1. Introduction

IBM High Performance Computing Systems Toolkit (HPCST) is a framework that provides services for performance data collection, bottleneck identification, solution discovery and implementation, and iteration of tuning process. It provides access to a wide array of information from static analysis, runtime behavior, algorithm property, architecture feature, and expert domain knowledge. Based upon such information, the framework provides a mechanism to compare and correlate performance metrics from different aspects (e.g., computation, memory, communication, I/O), and pinpoint the cause of performance problems. The framework also attempts to mitigate performance problems by suggesting and implementing solutions.

The methodology can be summarized as follows. We collect the cause of performance problems from literature and performance experts, and store them as patterns defined on performance metrics. Our framework inspects and instruments the application, and actively searches for known patterns in the pattern database. Once a pattern is discovered, we claim that the corresponding bottleneck is found, and the framework consults the knowledge database for possible solutions. Solutions are evaluated, and implemented if desired by the user. The framework is also designed to be open and extensible so that more bottleneck and solution definitions can be populated in the framework. The framework provides the necessary facilities to assist users.

2. Quick Start

1. Unpack the package into a directory
2. Inside the directory, setup the environment variables
   (ACTCT_BASE, IHPCST_BASE, PSIGMA_BASE, IHPCST_DB_PREFIX, etc)

   % source bin/set-actc-envs `pwd`

3. (Optional) Create a customized module by adding a new directory in
   $ACTCT_BASE/hpcst/modules/mmods/
   (For modules providing metrics based on runtime performance data, please refer to Section 3.2. BAS/BIS is helpful inserting probes into applications.)

4. (Optional) Modify the modules, metrics and bottlenecks information in
   $ACTCT_BASE/hpcst/data/

   Part of the file name prefix indicates the architecture (e.g., pwr5 is POWER5 and pwr5p is POWER5+).

5. Start the BDE server process on the front end node where job is submitted to run:

   % $ACTCT_BASE/hpcst/bin/hpcs_bde

6. Start the SDE server process on the front end node where job is submitted to run:
% $ACTCT_BASE/hpcst/bin/hpcs_sde (use TCP port 3491 by default)

7. Start the SIE server process on the front end node where job is submitted to run:

% $ACTCT_BASE/hpcst/bin/hpcs_sie (use TCP port 3493 by default)

8. Start the Control GUI process (uses TCP port 3492 by default)
   % hpcs_gui

9. In the Control GUI (as shown in Figure 1), fill out the information (the field that has the label colored in RED is required):

![Figure 1. Start of the Control GUI](image)

**Hostname** of BDE is where the BDE server process is running.

**Executable, including path**
Compiled with no pg: The application executable that is compiled without –pg option.
Compiled with –pg: The application executable that is compiled with –pg option.
**Runtime Information**
Application Arguments: The arguments passed in when running the application.
The script file (if not same as application binary): **The script file has to be set up in a way that the executable name is the only argument passed to the script.** The reason for that is because the framework may execute different versions of executable while doing the analysis. There is no need to give the executable name here.

**Working Directory**: If an application is needed to read some input files whose path is relative to some directory, then this relative directory has to be specified as the application’s working directory.

**Profile Granularity**
Select either function-level or loop-level hotspot profiling. **However, for the loop-level profiling to work, the source files have to be in the same directory as the application executable.**

Click on “HD” button to run the hotspot detector.

10. After HD finishes. The profiling result will be shown in the control GUI (Figure 2).

Each row in the data window represents a hotspot. Each column gives attributes of the hotspots (e.g., function name, start/end line number…etc.). The **self secs** column shows the CPU time for each hotspot. If the application is a MPI program, the additional **mpi_comm_time** column would be present to show the accumulation of the MPI communication time for each hotspot. By clicking each row, associated source code information should be highlighted in the source window. By clicking on each column label, the rows will be sorted based on the value in that column.

Based on the profiling result, the user may select one or more hotspots (by mark the SELECTION column) for further bottleneck investigation.
Once the selection is done, click on the **BDE** button for bottleneck detection. If users want to consult the SDE at the same time, he or she may click on the **BDE and SDE** button.
11. After BDE finishes. The bottleneck detection result will be shown in the control GUI (Figure 3).

Each row in the data window represents a bottleneck detected. Each column gives an attribute of the bottlenecks (e.g., bottleneck location, dimension(s), rules, description…etc.) By clicking each row, associated source code information should be highlighted in the source window (not shown in Figure 3). Then the user can consult SDE for solutions with selected bottleneck by clicking the SDE button.
12. After SDE finishes. The potential solution results will be shown in the control GUI (Figure 4). Each row in the data window represents a solution. Each column gives an attribute of the solutions (e.g., solution name, code impact, auto implement,…etc.) For the solutions that can be implemented by SIE, the solution has a YES value in the auto implement column. Users can then ask SIE to automatically implement the solution.
13. After SIE finishes. The changes done by SIE will be reported back to GUI as shown in Figure 5.
2.1. Other Useful Features

2.1.1. Port Number Setting

Users can override the default setting by providing a ports.xml file under $HOME/.hpcst directory. The format for the ports.xml file is as following:

```xml
<hpcst>
  <port>
    <BDE>3410</BDE>
    <SDE>3410</SDE>
    <SIE>3410</SIE>
    <GUI>6410</GUI>
  </port>
</hpcst>
```

The GUI, BDE, SDE and SIE programs have to be run after the ports.xml file is created, so they could get each other’s port number.
2.1.2. Environment Variables for Applications

Some of the BDE modules would need to run the application to gather the performance data (refer to section 3.6). To set up the environment variables for the future runs by BDE modules, users can prepare a hpcs.envs file under the same directory as the application executable. Each line in hpcs.envs is a key=value pair. The value is a simple string and no punctuation (e.g. quotation mark) is needed. Here is an example to set up MP_PROCS and MP_HOSTFILE in hpcs.envs.

```
MP_PROCS=8
MP_HOSTFILE=/home/.../hostfile
```

On aix, if the application has additional linking option like -bmaxdata or other shared library, it needs to be specified through the IHPCST_LIBS environment variable.

```
e.g.
IHPCST_LIBS=-lessl -bmaxdata:0x80000000
```

Another environment variable that may need to be set up for our parser in SDE and SIE is the IHPCST_PARSER_OPTIONS environment variable. It passes necessary information about the source to our parser. There are 3 kinds of frequently used options.

- source format (for fortran). For fixed format, use "-ffixed-form", for free format, use "-ffree-form".
- with or without C pre-processing, use "-nocpp" or "-cpp" accordingly.
- additional header file location information, use "-I<directory>"

As an example, the following setting tells the parser that the fortran code is in free format, and it should be parsed without C preprocessing. Also it passes in a directory (/usr/lpp/ppe.poe/include/thread) to check for the header files.

```
IHPCST_PARSER_OPTIONS=-ffree-form -nocpp -I/usr/lpp/ppe.poe/include/thread
```

2.1.3. Configuration

Users can save the current input fields in GUI into a configuration file. Selecting Configuration->Save Configuration As would save the current setting into a file. Users can load a configuration file by selecting Configuration->Load Configuration. Another way to load a configuration file is through the command line option when running the GUI. The GUI will fill out the fields according to the specified configuration file.

```
e.g. hpcs_gui -c configuration_file
     OR
     hpcs_gui --config configuration_file
```

2.1.4. Profiling Results for Other Ranks

If an application is a MPI program, the HD profiling result can report the cpu usage and the communication time for other ranks also (by default, it is to show the rank 0). Right-
click on the panel under the HD tab in the data window brings up the context menu shown in Figure 4. By selecting the **Display Profiling Results for Other Ranks** option, a dialog will pop up and the user can either select the rank id or select to show the min/max/sum value among all the processes. Figure 5 shows the results when the min/max/sum is being selected.

---

**Figure 4. Display results for other ranks**
Since the number of processes may be large, we provide a way for users to limit the number of processes being monitored by HD. Inside the $ACTCT_BASE/bin/set-actct-envs file, the IHPCST_HDPROF_TASKS environment variable is designed for this purpose. The HD profiling results will monitor the processes whose rank id falls between 0 and IHPCST_HDPROF_TASKS – 1.

### 2.1.5. Clockticks

Users can observe the clocktick information after the HD profiling is completed. Right-click on the source window brings up the context menu. Select the **Show Ticks** option. The user can either select the rank id or the min/max/sum value. Figure 6 shows an example with the min/max/sum clocktick. The lines will be highlighted when such a tick information exists.
3. Framework Overview

The Bottleneck Discovery Framework consists of a set of components belonging to two different areas of operation, as illustrated in Figure 7. These areas are User Interface and Bottleneck Detection. Two components communicate if there is an arrow connecting them, in which case the information flows in the direction of the arrow.

The Control GUI is a graphical user interface which coordinates the activity of the framework. Initially, the user will provide the name of the binary executable for the application. After receiving the name of the application executable, the Control GUI will issue a request to the Bottleneck Detection Engine (BDE) and provide it with the name of the executable. BDE will then request the Hotspot Detector (HD) to profile the application and provide a summary of the hotspots and a list of all source files used to produce the binary. The BDE will return this profile information to the Control GUI. The user examines the profile data and optionally chooses to narrow the analysis to selected hotspots (as opposed to the default behavior of analyzing the entire application). The user can also visualize the source code and select contiguous regions of source statements for analysis. Finally, the user can request the BDE to display the identified bottlenecks for selected hotspots. The BDE will then be invoked again and asked to perform the analysis on the selected portions of the code. The BDE will consult the BDE database and then issues a request to the Metrics Scheduler (MSCHED). The MSCHED is responsible for executing the modules (MMODs and PEMODs) and returning the required information to the BDE. The modules are responsible for collecting the performance metrics. The BDE consults the BDE database again based on the received information from MSCHED and composes a bottleneck description for all identified bottlenecks. This bottleneck
The user can then optionally pick one solution, among the ones that are flagged as automatically implementable. When one solution is selected, all solutions that conflict with it will be grayed-out and cannot be selected. When the user decides to apply the selected solutions, the specification of the selected solution and parameters is transmitted to the Solution Implementation Engine (SIE) which implements the solutions and returns a new version of the application source code or binary executable.
**3.1 Bottleneck Detection Engine Database (BDE DB)**

Bottleneck Detection Engine Database is the data repository of the BDE. It stores the information for bottlenecks (rules), performance metrics and modules for the framework. The framework user can add or modify bottlenecks in the database to customize the bottleneck detection.

A bottleneck is defined as a condition, called rule, on a set of metrics. Each rule can be either true or false. For example, a rule can be \#L1Misses > 100, which involves the metric \#L1Misses (number of L1 cache misses). Metrics can also depend on parameters or on other metrics, as in \#L1Misses(Power4) > 100, where Power4 is a parameter and it means the number of L1 cache misses on a Power4 system.

To organize the bottlenecks, each bottleneck is classified into one (or more) “performance dimensions”. Currently, five dimensions are used: CPU, Memory, I/O, Communication, Threads. This classification allows efficient bottleneck organization for further analysis and solution determination.

Each metric must have a corresponding module (MMOD) which is responsible for computing or estimating the metric. Furthermore, bottlenecks can have an optional associated performance improvement metric. This metric represents the expected improvement in performance if the bottleneck is removed (i.e., the application or the system are changed in such a way that the rule associated to the bottleneck evaluates to false). A performance improvement metric must also have an associated module (PEMOD) which is responsible for computing it. The association between modules (MMODs and PEMODs) and metrics is recorded in the BDE DB.

**3.2 Bottleneck Detection Engine (BDE)**

The Bottleneck Detection Engine (BDE) receives the user request from the Control GUI. When the request is for profile data, BDE simply issues a request to the HD and pass the profile data generated by the HD to the Control GUI. BDE’s primary responsibility is to coordinate with the MSCHED to serve the bottleneck detection request from the user. To respond to this request, the BDE first consults the BDE DB and extract the list of all bottlenecks which belong to one of the dimensions selected by the user. It then parses the rules associated with the selected bottlenecks and extracts a list of all metrics, and corresponding parameters, which are needed to evaluate the rules. The BDE then issues a request to the Metrics Scheduler (MSCHED) passing it a list of all metrics needed. After receiving all the metrics collected by MSCHED, the BDE evaluates the rules and composes a bottleneck description for all bottlenecks whose rules evaluate to be true. This description includes the bottleneck name, the area of code where it was evaluated and the percentage improvement if the bottleneck is removed. This information is returned to the Control GUI.
3.3 **Metrics Scheduler (MSCHED)**

The Metrics Scheduler (MSCHED) is responsible for executing the modules associated with the metrics requested by BDE and for collecting and returning these metrics to the BDE. The MSCHED will analyze the dependencies between the metrics (one metric could depend on other metrics, and therefore one module may have to be run before the others) and exploits as much parallelism as possible. One factor needs to be considered is mode: some modules can be run in parallel with other modules and some have to be run alone. The MSCHED will then analyze the dependencies between the metrics, the mode of each module and schedule them appropriately. It will try to exploit as much parallelism as possible while preserving the metric dependencies and mode specifications. Each module returns its metrics to MSCHED and, when all metrics have been collected, these are returned to the BDE.

3.4 **Hotspot Detector (HD)**

The Hotspot Detector (HD) accepts a request from BDE. The HD profiles the application and provides a hotspot summary (e.g., distribution of time spent among functions) together with a list of all source files used to produce the binary.

3.5 **Control GUI**

The Control GUI is a graphical user interface which coordinates the activities of the toolkit. Its role is to coordinate the operations of the framework and display the results to the user. The user may interact with framework such as configuring the application execution or selecting hotspots for bottleneck detection.

3.6 **Module**

A module in the framework is responsible for one or more metrics. To distinguish different functionality in the design, we use Metric Module (MMOD) for a module that provides one or more performance metric and Performance Estimation Module (PEMOD) for a module that provides one or more performance improvement metric.

Depending on the implementation, a module can use static analysis to estimate the metrics or use runtime analysis tools to compute/collect the metrics. The details are given in Section 3.6.1 and Section 3.6.2 respectively.

One important novelty of the framework is the extensibility. In the design, new metrics with associated modules can be added easily. This allows new bottleneck definitions (e.g., new performance problems or existing performance problems on new architecture/platforms). Section 4 describes more details.
3.6.1 Static Analysis
Static analysis is a quick way of finding programming patterns that may cause performance problems and hence constitute performance bottlenecks. One advantage of bottleneck discovery using static analysis is that it usually does not require multiple runs of a potentially long-running application, contrary to runtime analysis. If combined with runtime information, static analysis tools can provide better diagnosis of the problem than a compiler’s capability.

3.6.2 Runtime Performance Data
Another method to implement the module in the framework is using performance tools to collect runtime performance data. This kind of modules instrument the application and collect the performance data during runtime.

As examples, we build these kind of modules using IBM High Performance Computing Toolkit as the performance data collector. The toolkit includes profiling/tracing components (e.g., hardware counter library, MPI communication, program counter sampling, I/O activity and multi-threading) and binary instrumentation. The modules utilize the binary instrumentation to insert the profiling/tracing components into the application based on the performance metrics required to detect the performance bottlenecks.

The binary instrumentation contains two modules, Binary Analysis System (BAS) and Binary Instrumentation System (BIS). Both BAS and BIS are machine and compiler specific.

BAS provides the program structure of the executable to the visualization system. It analyzes an application binary executable and provides information about the statements and variables. BAS interprets the formats of the binary, instructions, addresses and the source program information contained in the binary in formats defined by the machine and compiler. BIS performs the binary instrumentation by modifying the application executable using the information provided by BAS. It provides a low-level binary modification facility, whose task is to modify the execution of an application in order to execute certain actions specified by the module.

Given the application and profiling/tracing toolkit components, BAS and BIS will be able to instrument the application and collect the performance data for the hotspots. Then the module will report the requested performance metrics back to the BDE after the execution.

Please refer to $ACTCT_BASE/psigma/examples/slib for an example about how BAS/BIS is used to insert probes into an application. This functionality is helpful when building a module that provides metrics based on runtime performance data.
3.7 Solution Determination Engine (SDE), Solution Legality Estimator (SLE), and Solution Implementation Engine (SIE)

SDE accepts input bottlenecks from BDE. It will consult the SDE database for solutions and also pass it to SLE/SIE to check and implement. SDE will forward the solutions to GUI. SIE accepts request from GUI to implement a set of solutions with the specified parameters. SIE will implement the solution and return modified source code to GUI.

4 Extending the HPCS Toolkit

The HPCS Toolkit is an extensible framework in the sense that the user can add bottleneck definitions. Adding bottleneck definitions to the toolkit expands its ability to detect possible performance problems.

Adding bottleneck definitions entails adding records to the BDE database, where each record defines the rule that should trigger the bottleneck. These rules are constructed by combining metrics using arithmetic and logical operators. Metrics can also be added to the toolkit by providing a corresponding module that is responsible for collecting the metric.

So, in general, extending the toolkit involves:

1. Adding metrics and bottleneck rules to BDE database. In particular, three tables in the BDE database have to be expanded.
2. Adding the modules responsible for collecting the added metrics and for estimating the impact of the added solutions

4.1 Extending BDE database

First is the metric table. The metric table contains the definition of performance metrics, and the way they should be collected. To put in a new metric, the user needs to give a new metric name, which can be used in bottleneck rules and the name of the module that collects this metric. The second is the module table, which specifies the location of the program to invoke for collecting the metrics associated with the module. The third is the bottleneck table.

The BDE database table files are located in

$ACTCT_BASE/hpcst/data/

The prefix of the file name indicates the architecture (e.g., pwr5 is POWER5 and pwr5p is POWER5+). Lines start with “#” are comments. The format description of each file is given in the top of the file.

For metrics, the format for each line in the file is

`metricName; modName; mparamList`
where `metricsName` is an identifier name of the metric. `modName` is an identifier for
module that is responsible for providing the metric and `mparamList` is a list of mandatory
parameters that must be passed to the module in order for it to provide the metric (i.e., the
same module executable may supports multiple metrics). If no mandatory parameter will
be passed to the module, `mparamList` should be empty (e.g., the line will be in the format of `metricName; modName;`).

Note: metrics can also take optional input arguments, which can be numerical values or
other metrics (or values and/or metrics combined with arithmetic operators). This is
specified in the rule definition.

New rules are added to the BDE database by means of additional entries in the bottleneck
file of the BDE database. These rules consist of metrics combined using arithmetic and
logical operators. Each rule can optionally have an estimated performance improvement
if the bottleneck is removed. This performance improvement is a metric like any other
and therefore it must be registered before it can be used in a bottleneck definition.

For bottlenecks, the format for each line in the file is

```
bottleneckName; cpu memory io comm threads; description; rule [-> pimetric]
```

where `bottleneckName` is the name of the bottleneck. Each column of `cpu memory io
comm threads` should be either 1 or 0 indicating whether the bottleneck belongs to each
specific dimension or not. `description` is a string describing the bottleneck. `rule` is the
bottleneck definition conforming the grammar:

```
rule := ruleTerm | rule or ruleTerm
ruleTerm := ruleFactor | ruleTerm and ruleFactor
ruleFactor := rulePrimary | not rulePrimary
rulePrimary := expr relop expr | ( rule )
metric := metricName | metricName ( exprList )
exprList := expr | exprList , expr
expr := term | expr addop term
term := primary | term mulop primary
primary := constant | metric | ( expr )
addop := + | -
mulop := * | /
relop := < | > | <= | >= | == | !=
```

```metricName := STRING
constant := REAL```
The *pimetric* is optional and is a metric in the above grammar.

For modules, the format for each line in the file is

```
modName; path; exmode
```

where *modName* is the name of the module, *path* is the executable (including full path) and *exmode* is the execution mode (whether the executable can be run in parallel with other executables. *exmode* should be either *shared* or *exclusive.*)

### 4.2 Adding a new module

Rather than modifying the tables, most of the user’s effort should focus on providing the program (module) that collects the performance information. For example, using existing performance tools and with the help of BAS and BIS (described in Section 3.6.2), the user may implement a module that instruments the application and parse performance data. The module is in the form of an executable, and communicates with the framework through a predefined standard interface. The application binary, source code, and hotspot information are provided to the module and, after collecting the information, the module reports the metrics requested.

The following guidelines apply when implementing and registering modules.

1. **Module Format:** Each module (MMOD or PEMOD) must be compiled as an executable file and placed somewhere on the system where BDE server process is Area B (for MMOD and PEMOD modules).

2. **Module Registration:** Each module must be registered with the HPCS Toolkit. Modules of type MMOD and PEMOD are registered by adding an entry of type *(modName, path, exmode)* to the modules file of the BDE database (Section 4.1).

3. **Module Parameters:** Metrics associated with modules of type MMOD and PEMOD can optionally take parameters which can be strings or other metrics registered in the database. The parameters supplied to the metrics will be passed to the corresponding module. Modules should be written in such a way that parameters are optional (i.e. a meaningful default is provided in the case that no parameters are supplied).

4. **Module Input Mechanism:** Module executables are provided with an input string (passed to the module as command-line argument) which represents the full path to the file that contains the input arguments for the module. The format of the input file is specified by Interface 9 for MMOD and PEMOD modules.
5. **Module Output Mechanism:** The module must write its return values in a file called `hpcs.modName.pid.output` and saved in the directory `tmpdirB` (for MMOD and PEMOD modules). The names of these directories are passed to the module in the input file. The format of the output file is specified by Interface 10.

Please refer to `$ACTCT_BASE/hpcst/modules/mmods/sample` for a module example. The package also includes a sample application that is at `$ACTCT_BASE/example/swim_omp`

### 4.3 Interface

**Interface 1**

```
Interface 1 = hdreq hdrequest
  hdrequest = appbinB, [appbinBpg], tmpdirB, execmdB, [profilegran], [appargList]
  appbinB = STRING (full path of the application binary in systemB)
  appbinBpg = STRING (full path of the application binary compiled with -pg for profiling, in systemB. This field is optional)
  tmpdirB = STRING (directory for temporary files on SystemB)
  execmdB = STRING (command to execute the application on SystemB)
  profilegran = STRING (granularity of profiling. Keywords provided by HD block)
  appargList = {apparg}*
  apparg = STRING (argument to be passed to the application)
```

**Implementation of the Interface:** Interface is stored in a data structure which is communicated through socket.

**Interface 3**

```
Interface 3 = num_procs, hdprof_tasks, num_funs, profileList, srcList, callgraph, hddirList
  num_procs = INT (number of processes)
  hdprof_tasks = INT (number of profiling tasks)
  num_funs = INT (number of functions)
  profileList = {profile}*
  profile = hotspot, hotspotmetric
  hotspot = (is_function, fileName, funcName, bln, eln)
  is_function = INT (indicate if the hotspot is a function or an arbitrary region)
  fileName = STRING (name of source file)
  funcName = STRING (name of source function)
```
bln = INT (begin line number within file)
eln = INT (end line number within file)
hotspotmetric = nameList, valueList
nameList = {name} (list of metric names provided for hotspot)
name = STRING (name of the metric)
valueList = {valueListInM}* (pointer to the list of metric values provided for hotspot. Length of nameList and valueList is the same)
valueListInM = {value}+ (list of metric values provided for hotspot for all ranks. Length of valueListInM is num_procs)
value = REAL
srcList = {filename}* (name of source file. Not full path, just name of file.)
callgraph = {funcallgraph}*
funcallgraph = funcName, parentList, childList
funcName = STRING (name of the function)
parentList = {parentName}* (name of the parent)
childList = {childName}* (name of the child)
hddirList = pathToGmonFiles, ihpcst_base
pathToGmonFiles = STRING (directory containing gmon.*.out files)
ihpcst_base = STRING (path to IHPCST_BASE)

Implementation of the Interface: the interface consists of three ASCII files:
- profileList: Name of file is hpcs.appbin.pid.hdprofile
  File format is as follows:
  
  profileList_size. num_funs, hdprof_tasks, num_procs
  {is_function, fileName, funcName, bln, eln, hotspotmetric_size EOL
   name, value_p0, ..., value_pk EOL
   ...
   name, value_P0, ..., value_pk EOL} * EOF

- srclistList: Name of file is hpcs.appbin.pid.hdsrclist
  File format is as follows:

  {filename EOL}* EOF

- callgraph: Name of file is hpcs.appbin.pid.hdcallgraph
  File format is as follows:

  {funcName EOL parentName, ..., parentName EOL childName, ..., childName EOL}* EOF
- hddirList: Name of file is hpcs.appbin.pid.hddir
  File format is as follows:

  {pathToGmonfiles EOL ihpcst_base EOL} EOF

Files will be saved in tmpdirB directory.

**Interface 5**

<table>
<thead>
<tr>
<th>Interface 5</th>
<th>= num_procs, bdereq bderequest, appspec, tmpdirs, hostnameList, callgraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_procs</td>
<td>= INT (number of process)</td>
</tr>
<tr>
<td>bderequest</td>
<td>= (hotspotList, dimList, sdeactivate, [appargList])</td>
</tr>
<tr>
<td>tmpdirs</td>
<td>= tmpdirB, tmpdirC</td>
</tr>
<tr>
<td>hotspotList</td>
<td>= {hotspot}</td>
</tr>
<tr>
<td>hotspot</td>
<td>= see Interface 3</td>
</tr>
<tr>
<td>dimList</td>
<td>= cpu, memory, io, communication, threads</td>
</tr>
<tr>
<td>cpu</td>
<td>= INT (flag if dimension activate or not)</td>
</tr>
<tr>
<td>memory</td>
<td>= INT (flag if dimension activate or not)</td>
</tr>
<tr>
<td>io</td>
<td>= INT (flag if dimension activate or not)</td>
</tr>
<tr>
<td>communication</td>
<td>= INT (flag if dimension activate or not)</td>
</tr>
<tr>
<td>threads</td>
<td>= INT (flag if dimension activate or not)</td>
</tr>
<tr>
<td>appbinB</td>
<td>= filepath (path to executable binary file on systemB)</td>
</tr>
<tr>
<td>appbinC</td>
<td>= filepath (path to executable binary file on systemC)</td>
</tr>
<tr>
<td>appbinD</td>
<td>= filepath (path to executable binary file on systemD)</td>
</tr>
<tr>
<td>filepath</td>
<td>= STRING (full path of file)</td>
</tr>
<tr>
<td>srchandleB</td>
<td>= {dirpath}* (path of dirs containing src files on systemB)</td>
</tr>
<tr>
<td>srchandleC</td>
<td>= {dirpath}* (path of dirs containing src files on systemC)</td>
</tr>
<tr>
<td>srchandleD</td>
<td>= {dirpath}* (path of dirs containing src files on systemD)</td>
</tr>
<tr>
<td>dirpath</td>
<td>= STRING (full path of directory containing source files)</td>
</tr>
<tr>
<td>srclist</td>
<td>= see Interface 3</td>
</tr>
<tr>
<td>topleveldirB</td>
<td>= STRING (full path of top-level directory for compilation)</td>
</tr>
<tr>
<td>topleveldirC</td>
<td>= STRING (full path of top-level directory for compilation)</td>
</tr>
<tr>
<td>topleveldirD</td>
<td>= STRING (full path of top-level directory for compilation)</td>
</tr>
<tr>
<td>compilecmdB</td>
<td>= STRING (command to execute in topleveldir to compile)</td>
</tr>
<tr>
<td>compilecmdC</td>
<td>= STRING (command to execute in topleveldir to compile)</td>
</tr>
<tr>
<td>compilecmdD</td>
<td>= STRING (command to execute in topleveldir to compile)</td>
</tr>
</tbody>
</table>
execmdb = STRING (command to execute the application)
execmdc = STRING (command to execute the application)
execmdd = STRING (command to execute the application)
sdeactivate = INT
   (flag whether solution determination active or not)
apparglist = (see Interface 1)
tmpdirb = STRING (directory for temporary files on SystemB)
tmpdirc = STRING (directory for temporary files on SystemC)

hostnameList = hostnameA, hostnameB, hostnameC, hostnameD
hostnameA = STRING (hostname of systemA)
hostnameB = STRING (hostname of systemB)
hostnameC = STRING (hostname of systemC)
hostnameD = STRING (hostname of systemD)
callgraph = see Interface 3

Implementation of the Interface: data structures communicated through socket.

Interface 9

Interface 9 = num_procs, hotspotList, appbinB, pathsrclistBfile,
tmpdirB, topleveldirB, compilecmdB,
execmdB, [mparamList], [oparamList], [appargList]
num_procs = INT (number of processes)
hotspotList = {hotspot, pathToHotspotFileName}* 
hotspot = see Interface 3
appbinB = see Interface 5
pathsrclistBfile = filepath
   (full path to file containing list of source files full path on systemB)
filepath = see Interface 5
pathToHotspotFileName = STRING (full path to the directory containing the file
   name specified in the hotspot)
topleveldirB = see Interface 5
compilecmdB = see Interface 5
execmdB = see Interface 5
mparamList = {mparam}* 
mparam = STRING (mandatory parameter to mmod or pemod)
oparamList = {oparam}* 
oparam = STRING {optional parameter to mmod or pemod}
tmpdirB = see Interface 1
apparglist = see Interface 1

Implementation of the Interface: One ASCII files is created and its
name is passed to the mmod or pemod executable on the command line.
File is saved in tmpdirB directory. Name of the file is $\text{hpcs.modName.tid.id.input}$ where modName is the name of the module. tid is the thread id and id is a unique identifier for the instance of the module (multiple modules can be running at the same time, for the same thread).
Format of file is as follows:

\begin{verbatim}
num_procs
hotspotList_size
{is_function, fileName, funcName, bln, eln, pathToHotspotFileName EOL}*
appbinB EOL
pathsSrcListBfile EOL
tmpdirB EOL
toplevelDirB EOL
compileCmdB EOL
execCmdB EOL
[mparam1, mparam2, ..., mparamN] EOL
[oparam1, oparam2, ..., oparamN] EOL
[apparg1, apparg2, ..., appargN] EOL
EOF
\end{verbatim}

where the format of the pathsSrcListBfile file is

\begin{verbatim}
{pathToSourcefile EOL}* EOF
\end{verbatim}

**Interface 10**

\begin{verbatim}
Interface 10 = hotspot_mvalueList
hotspot_mvalueList = {hotspot, mvalueList}*
hotspot = see Interface 3
mvalueList = {mvalue}*
mvalue = metricName, valueList
metricName = see Interface 6
valueList = {value}+ (a list of values for all ranks)
value = REAL
(value of the metric metricName for parameters mparamList and for code region hotspot given to module.)
\end{verbatim}

**Implementation of the Interface:** One ASCII files is created by mmod or pemod. Its name is fixed to be $\text{hpcs.modName.pid.output}$ where modName is the name of the mmod or pemod module (as registered in the database) and pid is the process id.
File is saved in tmpdirB directory. Format of the file is the following:

```
HOTSPOT filename,funcname,bln,eln
{metricName, value0, value1, ..., valueN EOL}*
```
HOTSPOT filename,funcname,bln,eln
{metricname, value0, value1,…, valueN EOL}* 
…
…
EOF

5 Contact

David Klepacki klepacki@us.ibm.com
I-Hsin Chung ihchung@us.ibm.com
Guojing Cong gcong@us.ibm.com
Sophia Wen hfwen@us.ibm.com