HPC–Bench:

A Tool to Optimize Benchmarking Workflow for High Performance Computing

Gianina Alina Negoita

Department of Computer Science
Iowa State University
Ames, Iowa, USA
Horia Hulubei National Institute
for Physics and Nuclear Engineering
76900 Bucharest-Magurele, Romania
Email: alina@iastate.edu

Glenn R. Luecke

Department of Mathematics
Iowa State University
Ames, Iowa, USA
Email: grl@iastate.edu

Shashi K. Gadia
and
Gurpur M. Prabhu

Department of Computer Science
Iowa State University
Ames, Iowa, USA
Email: gadia@iastate.edu
Email: prabhu@iastate.edu

***Abstract*—HPC–Bench is a general purpose tool to optimize benchmarking workflow for high performance computing (HPC) to aid in the efficient evaluation of performance using multiple applications on an HPC machine with only a "click of a button". HPC–Bench allows multiple applications written in different languages, multiple parallel versions, multiple numbers of processes/threads to be evaluated. Performance results are put into a database, which is then queried for the desired performance data, and then the R statistical software package is used to generate the desired graphs and tables. The use of HPC–Bench is illustrated with complex applications that were run on the National Energy Research Scientific Computing Center's (NERSC) Edison Cray XC30 HPC computer.**

Keywords–HPC; benchmarking tools; workflow optimization.

I. INTRODUCTION

Today's high performance computers (HPC) are complex and constantly evolving making it important to be able to easily evaluate the performance and scalability of parallel applications on both existing and new HPC computers. The evaluation of the performance of applications can be long and tedious. To optimize the workflow needed for this process, we have developed a tool, HPC–Bench, using the Cyclone Database Implementation Workbench (CyDIW) developed at Iowa State University [1], [2]. HPC–Bench integrates the workflow into CyDIW as a plain text file and encapsulates the specified commands for multiple client systems. By clicking the "Run All" button in CyDIW's graphical user interface (GUI) HPC–Bench will automatically write appropriate scripts and submit them to the job scheduler, collect the output data for each application and then generate performance tables and graphs. Using HPC–Bench optimizes the benchmarking workflow and saves time in analyzing performance results by automatically generating performance graphs and tables. Use of HPC–Bench is illustrated with multiple MPI and SHMEM applications [3], which were run on the National Energy Research Scientific Computing Center's (NERSC) Edison Cray XC30 HPC computer for different problem sizes and for different number of MPI processes/SHMEM processing elements (PEs) to measure their performance and scalability.

There are tools similar to HPC–Bench, but each of these tools has been designed to only run specific applications and measure their performance. For example, ClusterNumbers [4] is a public domain tool developed in 2011 that automates the processor benchmarking HPC clusters by automatically analyzing the hardware of the cluster and configuring specialized benchmarks (HPC Challenge [5], IOzone [6], Netperf [7]). ClusterNumbers, the NAS Parallel Benchmarks [8] and the other benchmarking software are designed to only run and give performance numbers for particular benchmarks, whereas HPC–Bench is designed for easy use with any HPC application and to automatically generate performance tables and graphs. PerfExpert [9] is a tool developed to detect performance problems in applications running on HPC machines. Since it is designed to detect performance problems, PerfExpert is different from HPC–Bench.

The objective of this work is to develop an HPC benchmarking tool, HPC–Bench, as described above and then demonstrate its usefulness for a complex example run on NERSC's Edison Cray XC30. This paper is structured as follows: Section II describes the design of the HPC–Bench tool, which is divided in five Parts. Section III describes the complex example mentioned above. Section IV contains our conclusions.

II. TOOL DESIGN

A simple definition of a workflow is the repetition of a series of activities or steps that are necessary to complete a task. The scientific HPC workflow takes in inputs, e.g., input data, source codes, scripts and configuration files, runs the applications on an HPC cluster and produces outputs that might include visualizations such as tables and graphs. Figure 1 shows a typical example for the scientific HPC workflow diagram.

Scientific HPC workflows are a means by which scientists can model and rerun their analysis. HPC–Bench was designed to optimize the evaluation of the performance of multiple applications. HPC–Bench was implemented using the public domain workbench called Cyclone Database Implementation Workbench (CyDIW). CyDIW was used to develop HPC–Bench for the following reasons:

 ISBN: 978-1-61208-613-2

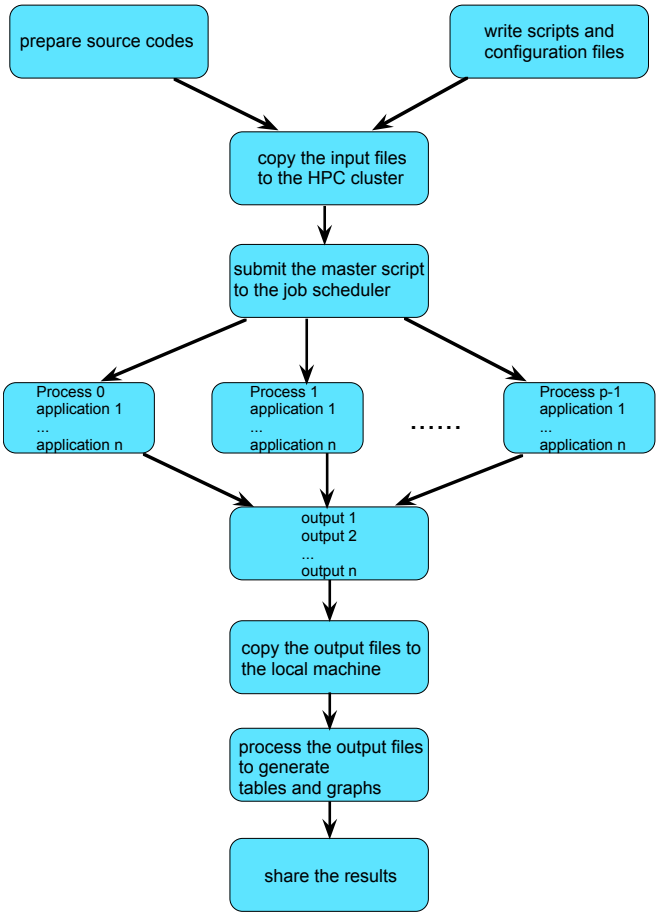

Figure 1. An example for the scientific HPC workflow using n applications that are run on p processes.

- It is easy-to-use, portable (Mac OS, Linux, Windows platforms) and freely available [2].
- It has existing command-based systems registered as clients. The clients used for HPC–Bench are the OS, the open source R environment and the Saxon XQuery engine.
- It has its own scripting language, which includes variables, conditional and loop structures, as well as comments used for documentation, instructions and execution suppression.
- It has a simple and easy-to-use GUI that acts as an editor and a launchpad for execution of batches of CyDIW and client commands.

HPC–Bench uses CyDIW's GUI and database capabilities for managing performance data and contains about 1,000 lines of code. HPC–Bench consists of the following five Parts with illustrations taken from the example described in Section III:

Part 1: XML schema design. An XML schema, known as an XML Schema Definition (XSD), describes the structure of an XML document, i.e., rules for data content. Elements are the main building blocks that contain data, other elements and attributes. Each element definition within the XSD must have a 'name' and a 'type' property. Valid data values for an element in the XML document can be further constrained using the 'default' and the 'fixed' properties. XSD also dictates which subelements an element can contain, the number of instances an element can appear in an XML document, the name, the type and the use of an attribute, etc. The graphical XML schema for this work was created and edited using Altova XMLSpy, see Figure 2. Note the element 'HPC_EXP' contains a sequence of unlimited 'Test' elements, each 'Test' element contains a sequence of 3 'Message' elements, each 'Message' element contains a sequence of 12 'Implementation' elements, each 'Implementation' element contains a choice of unlimited number of 'Process_Rank' elements or 9 'Num_Processes' elements. Each 'Process_Rank' and 'Num_Processes' elements contain a sequence of 'avg', 'max', 'median', 'min' and 'standard_deviation' elements. When using a 'sequence' compositor in XSD, the child elements in the XML document must appear in the order declared in XSD. When using a 'choice' compositor in XSD, only one of the child elements can appear in the XML document. In this work, 'Process_Rank' element will appear in the XML document for the first 'Test' element and 'Num_Processes' otherwise. 'Test' elements stand for applications, 'Message' elements stand for problem sizes, 'Implementation' elements stand for parallel versions, 'Process_Rank' elements stand for process' rank, 'Num_Processes' elements stand for number of MPI processes/SHMEM PEs, while 'avg', 'max', 'median', 'min' and 'standard_deviation' elements stand for statistical timing, respectively.

Part 2: A password-less login to the HPC cluster was implemented. Next, HPC–Bench writes scripts for the submission of the batch jobs. One script is created for each application in a loop and a master script. The master script sets up the environment variables and calls the scripts for each application. This is accomplished by doing the following:

- Use CyDIW's loop structure, *foreach*, to loop through each application.
- Use CyDIW's build-in functions: *createtxt*, *open*, *append*, *appendln*, *appendfile* and *close* to create scripts as text files.
- Use the OS client system registered in CyDIW to copy the files to the HPC cluster.

Part 3: HPC–Bench submits the batch job for execution on the HPC cluster and waits for the job to finish. Suspending the HPC–Bench execution is accomplished by doing the following:

- Launch the job.
- Store its id in a variable.
- Sleep until the 'qstat' command fails, by simply checking the exit status of the 'qstat' command. Once the job is completed, it is no longer displayed by the 'qstat' command.

HPC–Bench next copies the output text files from the HPC cluster to the local machine and converts them to a single written XML file (shown in Figure 3) that follows the XML schema design from Figure 2. An 'awk' script parses the output text files, then a 'shell' script uses the parsed data to create and write the XML file. The XML file is then validated against the XML schema. For example, the 'type' property for an element in XSD must correspond to the correct format of its value in the XML document, otherwise this will cause a validation error when a validating parser attempts to parse the data from the XML document.

 ISBN: 978-1-61208-613-2

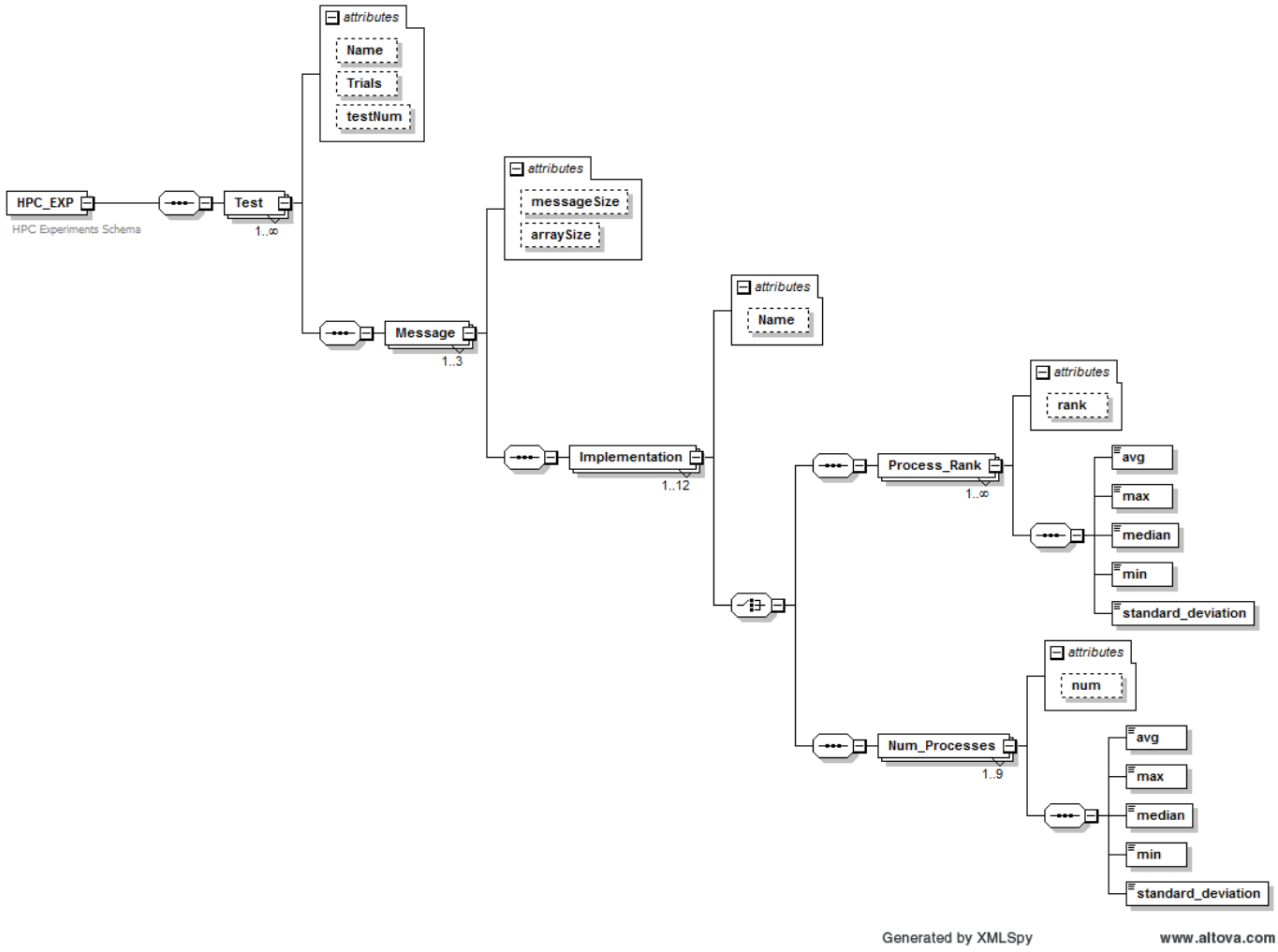

Figure 2. Graphical XML schema using Altova XMLSpy.

```
<HPC_EXP xsi:noNamespaceSchemaLocation="HPCExp.SKG.02.xsd" xmlns:xsi="http://www.w3.org/2001/
        XMLSchema-instance">
    <Test Name="Accessing Distant Messages" Trials="256" testNum="1">
        <Message messageSize="8 bytes" arraySize="1">
            <Implementation Name="shmem_get">
                <Process_Rank rank="1">
                    <avg>7.2357058256976259E-4</avg>
                    <max>9.7059558517284452E-3</max>
                    <median>6.1037067888379840 6E-4</median>
                    <min>4.4106622240733028 6E-4</min>
                    <standard_deviation>8.6332842120298439 5E-4</standard_deviation>
                </Process_Rank>
                <Process_Rank rank="2">
                    <avg>3.3744582335485211 2E-3</avg>
                    <max>1.4090379008746356 2E-2</max>
                    <median>3.1174510620574761 6E-3</median>
                    <min>2.5226988754685547 2E-3</min>
                    <standard_deviation>1.4073810507505 95E-3</standard_deviation>
                </Process_Rank>
                ... data for other ranks, implementations and messages...
            </Implementation>
        </Message>
    </Test>
    <Test Name="Circular Right Shift" Trials="256" testNum="2">
        <Message messageSize="8 bytes" arraySize="1">
            <Implementation Name="shmem_get">
                <Num_Processes num="2">
                    <avg>7.0822053311120358 5E-4</avg>
                    <max>1.1219075385256143 2E-2</max>
                    <median>6.0974593919200332 7E-4</median>
                    <min>4.1982507288629733 9E-4</min>
                    <standard_deviation>9.3970636331058724 E-4</standard_deviation>
                </Num_Processes>
                ... data for other number of processes, implementations,
                messages and Tests ...
    </Test>
</HPC_EXP>
```

Figure 3. The XML file containing the output data validated against the XSD from Figure 2.

Part 4: HPC–Bench then queries the XML file for the desired performance data using the XQuery language to generate

- performance tables

and

- the XML input files to the R statistical package that will be used to generate various graphs.

Queries were declared as string variables in CyDIW and then run. Nested *foreach* command was used to iterate through applications 2 to 5 and through different problem/message sizes. Each output generated by the queries was directed to an XML file, see Figure 4.

```
// Loop through each Test from 2-5;
$CyDB:> foreach $$j in [2, 5]
{
    // Loop through each message size: 8 bytes, 10 Kbytes and 1 Mbyte;
    $CyDB:> foreach $$k in [1, 3] {
        $CyDB:> set $$queryRatioTest$$j[$$k] := ...
        $CyDB:> run $Saxon $$queryRatioTest$$j[$$k] out >> output_tableRatio_Test$$j_$$
                messageSize2[$$k].xml;
    }
}
```

Figure 4. Example setting the queries as variables and running the queries.

For the first application, we queried the average of the median times over all the ranks for each problem/message size and for each parallel version/implementation. See Figure 5 for generating a performance table for application 1. For the other applications we queried the median times for each run (specified by the number of processes used) for each problem/message size and for each parallel version/implementation. See Figure 6 for producing performance tables for applications 2 to 5.

```
$Saxon:>
<Test1TABLE1Ratios xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <table border="1" >
        {
        let $a := doc("ComS363/Final_Project/input.MPI3.xml")//Test[@testNum="1"]
        return
        <tr> <td>Message Size</td>
            <td >{$a/Message[@messageSize="8 bytes"]/Implementation[@Name="shmem_get"]/
                  @Name/string()}</td>
            <td >{$a/Message[@messageSize="8 bytes"]/Implementation[@Name="mpi_get"]/
                  @Name/string()}</td>
            <td >ratio1</td>
            <td >{$a/Message[@messageSize="8 bytes"]/Implementation[@Name="shmem_put"]/
                  @Name/string()}</td>
            <td >{$a/Message[@messageSize="8 bytes"]/Implementation[@Name="mpi_put"]/
                  @Name/string()}</td>
            <td >ratio2</td>
            <td >{$a/Message[@messageSize="8 bytes"]/Implementation[@Name="mpi_send_recv"
                ]/@Name/string()}</td>
            <td >ratio3</td>
        </tr>
        }
        {
        let $a := doc("ComS363/Final_Project/input.MPI3.xml")//Test[@testNum="1"]
        for $x in $a//@messageSize
        let $i := $a/Message[@messageSize=$x]/Implementation[@Name='shmem_get']//median
        let $j := $a/Message[@messageSize=$x]/Implementation[@Name='mpi_get']//median
        let $k := $a/Message[@messageSize=$x]/Implementation[@Name='shmem_put']//median
        let $l := $a/Message[@messageSize=$x]/Implementation[@Name='mpi_put']//median
        let $m := $a/Message[@messageSize=$x]/Implementation[@Name='mpi_send_recv']//
                  median
        return
        <tr>
            <td> {$x/string()} </td>
            <td>{ round(avg($i) * 10000) div 10000.0} </td>
            <td>{ round(avg($j) * 10000) div 10000.0} </td>
            <td >{round(avg($j) div avg($i) * 100) div 100.0}</td>
            <td>{ round(avg($k) * 10000) div 10000.0} </td>
            <td>{ round(avg($l) * 10000) div 10000.0} </td>
            <td >{round(avg($l) div avg($k) * 100) div 100.0}</td>
            <td>{ round(avg($m) * 10000) div 10000.0} </td>
            <td >{round(avg($m) div avg($i) * 100) div 100.0}</td>
        </tr>
        }
    </table>
</Test1TABLE1Ratios>;
```

Figure 5. Query that gives a performance table for application 1.

```
$CyDB:> foreach $$j in [2, 5] // Loop through each Test from 2-5;
{
$CyDB:> set $$queryRatio_8bytes[$$j] :=
<Test$$j_TABLE$$j_Ratios_8bytes $$namespace>
    <table border="1" >
        {
        let $a := $$xmldoc//Test[@testNum="$$j"]/Message[@messageSize="8 bytes"]
        return
        <tr> <td >Message Size</td > <td >8 bytes </td >
            <tr> Number of Processes </tr>
            <td >{$a/Implementation[@Name="shmem_get"]/@Name/string()}</td>
            <td >{$a/Implementation[@Name="mpi_get"]/@Name/string()}</td>
            <td >ratio1</td>
            <td >{$a/Implementation[@Name="shmem_put"]/@Name/string()}</td>
            <td >{$a/Implementation[@Name="mpi_put"]/@Name/string()}</td>
            <td >ratio2</td>
            $$implementationRatioString1[$$j]
        </tr>

        }
        {
        let $a := $$xmldoc//Test[@testNum="$$j"]/Message[@messageSize="8 bytes"]
        for $x in $a/Implementation[@Name='shmem_get']//@num
        let $i := $a/Implementation[@Name='shmem_get']/Num_Processes[@num=$x]/median
        let $j := $a/Implementation[@Name='mpi_get']/Num_Processes[@num=$x]/median
        let $k := $a/Implementation[@Name='shmem_put']/Num_Processes[@num=$x]/median
        let $l := $a/Implementation[@Name='mpi_put']/Num_Processes[@num=$x]/median
        return
        <tr>
            <td> {$x/string()} </td>
            <td>{ round($i * 10000) div 10000.0} </td>
            <td>{ round($j * 10000) div 10000.0} </td>
            <td>{ round($j div $i * 100) div 100.0} </td>
            <td>{ round($k * 10000) div 10000.0} </td>
            <td>{ round($l * 10000) div 10000.0} </td>
            <td>{ round($l div $k * 100) div 100.0} </td>
            $$implementationRatioString2[$$j]
        </tr>
        }
    </table>
</Test$$j_TABLE$$j_Ratios_8bytes>;
$CyDB:> set $$queryRatio_10Kbytes[$$j] :=....
...
$CyDB:> set $$queryRatio_1Mbyte[$$j] :=....
}

// Produce the tables for Tests 2-5 for all message sizes;
// Loop through each Test from 2-5;
$CyDB:> foreach $$j in [2, 5]
{
$CyDB:> run $$prefix $$queryRatio_8bytes[$$j] out >> output_tableRatio_Test$$j_8bytes.xml;
$CyDB:> run $$prefix $$queryRatio_10Kbytes[$$j] out >> output_tableRatio_Test$$j_10Kbytes.xml
        ;
$CyDB:> run $$prefix $$queryRatio_1Mbyte[$$j] out >> output_tableRatio_Test$$j_1Mbyte.xml;
}
```

Figure 6. Query that gives performance tables for applications 2 to 5.

The database was then queried for the data needed to generate the performance graphs. Figure 7 shows the query that gives the median times for all the parallel versions/implementations for 8-byte messages for application 2. The XML file containing the performance data obtained by this query is shown in Figure 8.

```
$CyDB:> set $$query_plot_8bytes[2] :=
<Test$$j_plot$$j_8bytes $$namespace>
    {
    let $a := $$xmldoc//Test[@testNum="$$j"]/Message[@messageSize="8 bytes"]
    for $x in $a/Implementation[@Name='shmem_get']//@num
    return
    <Num_Processes>
        {
        <num_pes> {$x/string()} </num_pes>,
        <shmem_get> { round($a/Implementation[@Name='shmem_get']/Num_Processes[@num=$x]/
                median * 10000) div 10000.0} </shmem_get>,
        <mpi_get>{ round($a/Implementation[@Name='mpi_get']/Num_Processes[@num=$x]/median *
                10000) div 10000.0} </mpi_get>,
        <shmem_put>{ round($a/Implementation[@Name='shmem_put']/Num_Processes[@num=$x]/median
                * 10000) div 10000.0} </shmem_put>,
        <mpi_put>{ round($a/Implementation[@Name='mpi_put']/Num_Processes[@num=$x]/median *
                10000) div 10000.0} </mpi_put>,
        $$implementationString[$$j]
        }
    </Num_Processes>
    }
</Test$$j_plot$$j_8bytes>
;
$CyDB:> run $Saxon $$query_plot_8bytes[2] out >> output_plot_Test2_8bytes.xml;
```

Figure 7. Query that gives the performance data needed to generate the performance graph for 8-byte messages for application 2.

```
<Root>
  <Test2_plot2_8bytes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Num_Processes>
      <num_pes>2</num_pes>
      <shmem_get>0.0005</shmem_get>
      <mpi_get>0.0113</mpi_get>
      <shmem_put>0.0013</shmem_put>
      <mpi_put>0.0096</mpi_put>
      <mpi_sendrecv>0.0026</mpi_sendrecv>
      <mpi_isend_irecv>0.0037</mpi_isend_irecv>
      <mpi_send_recv>0.0054</mpi_send_recv>
   </Num_Processes>
   <Num_Processes>
      <num_pes>4</num_pes>
      <shmem_get>0.0051</shmem_get>
      <mpi_get>0.0169</mpi_get>
      <shmem_put>0.007</shmem_put>
      <mpi_put>0.0155</mpi_put>
      <mpi_sendrecv>0.0093</mpi_sendrecv>
      <mpi_isend_irecv>0.0076</mpi_isend_irecv>
      <mpi_send_recv>0.0084</mpi_send_recv>
   </Num_Processes>
.......
  </Test2_plot2_8bytes>
</Root>
```

Figure 8. The XML file generated by the query above for application 2.

Part 5: HPC–Bench uses R to generate the performance graphs. This is accomplished by first converting the XML files generated by the queries for graphs from Part 4 (see Figure 8 as an example) to R dataframes and then setting up the plotting environment, e.g., the size of the graphs, the style of the X and Y axes, graph labels, colors, legends, etc.

The first step for generating the performance graphs is to install the "XML", "plyr", "ggplot2", "gridExtra" and "reshape2" R packages and load them in R. The "plyr" package is used to convert the XML file to a dataframe. Next, HPC–Bench reads the XML file into an R tree, i.e., R-level XML node objects using the *xmlTreeParse()* function.

 ISBN: 978-1-61208-613-2

Then HPC–Bench uses the *xmlApply()* function for traversing the nodes (applies the same function to each child of an XML node). $function(node)$ $xmlSApply(node, xmlValue)$ does the initial processing of an individual *Num_Processes* node, where *xmlValue()* returns the text content within an XML node. This function must be called on the first child of the root node, e.g., $xmlSApply(doc[[1]], xmlValue)$. All the *Num_Processes* nodes are processed with the command $xmlSApply(doc[[1]], function(x)\ xmlSApply(x, xmlValue))$. The result is a character matrix whose rows are variables and whose columns are records. After transposing this matrix, it is converted to a dataframe. As an example, see Figure 9 that generates the dataframe shown in Table I for application 2. This completes working with XML files and the rest is R programming.

```
# Nodes traversing function
function(node) xmlSApply(node, xmlValue)
doc = xmlRoot(xmlTreeParse(inputFile.xml)
numLoop = xmlSize(doc[[1]])
tmp = xmlSApply(doc[[1]], function(x) xmlSApply(x, xmlValue))
tmp = t(tmp) # transpose matrix
df = as.data.frame(matrix(as.numeric(tmp), numLoop))
names(df)<- c("Number Processes", "shmem_get", "mpi_get", "shmem_put", "mpi_put", "
        mpi_sendrecv", "mpi_isend_irecv", "mpi_send_recv")
```

Figure 9. Code to convert an XML file to an R dataframe.

TABLE I. THE R DATAFRAME GENERATED WITH THE CODE FROM FIGURE 9 FOR 8-BYTE MESSAGE SIZE FOR APPLICATION 2.

	Num Proc	*shmem get*	*mpi get*	*shmem put*	*mpi put*	*mpi send--recv*	*mpi isend irecv*	*mpi send recv*
1	2	0.0005	0.0113	0.0013	0.0096	0.0026	0.0037	0.0054
2	4	0.0051	0.0169	0.0070	0.0155	0.0093	0.0076	0.0084
3	8	0.0046	0.0178	0.0084	0.0171	0.0118	0.0106	0.0125
4	16	0.0056	0.0246	0.0088	0.0250	0.0124	0.0115	0.0137
5	32	0.0048	0.0289	0.0088	0.0269	0.0142	0.0126	0.0113
6	64	0.0053	0.0357	0.0112	0.0329	0.0144	0.0134	0.0160
7	128	0.0054	0.0494	0.0122	0.0378	0.0165	0.0190	0.0215
8	256	0.0057	0.0518	0.0120	0.0502	0.0207	0.0225	0.0232
9	384	0.0093	0.0584	0.0198	0.0540	0.0223	0.0224	0.0247

After obtaining the R dataframes, HPC–Bench sets up the plotting environment as follows:

- Use the "ggplot2","gridExtra" and "reshape2" R packages to create graphs and put multiple graphs on one panel.
- Write a function to create minor ticks and then write another function to mirror both axes with ticks.
- Set and update a personalized theme: *theme_set(theme_bw()), theme_update(. . .)*.
- For each application, plot the dataframe for each problem/message size using the *ggplot()* function with personalized options. See Figure 10.

```
p <- ggplot(data=df.melted, aes(x='Number Processes', y=value, group=variable, shape=factor(
        variable), color=variable))
p <- p + geom_line(aes(linetype=variable)) + geom_point(fill = "white", size = 2.5)
p <- p + geom_line(aes(linetype=variable)) + geom_point(fill = "white", size = 2.5)
p <- p + scale_colour_manual(messageSize[c(i)], values=c("red", "red", "blue", "blue", "
        brown4", "darkgreen", "green"), labels=c("SHMEM get", "MPI get","SHMEM put", "MPI
        put", "MPI sendrecv", "MPI isend&irecv", "MPI send&recv"))
```

Figure 10. Code that generates a plot using the df dataframe.

For each application and for each problem/message size, HPC–Bench plots the desired timing data for all versions/implementations. Next, for each application, HPC–Bench places the three plots for different problem/message sizes (p1, p2 and p3) into one panel using *gtable* to generate a graph, that is then printed to PDF format, see Figure 11. At the end of the HPC–Bench execution, performance graphs are displayed for all applications in popup windows. Figures 14 and 15 illustrate this.

```
ge <- gtable:::rbind_gtable(p1, p2, "first")
g <- gtable:::rbind_gtable(ge, p3, "first")
grid.newpage()
# grid.draw(ge) # draw 2 figures
grid.draw(g) # draw 3 figures, show the plot
# Print to pdf using pdf and plot
pdf(outputFile)
plot(g)
dev.off()
```

Figure 11. Code that places 3 plots into one panel.

Figure 12 shows the HPC workflow diagram for HPC–Bench. The blue boxes are components of the HPC workflow, which include input data and output data to manage, as well as source codes, scripts and configuration files for the system. The red boxes show the portions of the HPC workflow controlled by HPC–Bench.

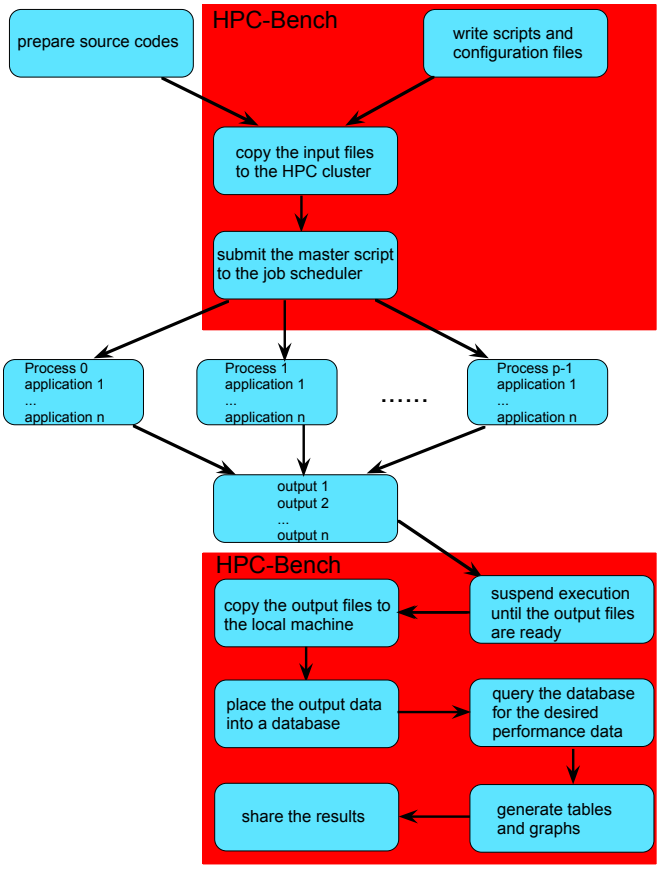

Figure 12. HPC workflow diagram for HPC–Bench.

Since the output processing part cannot begin until all the runs are complete, HPC–Bench suspends execution until all the output data is available. HPC–Bench then puts the output data into a database and queries it for the desired results.

III. EXAMPLE USING HPC–BENCH

In this section, we illustrate how HPC–Bench can be used in a complex benchmarking environment. The example and the benchmarking environment information come from [3]. The

benchmark tests used for this example were: accessing distant messages, circular right shift, gather, broadcast, and all-to-all. Each test has several parallel versions, which use: MPI *get*, *put*, blocking and non-blocking *sends*/*receives*, *gather*, *broadcast* and *alltoall* routines as well as the SHMEM *get*, *put*, *broadcast* and *alltoall* routines.

The NERSC's Edison Cray XC30 with the Aries interconnect was used for benchmarking. Edison has 5576 XC30 nodes with 2 Intel Xeon E5-2695v2 12-chip processor for a total of 24 cores per node. There are 30 cabinets and each cabinet consists of 192 nodes. Cabinets are interconnected using the Dragonfly topology with 2 cabinets in a single group.

For this example, 2 cabinets in a single group (2x192 nodes) were reserved. Each application was run with 2 MPI processes/SHMEM PEs per node using message sizes of 8 bytes, 10 Kbytes and 1 Mbyte and 2 to 384 MPI processes/SHMEM PEs.

Use of HPC–Bench is illustrated via CyDIW's GUI, shown in Figure 13. The GUI is intentionally designed to be as simple as possible for ease-of-use: it has a "Commands Pane", an "Output Pane" and a "Console". The "Commands Pane" acts as an editor and a launch-pad for execution of batches of commands, written as text files. The output can be shown in the "Output Pane", directed to files, or displayed in popup windows. The "Output Pane" is an html viewer, but it can display plain text as well. For example, a user can see an html table computed by an XQuery query displayed in the "Output Pane". The html code or the display in an html browser can be viewed without having to get out of the GUI in order to use a text editor or an html browser. The "Console" displays the status and error messages for the commands.

In CyDIW's GUI, click "Open" and then browse to the HPC–Bench file to open HPC–Bench. One can run all the applications from scratch and produce the performance tables and graphs in a "click of a button" by clicking the "Run All" button. HPC–Bench displays one three-panel graph for each application in a popup window. See Figures 14 and 15 as examples for performance graphs produced by HPC–Bench.

Figure 14 shows the median time in milliseconds (ms) versus the process' rank for the accessing distant messages test with 8-byte, 10-Kbyte and 1-Mbyte messages. The purpose of this test is to determine the performance differences of 'sending' messages between 'close' processes and 'distant' processes using SHMEM and MPI routines. The curves represent various implementations of this test using the SHMEM and MPI *get* and *put* routines, as well as the MPI *send*/*receive* routines as shown in the legend. Figure 14 shows that times to access messages within a group of two cabinets on NERSC's Edison Cray XC30 were nearly constant for each implementation, showing the good design of the machine.

Figure 15 shows the median time in milliseconds (ms) versus the number of processes for the circular right shift test with 8-byte, 10-Kbyte and 1-Mbyte messages. In this test, each process 'sends' a message to the right process and 'receives' a message from the left process. The curves represent various implementations of this test using the SHMEM and MPI *get* and *put* routines, as well as the MPI two-sided routines, e.g., *send*/*receive*, *isend*/*ireceive* and *sendrecv* as shown in the legend. Figure 15 shows that all implementations scaled well with the number of processes for all message sizes.

HPC–Bench can be easily modified by clicking the "Edit" button to run only selected applications or to change the number of processes, library version or configuration to run on, as well as to add more queries to do a different performance analysis. Alternatively, one can run parts of HPC–Bench selecting which parts to run and then clicking the "Run Selected" button. This is useful when one would like to produce additional tables and graphs from existing output data without having to rerun the applications.

IV. Conclusion

HPC–Bench is a general purpose tool to minimize the workflow time needed to evaluate the performance of multiple applications on an HPC machine at the "click of a button". HPC–Bench can be used for performance evaluation for multiple applications using multiple MPI processes, Cray SHMEM PEs, threads and written in Fortran, Coarray Fortran, C/C++, UPC, OpenMP, OpenACC, CUDA, etc. Moreover, HPC–Bench can be run on any client machine where R and the CyDIW workbench have been installed. CyDIW is preconfigured and ready to be used on a Windows, Mac OS or Linux system where Java is supported. The usefulness of HPC–Bench was demonstrated using complex applications on a NERSC's Cray XC30 HPC machine.

Acknowledgment

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Personnel time for this project was supported by Iowa State University.

References

[1] X. Zhao and S. K. Gadia, "A Lightweight Workbench for Database Benchmarking, Experimentation, and Implementation," IEEE Transactions on Knowledge and Data Engineering, vol. 24, no. 11, Nov. 2012, pp. 1937–1949, DOI: 10.1109/TKDE.2011.169, ISSN: 1041-4347.

[2] "Cyclone Database Implementation Workbench (CyDIW)," 2012, URL: http://www.research.cs.iastate.edu/cydiw/ [accessed: 2018-01-10].

[3] G. A. Negoita, G. R. Luecke, M. Kraeva, G. M. Prabhu, and J. P. Vary, "The Performance and Scalability of the SHMEM and Corresponding MPI Routines on a Cray XC30," in Proceedings of the 16^{th} International Symposium on Parallel and Distributed Computing (ISPDC 2017) July 3–6, 2017, Innsbruck, Austria. IEEE, Jul. 2017, pp. 62–69, DOI: 10.1109/ISPDC.2017.19, ISBN: 978-1-5386-0862-3.

[4] "ClusterNumbers," 2011, URL: https://sourceforge.net/projects/clusternumbers/ [accessed: 2018-01-10].

[5] "The HPC Challenge Benchmarks," URL: http://icl.cs.utk.edu/hpcc/ [accessed: 2018-01-10].

[6] "IOzone," URL: http://iozone.org/ [accessed: 2018-01-10].

[7] "Netperf," URL: https://hewlettpackard.github.io/netperf/ [accessed: 2018-01-10].

[8] "The NAS Parallel Benchmarks derived from computational fluid dynamics (CFD) applications," URL: www.nas.nasa.gov/publications/npb.html [accessed: 2018-01-10].

[9] M. Burtscher, B. D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne, "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, November 13–19, 2010, New Orleans, LA, USA. ACM/IEEE, Nov. 2010, pp. 1–11, DOI: 10.1109/SC.2010.41.

 ISBN: 978-1-61208-613-2

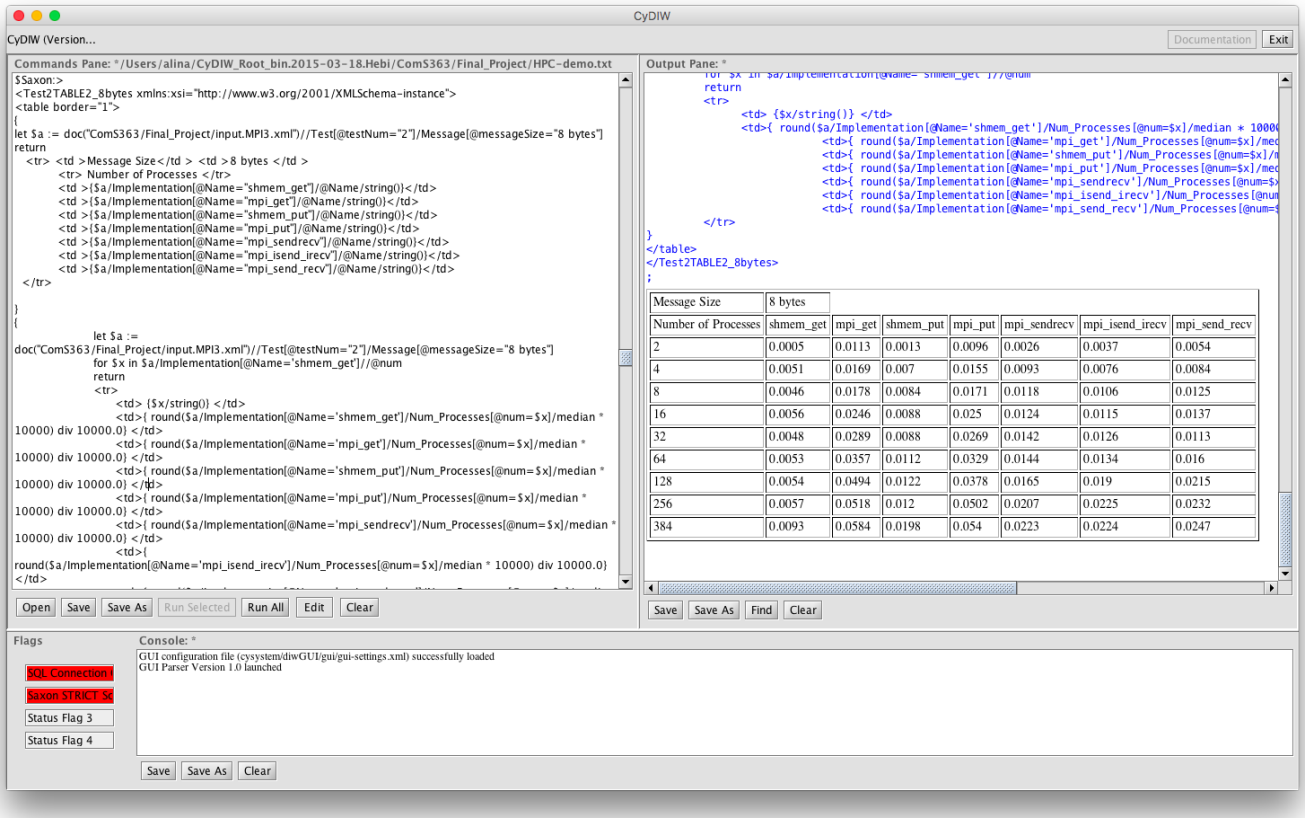

Figure 13. CyDIW's GUI showing the table generated by XQuery for 8-byte message for application 2, containing the same performance data as Table I.

Message Size	8 bytes						
Number of Processes	shmem_get	mpi_get	shmem_put	mpi_put	mpi_sendrecv	mpi_isend_irecv	mpi_send_recv
2	0.0005	0.0113	0.0013	0.0096	0.0026	0.0037	0.0054
4	0.0051	0.0169	0.007	0.0155	0.0093	0.0076	0.0084
8	0.0046	0.0178	0.0084	0.0171	0.0118	0.0106	0.0125
16	0.0056	0.0246	0.0088	0.025	0.0124	0.0115	0.0137
32	0.0048	0.0289	0.0088	0.0269	0.0142	0.0126	0.0113
64	0.0053	0.0357	0.0112	0.0329	0.0144	0.0134	0.016
128	0.0054	0.0494	0.0122	0.0378	0.0165	0.019	0.0215
256	0.0057	0.0518	0.012	0.0502	0.0207	0.0225	0.0232
384	0.0093	0.0584	0.0198	0.054	0.0223	0.0224	0.0247

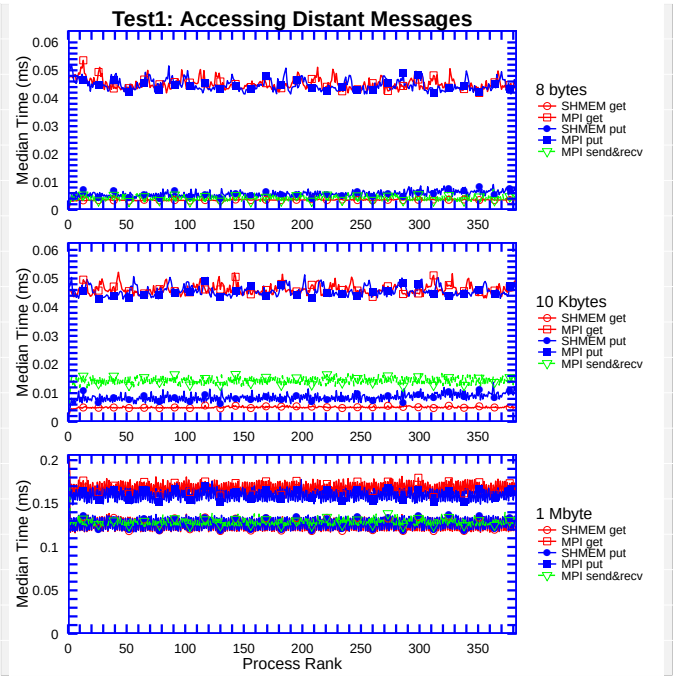

Figure 14. An example of a graph generated by HPC–Bench for application 1, accessing distant messages test.

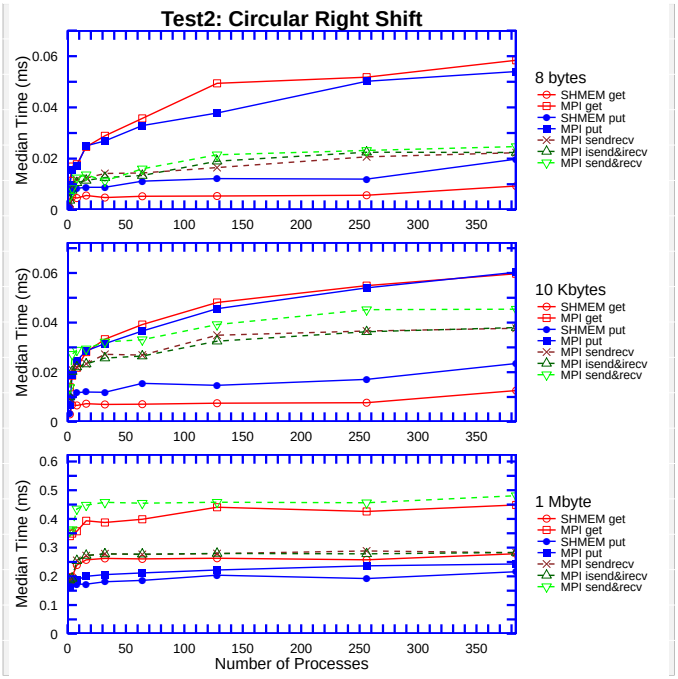

Figure 15. An example of a graph generated by HPC–Bench for application 2, circular right shift test.