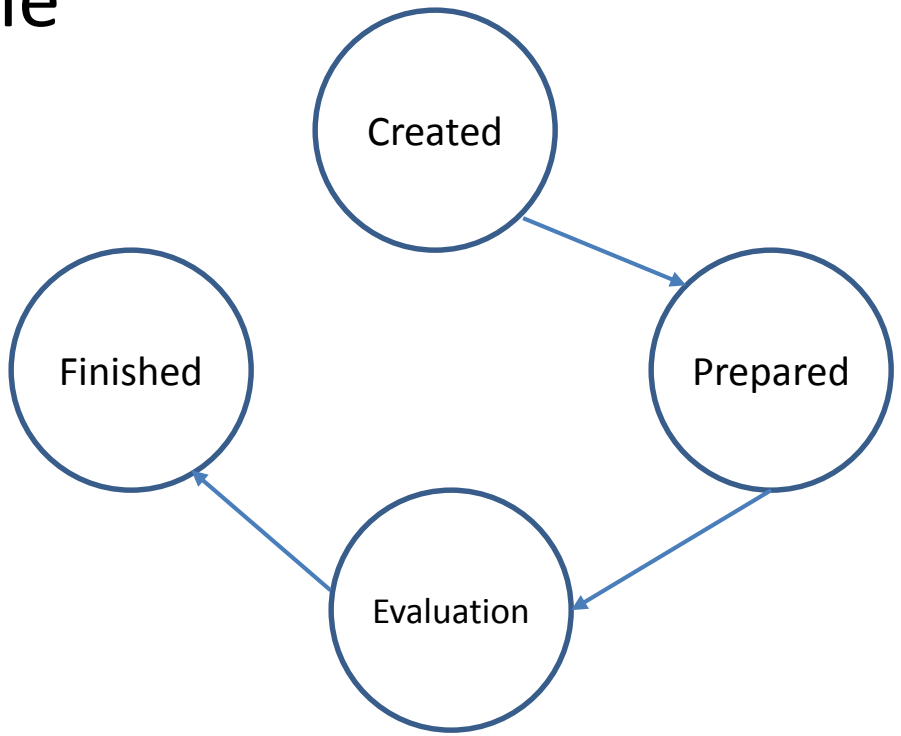
- Codifies the expert knowledge to automate the tuning process
- Interfaces with PTF and uses its infrastructure to collect performance data for tuning the application
- To interact with PTF it is required that the plugins implement the **Tuning Plugin Interface** (TPI)

Plugin Workflow



Scenarios: READEX configuration

- Scenarios specify concrete setting for tuning parameters
  - One point in the search space
- Scenario life cycle



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 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

Iplugin Interface

```
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;
};
```

OpenMP Scalability Plugin Tutorial

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

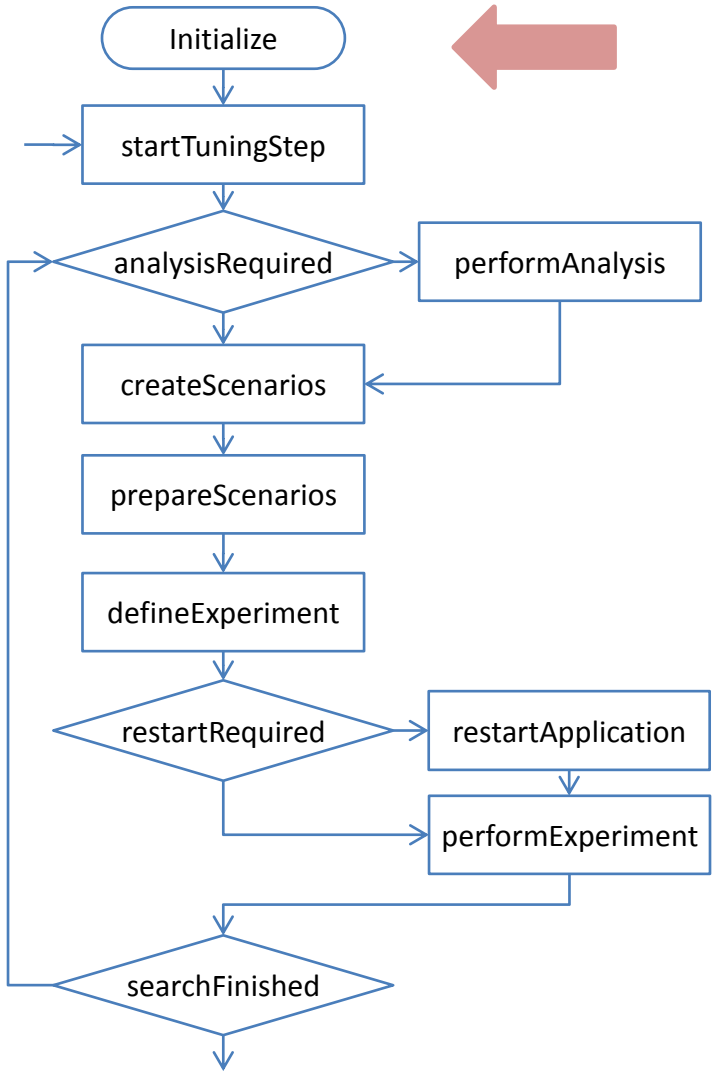
- `int getVersionMinor(void);`
  Returns the minor version number of the plugin

- `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 runtime system to offer services to the plugins
- pool_set contain the scenario and the results pool
- Define the tuning parameters
- Define the search algorithm



Plugin Initialization

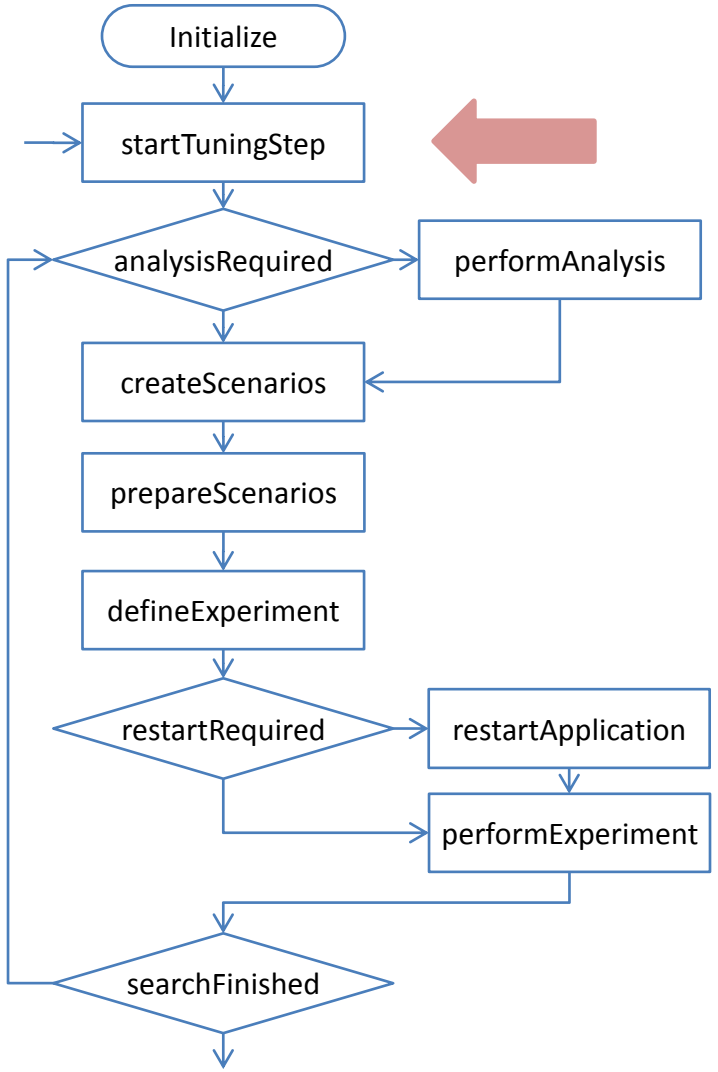
```
void ExamplePlugin::initialize(DriverContext *context, ScenarioPoolSet *pool_set) {
  this->context = context;
  this->pool_set = pool_set;
  TuningParameter *numberOfThreadsTP = new TuningParameter();
  numberOfThreadsTP->setId(0);
  numberOfThreadsTP->setName("NUMTHREADS");
  numberOfThreadsTP->setPluginType(ExamplePlugin);
  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);
  }
}
```

Create Tuning Parameter

Create Search Strategy

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



Start Tuning Step

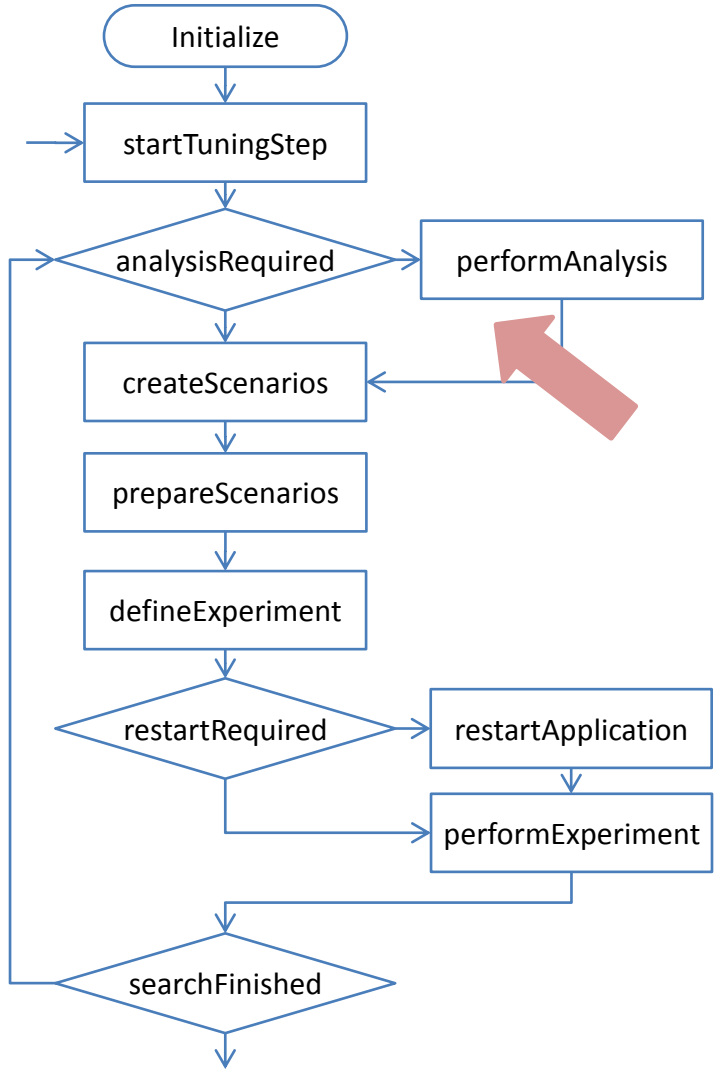
```
void ExamplePlugin::startTuningStep(void)
{
  VariantSpace *variantSpace=new VariantSpace();
  SearchSpace *searchSpace=new SearchSpace();

  for (int i = 0; i < tuningParameters.size(); i++)
  {
    variantSpace->addTuningParameter(tuningParameters[i]);
  }
  searchSpace->setVariantSpace(variantSpace);
  searchSpace->addRegion(appl->get_phase_region());
  searchAlgorithm->addSearchSpace(searchSpace);
}
```

Create Search Space

Analysis Required

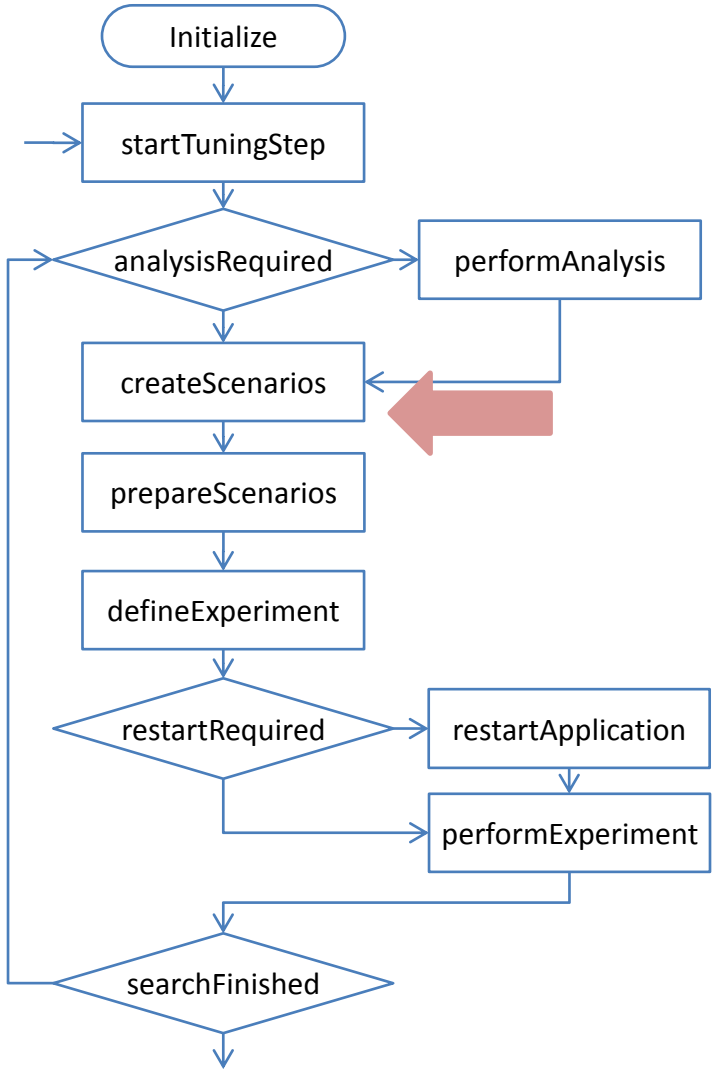
- In this step a performance analysis strategy could be triggered to get performance info
- Can be used to decide the further execution of plugin
  - e.g. Used to predict the optimal frequency for the DVFS plugin

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



Creating Scenarios

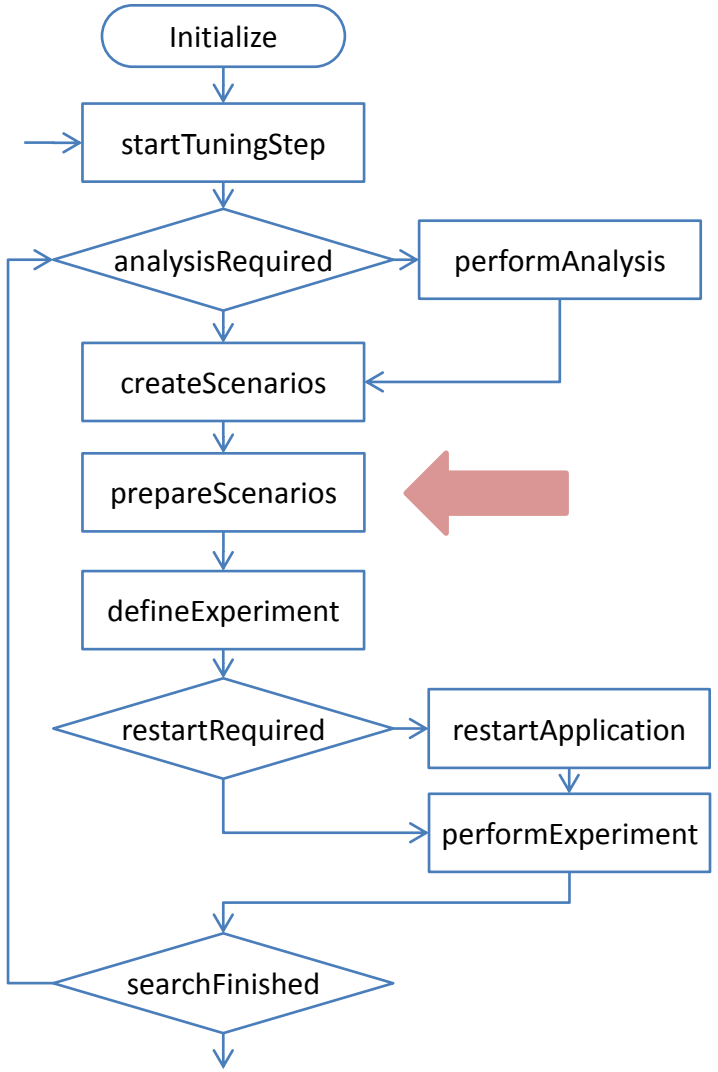
- Create scenarios from all possible combinations of the tuning parameters in the search space.




```
void ExamplePlugin::createScenarios(void)
{
  searchAlgorithm->createScenarios();
}
```

Prepare Scenarios

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

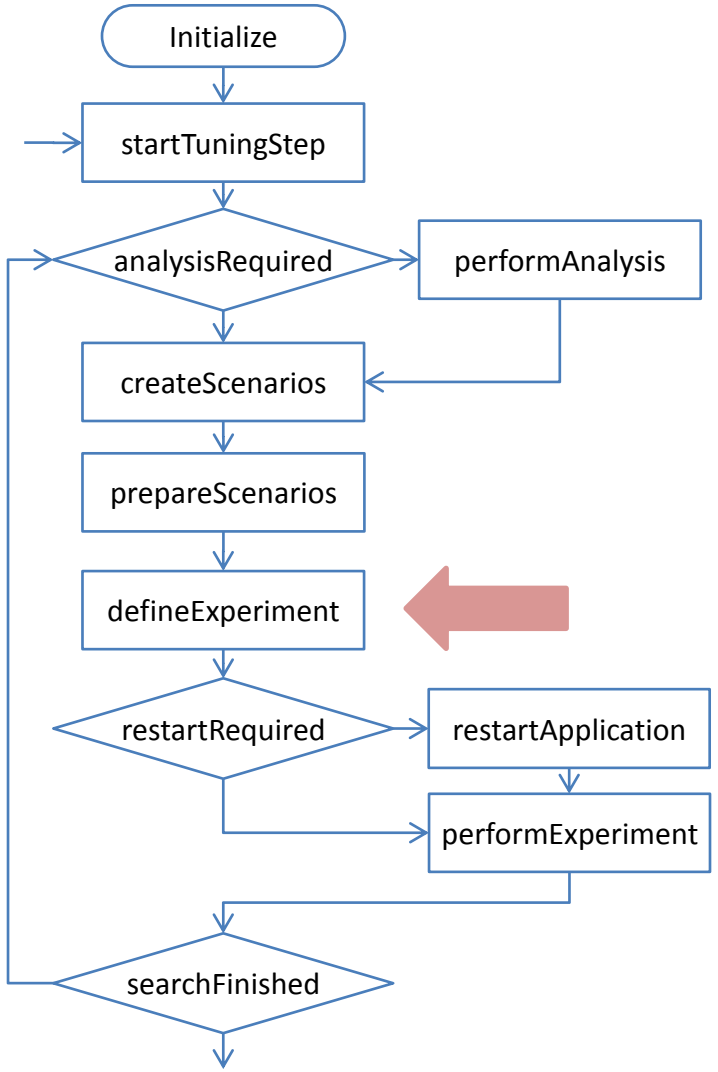



```
void ExamplePlugin::prepareScenarios(void)
{
  while(!pool_set->csp->empty()) {
    pool_set->psp->push(pool_set->csp->pop());
  }
}
```

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 are evaluated through experiments

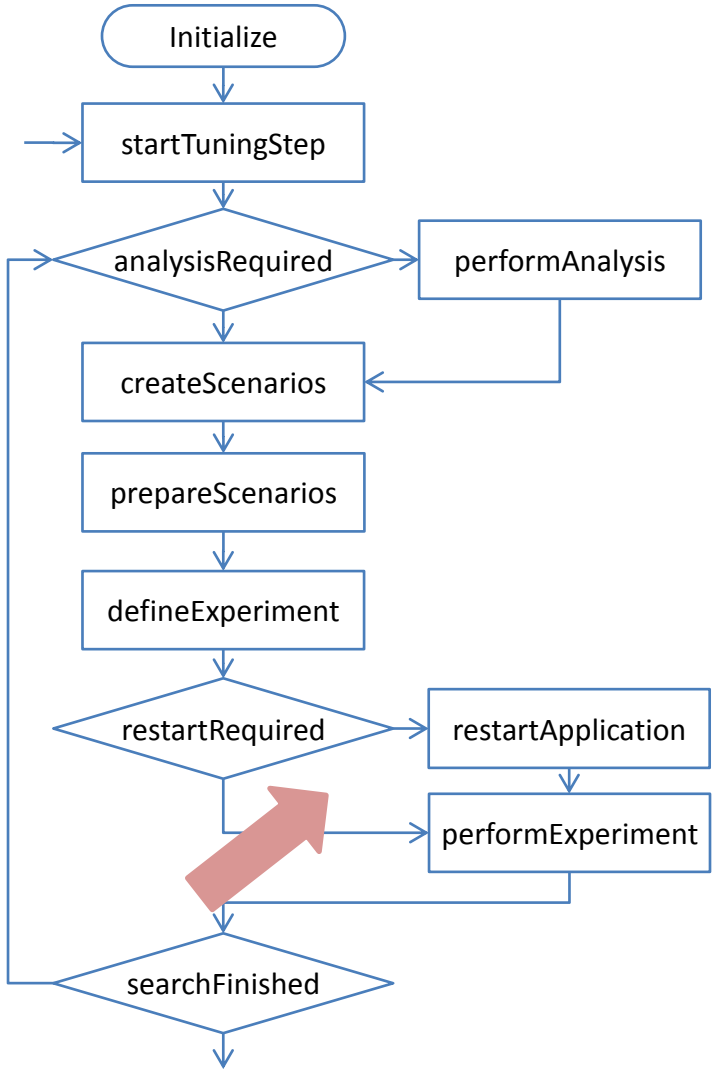
```
void ExamplePlugin::defineExperiment(...)
{
    Scenario *scenario = pool_set->psp->pop();

    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

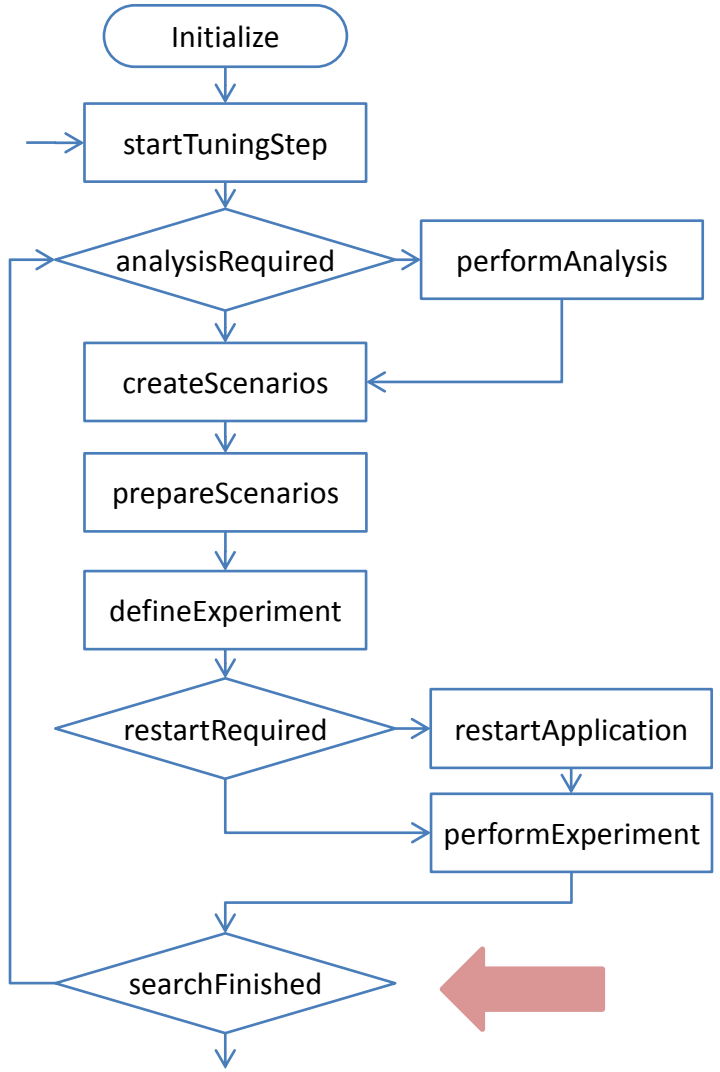
```
bool ExamplePlugin::restartRequired(
        std::string& env, int& numprocs,
        std::string& command,
        bool& is_instrumented)
{
  return false;
}
```



Search Finished

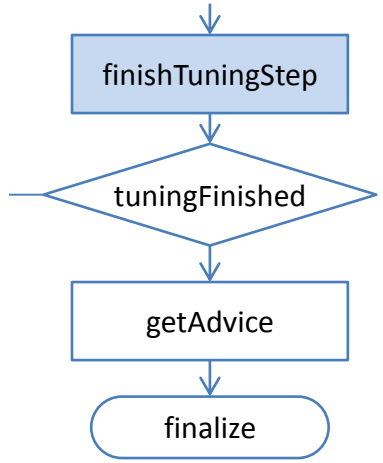
- Step determines if the search is finished
- It can be delegated to search algorithm to decide whether the search is finished

```
bool ExamplePlugin::searchFinished(void)
{
    return searchAlgorithm->searchFinished();
}
```



Finish Tuning Step

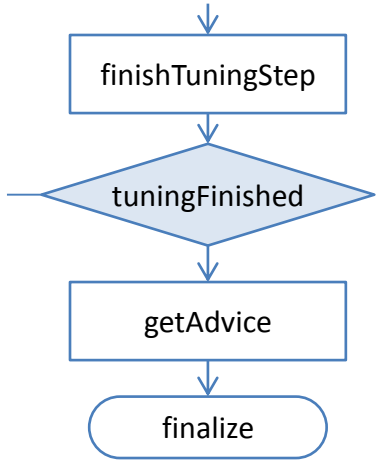
- Perform any post-processing related to the current tuning step
  - Clean up temporal or tuning step specific data structures
  - Process current results and update aggregated metrics
  - Set-up data structures for the next tuning step
- Empty implementations valid
  - no post-processing required
 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




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

Finish Tuning Step

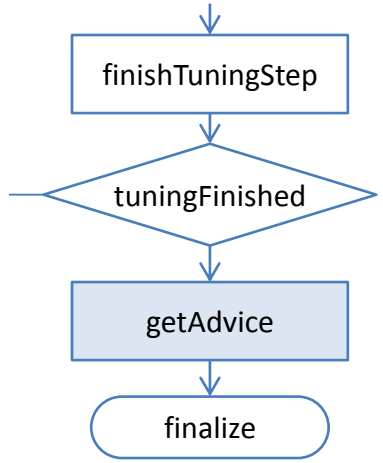
- The plugin indicates that it will continue with an additional tuning step, or that its tuning loop is finished
- Used in multi-step tuning strategy to search for other scenarios




```
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

```
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*>* tuningSpec = sc->getTuningSpecifications();
    map<TuningParameter*,int> tpValues = tuningSpec->front()->getVariant()->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());
}
```

Populate the results

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

AUTOTUNE
www.autotune-project.eu

BT-MZ/bt.f

```
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
 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
```

User Region annotated by pragmas

Analysis & Region Selection Tutorial

Agenda

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

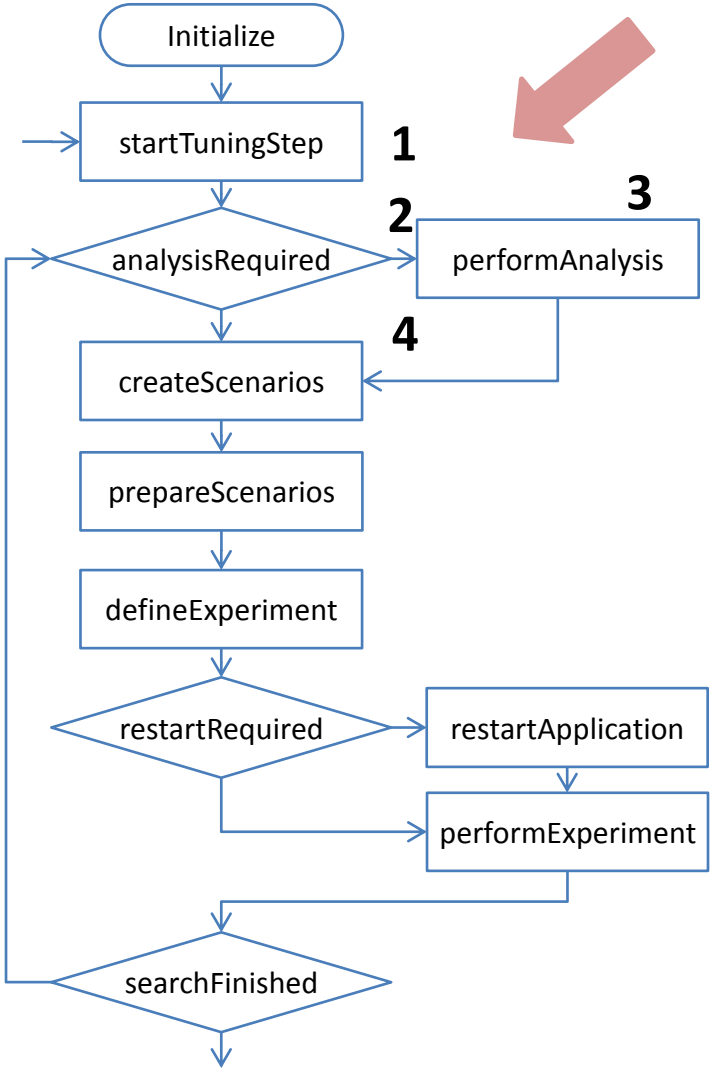
- Handling multiple Regions
  - Acquiring & processing region information
  - Using PTF pre-analysis feature
  - Filtering performance dominant regions

Tuning Problem

- Available regions can be selected
  - These regions are dynamically nested under the user region
  - Some regions have a bigger performance impact
    - Compute loops,
    - MPI operations,
    - OpenMP parallel regions, etc.
- Approach
  - Using pre-analysis functionality of PTF to find a region having maximum execution time

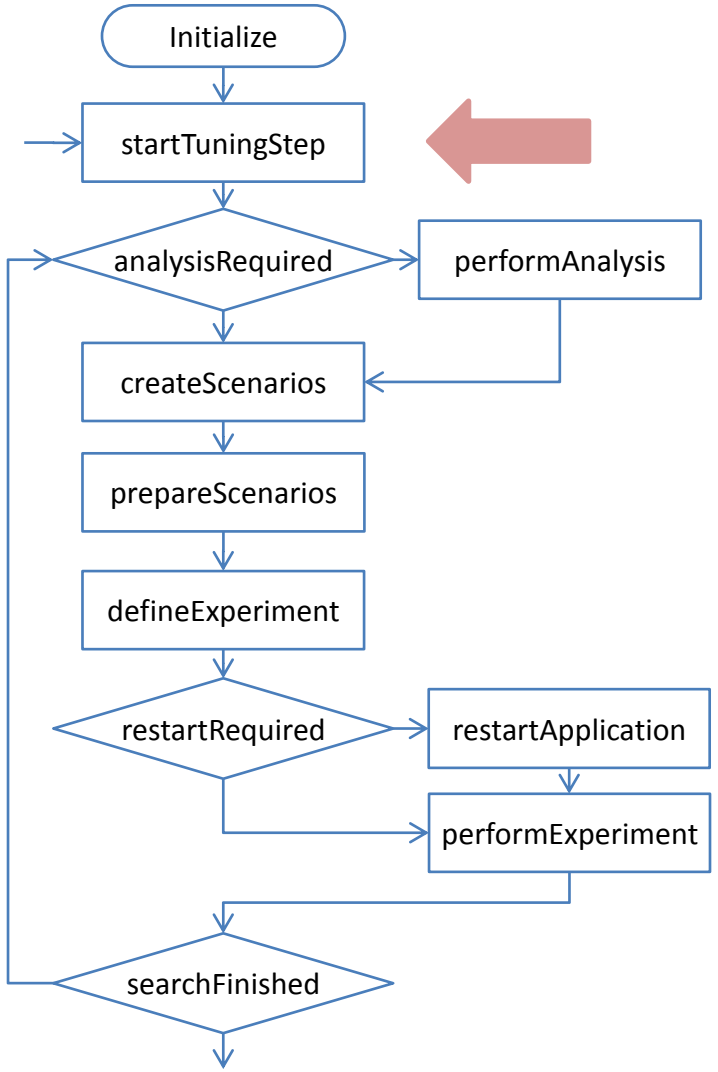
Implementation

- **Step1**: Get list of available regions while starting tuning step
- **Step2**: Request pre-analysis
- **Step3**: 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

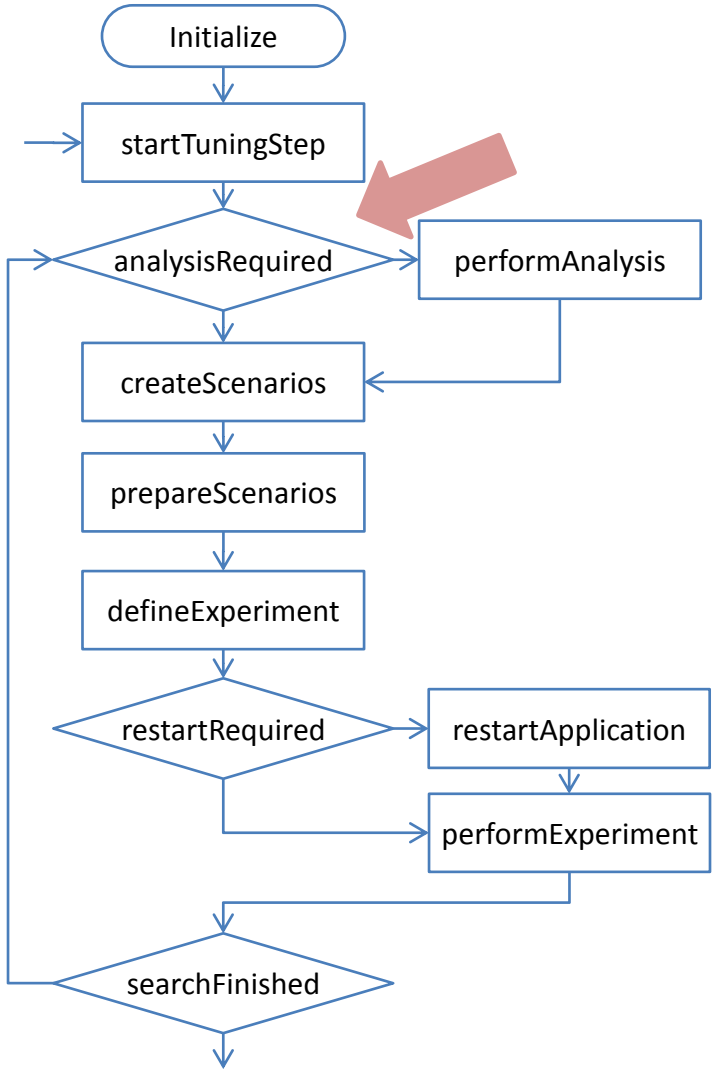


Find Regions


```
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



Requesting Pre-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;
```

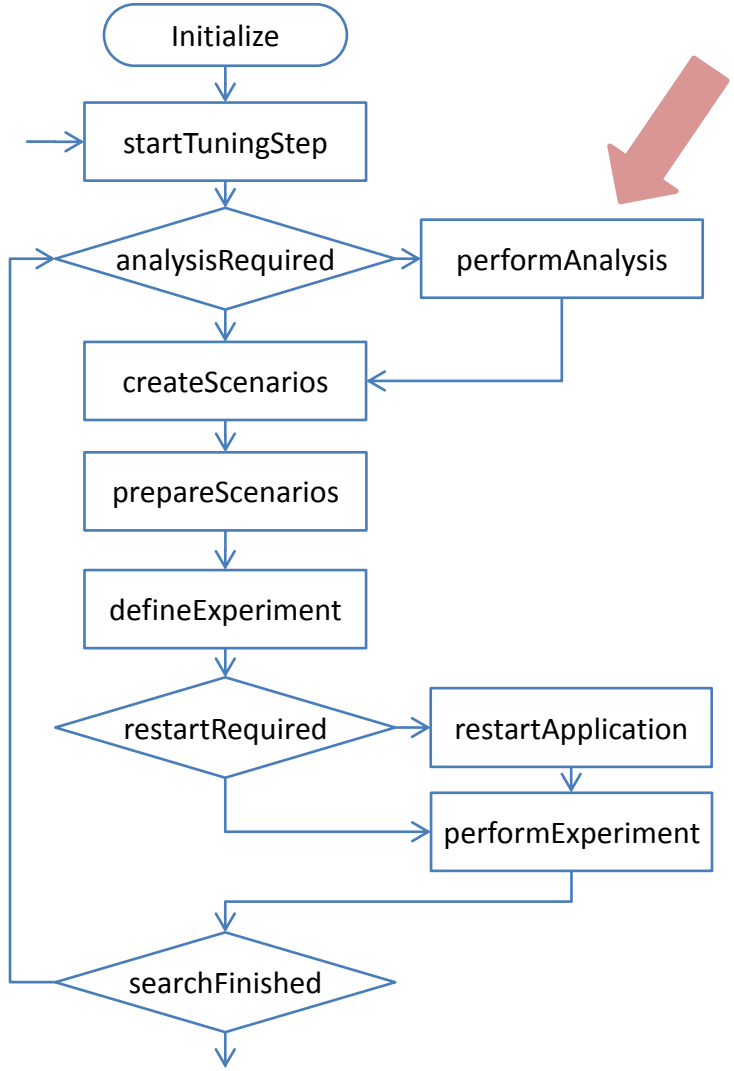
Configuring Analysis Strategy Request

```
  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;
}
```

Configuring Property to be requested

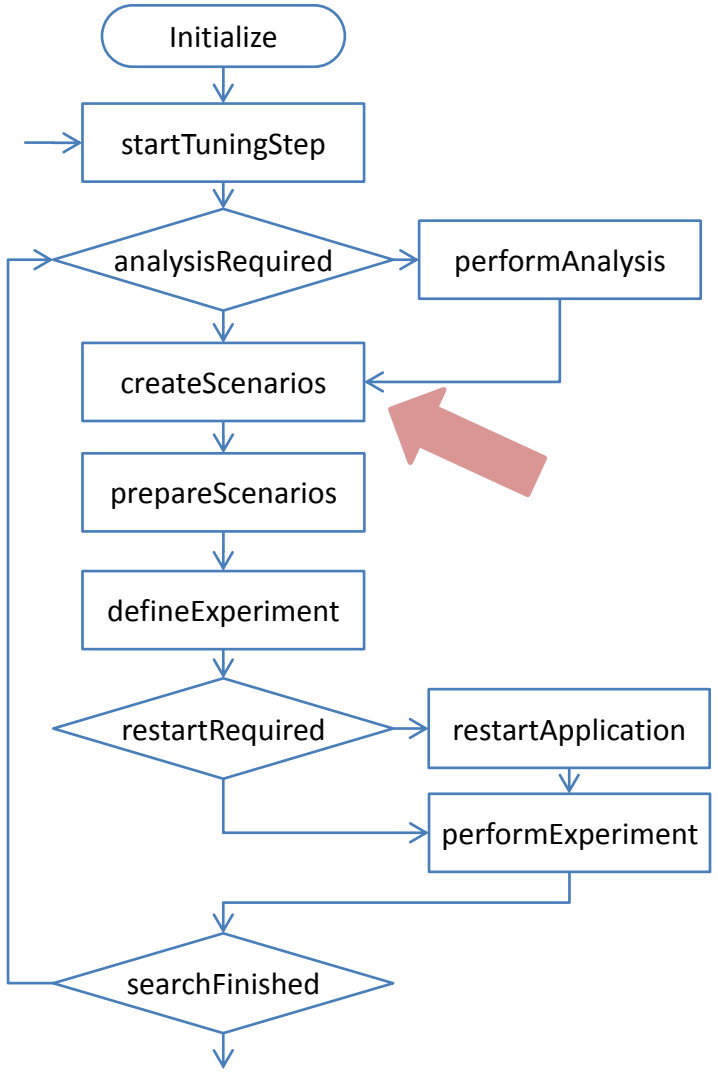
Performing Analysis

- 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



Post Postprocessing Analysis Results

AUTOTUNE
www.autotune-project.eu

```
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();
}
```

Post Postprocessing Analysis Results

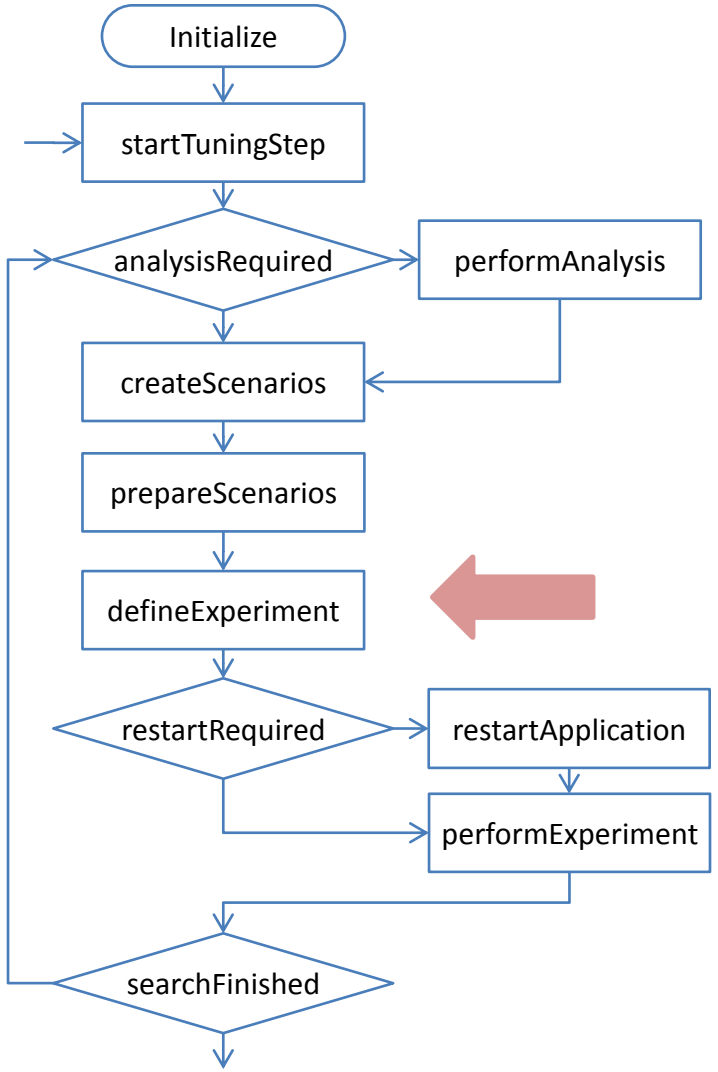

```
void TutMultipleRegions::createScenarios(void) {
  list<MetaProperty>::iterator property;
  MetaProperty longest_running_property;
  list<MetaProperty> found_properties =
                      pool_set->arp->getPreAnalysisProperties(0);
  ...

  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);
  ....
}
```

Find the longest parallel region

Define Experiment

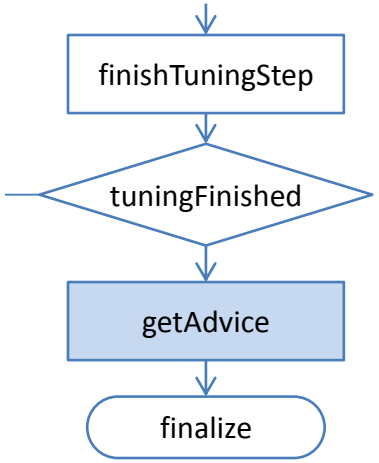
- Select each scenario, set its objective property to `EXECTIME` in the previously selected region at rank 0, and then move it to the experiment scenario pool

```
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



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

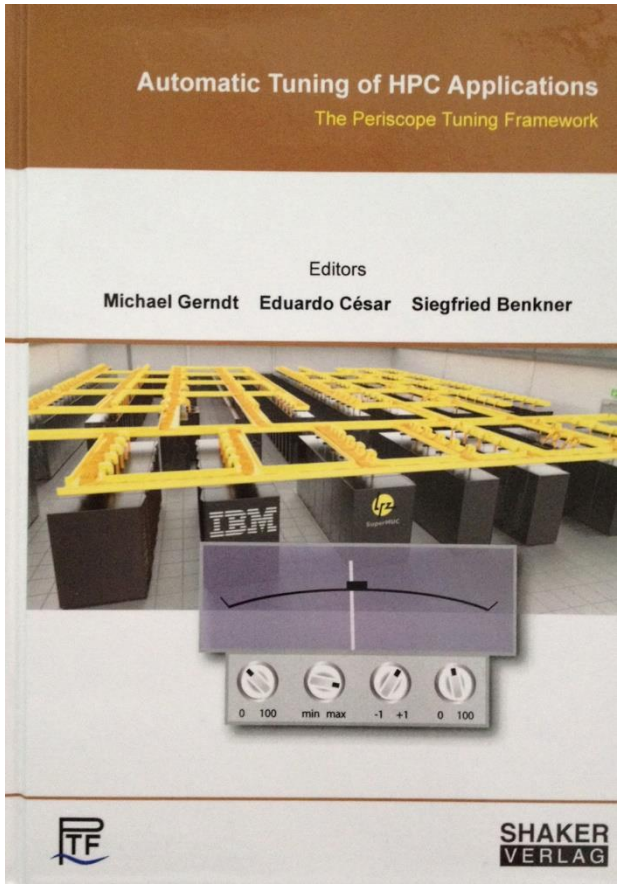
Source code in tar packages:

- 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

Additional Tutorial Plugins

The following tutorials are available :

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

Tuning Plugins

- MPI parameters (UAB, TUM)
  - Eager Limit, Buffer space, collective algorithms
  - Application restart or MPIT Tools Interface
- DVFS (LRZ)
  - Frequency tuning for energy delay product
  - Model-based prediction of frequency, pre-analysis
  - Region level tuning
- Parallelism capping (TUM)
  - Thread number tuning for energy delay product
  - Exhaustive and curve fitting based prediction

Tuning Plugins

- Master/worker (UAB)
  - Partition factor and number of workers
  - Prediction through performance model based on data measured in pre-analysis
- Parallel Pattern (UNI Vienna)
  - Tuning replication and buffer between pipeline stages
  - Based on component distribution via StarPU

Distributed Tuning Plugins

- MPI IO (TUM)
  - Tuning data sieving and number of aggregators
  - Exhaustive and model based
- Compiler Flag Selection (TUM)
  - Automatic recompilation and execution
  - Selective recompilation based on pre-analysis
  - Exhaustive and individual search
  - Scenario analysis for significant routines
  - Combination with Pathway
  - Machine learning support

AutoTune Consortium

AUTOTUNE
www.autotune-project.eu

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