Parallel Performance Wizard Tutorial

a hands-on look at the Parallel Performance Wizard tool
for v3.2

Adam Leko

Table of Contents

Parallel Performance Wizard Tutorial 1

1 Compiling the IS benchmark 2

2 The Initial Run 3

3 Tweaking Instrumentation and Measurement 4

4 Analyzing the Profile Data 6

5 Generating Trace Data 11

6 Browsing the Trace with Jumpshot 12

7 Summary 16

Parallel Performance Wizard Tutorial

This document provides a hands-on look at the Parallel Performance Wizard (PPW) performance analysis tool. In this tutorial we will analyze the GWU Unified Parallel C (UPC) implementation of the NPB2.4 integer sort (IS) benchmark, showing how to use PPW to identify potential performance issues in the code.

This tutorial assumes you already have PPW installed and working properly. For instructions on how to install PPW, please see the PPW user manual which is available online at the PPW website.

Additionally, data from this tutorial was gathered on a 32-node AMD Opteron cluster with a Quadrics interconnect. If follow along with this tutorial on other systems, you might see slightly different performance characteristics for this benchmark.

Even though this tutorial focuses on UPC, most of the techniques presented here also apply to other languages (such as SHMEM). See the PPW user manual for instructions on how to use PPW with other parallel languages besides UPC.

Special thanks goes to the HPCL lab at GWU for releasing their NPB implementations to the general public, provided excellent source material for dissection in this tutorial.

1 Compiling the IS benchmark

To analyze the IS benchmark, the first thing we must do is obtain a copy of it. The easiest way to do this is to download the Berkeley UPC source code distribution, and then copy the '`NPB2.4`' directory from the '`upc-tests`' directory, as in

```
$ tar xvzf berkeley_upc-2.8.0.tar.gz
$ cd berkeley_upc-2.8.0
$ cp -av upc-tests/NPB2.4 ~
```

This should result in a new directory named '`NPB2.4`' being placed in your home directory. Note that your system may require slightly different commands as above, especially if you are using a different version of Berkeley UPC.

Once you have a copy of the NPB2.4 benchmarks, the next thing to do is to compile them. Normally, this benchmark suite is compiled using a command such as

```
$ make IS NP=32 CLASS=B
```

To generate performance data files for use with PPW, you must compile the benchmark with the `ppwupcc` command instead of the regular `upcc` command. The `ppwupcc` command is set up as a *very* thin wrapper to the normal `upcc` command, and will pass through any flags it does not understand. The idea is to substitute `ppwupcc` into your makefiles or build scripts, and `ppwupcc` will take care of the rest of bookkeeping required by PPW, such as linking against any libraries it requires and keeping snapshots of source code files for later viewing.

For the IS benchmark, if you grabbed a copy of the NPB2.4 source code from Berkeley UPC, then you simply change your compilation command to

```
$ make UPCC=ppwupcc IS NP=32 CLASS=B
```

If you obtained the NPB2.4 source tree from somewhere else, you might have to hack up the '`make.def`' file in the '`config`' directory.

2 The Initial Run

In general, if you don't know much about the performance of the application you are studying, it is a good idea to perform an initial "profiling run" on a moderate number of nodes using representative input data. This is exactly what we are going to do with our IS benchmark.

Since we don't really have an idea of how time is being spent in the IS benchmark, it would be really great if PPW could provide us with a breakdown of how much time was spent executing different functions in the IS source code. To get this information, we make use of the '`--inst-functions`' flag. Go to the NPB2.4 directory that you set up in the previous section and type

```
$ make UPCC="ppwupcc --inst-functions" IS NP=32 CLASS=B
```

or edit the '`make.def`' file as before. After a short delay, you should have a shiny new '`is.B.32`' executable in the '`IS`' directory.

We can now run the '`is.B.32`' executable directly as we normally would (`srun`/`yod`/`upcrun`/etc). If you do this, you should get a new data file in the directory where you ran the IS benchmark under a name like '`ppw-2234.par`'. This works OK, but it is more useful if we can tell PPW which filename to use. To do this, prefix your normal run command with the `ppwrun` command, such as

```
$ ppwrun --output=is.B.par srun -N 32 is.B.32
```

Typing `ppwrun` with no options will show a short help screen that gives all the different options available for controlling PPW's measurement code at runtime. If you make use of the '`--output`' option, remember to give your data files an extension of '`.par`' to make them easy to find in the GUI later on.

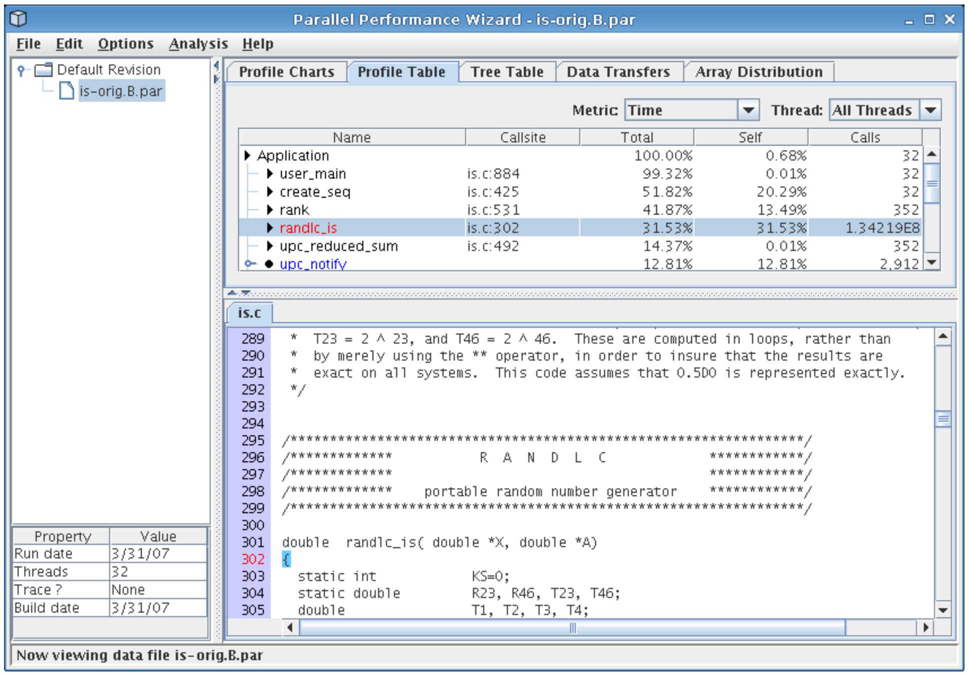
> **Note:** `ppwrun` works its magic by setting a bunch of environment variables, and relies on the job spawner to propagate these new environment variables when launching the job. If it seems like `ppwrun` is ignoring your options, then your job spawner probably doesn't handle propagating environment variables. If so, see the documentation for `ppwrun` in the user manual for some workarounds. Luckily, most job spawners do a good job of environment propagation.

Now that we've figured out how to run applications with PPW, go ahead and run the '`is.B.32`' executable using `ppwrun`. Then, take the new data file created after running the benchmark and transfer it to your local workstation using `scp` or something similar. Start up the PPW frontend GUI, and open up the newly-created data file.

If you don't have an easy way of transferring data files from your parallel machine to your local workstation, then you may use the `ppwprof` and/or `ppwprof.pl` to view text-only performance information from the PAR data file. These commands provide only a subset of the functionality available from the PPW GUI, but are useful for quickly browsing performance data without having to transfer files around. The rest of this tutorial assumes you are using the PPW GUI to browse performance data.

3 Tweaking Instrumentation and Measurement

One of the first things we must do when looking at a data file is to determine if the performance data we obtained is accurate and hasn't been adversely affected by the extra operations that PPW performs while tracking performance data. The most foolproof method of doing this is to compare the overall time taken by a profiled run with the time take for a run compiled without using `ppwupcc`. Another way to do this is to look for any suspicious-looking data using the PPW GUI. For this tutorial, we opt for the second method, using PPW itself to help identify suspect data.



Name	Callsite	Total	Self	Calls
Application		100.00%	0.68%	32
user_main	is.c:884	99.32%	0.01%	32
create_seq	is.c:425	51.82%	20.29%	32
rank	is.c:531	41.87%	13.49%	352
randlc_is	is.c:302	31.53%	31.53%	1.34219E8
upc_reduced_sum	is.c:492	14.37%	0.01%	352
upc_notify		12.81%	12.81%	2,912

```
 *  T23 = 2 ^ 23, and T46 = 2 ^ 46.  These are computed in loops, rather than
 *  by merely using the ** operator, in order to insure that the results are
 *  exact on all systems.  This code assumes that 0.5D0 is represented exactly.
 */


/*****************************************************************/
/*************           R  A  N  D  L  C             ************/
/*************                                        ************/
/*************    portable random number generator    ************/
/*****************************************************************/

double  randlc_is( double *X, double *A)
{
  static int          KS=0;
  static double       R23, R46, T23, T46;
  double              T1, T2, T3, T4;
```

Property	Value
Run date	3/31/07
Threads	32
Trace ?	None
Build date	3/31/07

Now viewing data file is-orig.B.par

Figure 3.1: Profile table showing rand1c_is function

If you were paying attention to the time taken for the run in the previous section, you may have noticed it took a bit longer than a normal run. If we open up that data file and view the profile table visualization (see Figure 3.1), we see something interesting: when we instrumented our program using the '`--inst-functions`' flag, we inadvertently captured a ton of calls to the '`rand1c_is`' function. This function is highlighted in red because PPW thinks that the overhead imposed by tracking each call to this function may have caused it to over-report its influence on overall execution time.

To further check out this function, we un-hide the min and max columns of the table by right-clicking on the table's column headers. When we do this, we see that this function was called about 134 million times, but the total, min, max times show that each individual call took only a few microseconds to execute. Since PPW generally has a fixed amount of work added to each function call, this function was heavily penalized with extra overhead. Ouch!

This 'rand1c_is' function is only used in initialization, so we're not really interested in its performance. We could recompile our code without using the '--inst-functions' flag, but this would mean we'd lose *all* function-level data, which is too restrictive. Instead, we can make use of some '#pragma's offered by the GASP interface to ask the compiler to avoid instrumenting this particular function. Find the 'rand1c_is' function, then add

```
#pragma pupc off
```

before the function body definition (somewhere around line 295) and

```
#pragma pupc on
```

just after the end of the function (somewhere around line 361). The next time we run our IS benchmark, we will avoid getting performance information for this function, even if we compile with '--inst-functions'.

Looking for more things to cut out, from Figure 3.1 we see that the 'create_seq' function takes up a significant percentage of overall execution time, but is unrelated to the core integer sort algorithm. We need a way for PPW to pretend that time spent executing this particular function never happened in the first place. In other words, we want PPW to subtract out the time spent in this function when displaying performance data. The GASP interface comes to the rescue here again, by providing a simple API for disabling measurement during your program's execution. To do this, we add a special include to the top of the IS source code file 'is.c':

```
#include <pupc.h>
```

Then we add calls to 'pupc_control' around the part of the code that calls 'create_seq':

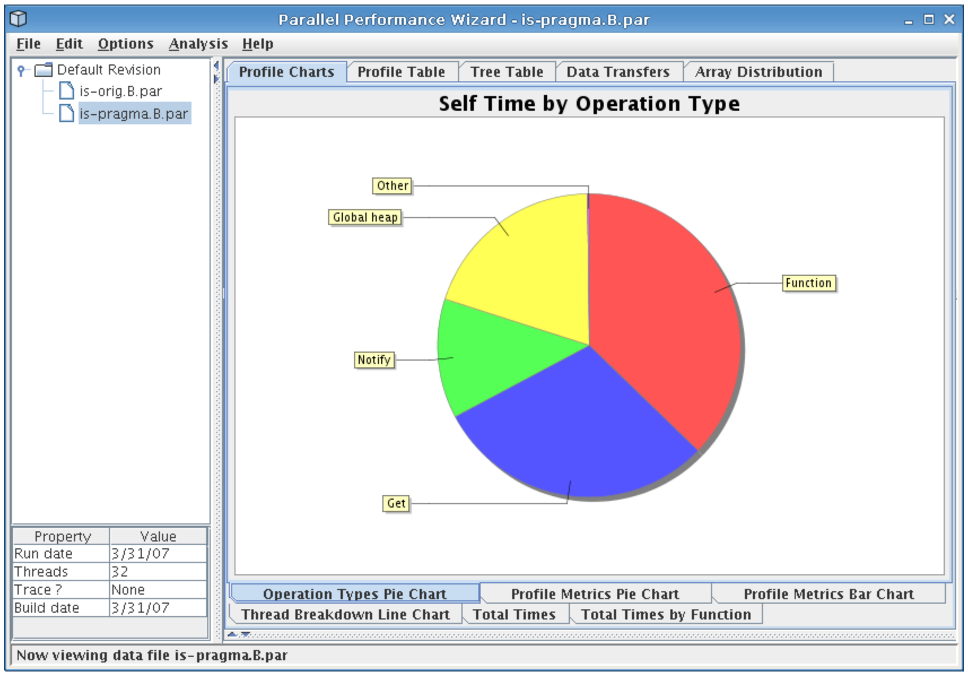
```
/*  Generate random number sequence and subsequent keys on all procs */
pupc_control(0);
create_seq( find_my_seed(
   MYTHREAD,
 THREADS,
   4*TOTAL_KEYS,
   314159265.00,      /* Random number gen seed */
   1220703125.00 ),   /* Random number gen mult */
   1220703125.00 );   /* Random number gen mult */
pupc_control(1);
```

The next time we run our benchmark, PPW will avoid capturing performance information for the 'pupc_control' function, and will subtract out time spent executing that code from the 'main' function's totals. You'll probably want to place similar statements around calls to the calls to the 'full_verify' function, too.

Keep in mind that while the '#pragma' and 'pupc_control' techniques achieve similar results, they perform different functions. The '#pragma pupc's instruct the compiler to avoid instrumenting particular sections of your code, while the 'pupc_control' API calls do not affect instrumentation but turn the measurement code on and off at runtime. If you're looking to reduce overhead, use the '#pragma pupc' directives, but if you're looking to ignore parts of your application, use the 'pupc_control' API calls.

4 Analyzing the Profile Data

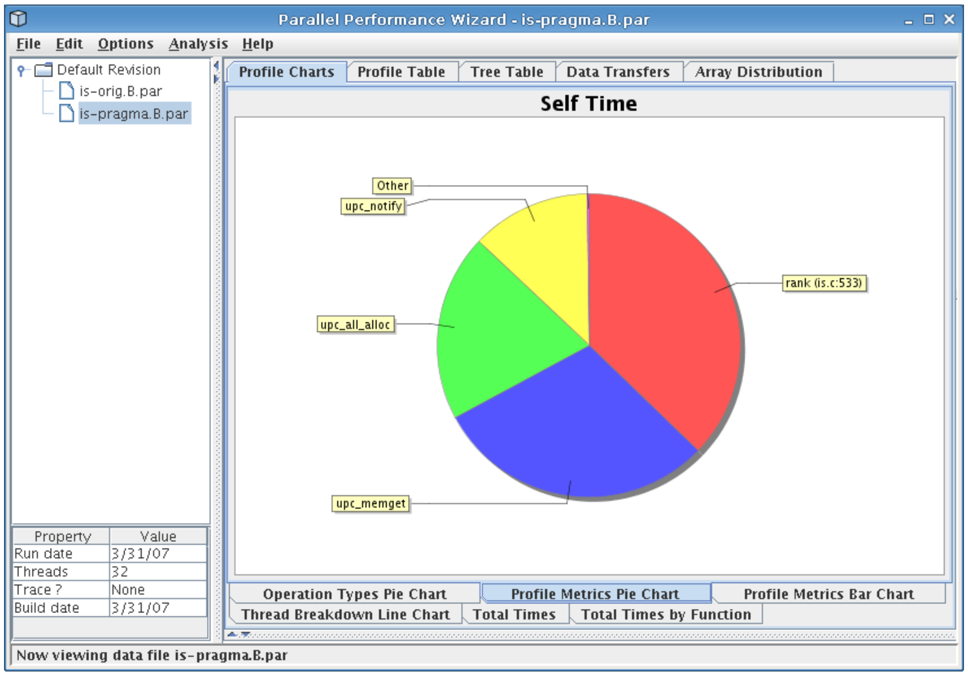
Once you've tweaked the instrumentation and measurement for the IS benchmark, run the benchmark again using the `ppwrun` command as described in Chapter 2 [The Initial Run], page 3. Transfer the resulting data file to your workstation, and open it up with the PPW GUI.


Parallel Performance Wizard - is-pragma.B.par
File Edit Options Analysis Help
Default Revision
is-orig.B.par
is-pragma.B.par
Profile Charts
Profile Table
Tree Table
Data Transfers
Array Distribution
Self Time by Operation Type
Other
Global heap
Function
Notify
Get
Operation Types Pie Chart
Profile Metrics Pie Chart
Profile Metrics Bar Chart
Thread Breakdown Line Chart
Total Times
Total Times by Function
Now viewing data file is-pragma.B.par

Property	Value
Run date	3/31/07
Threads	32
Trace ?	None
Build date	3/31/07

Figure 4.1: Operation types pie chart for the IS benchmark

The first thing you should see when opening the file in the GUI is the operation types pie chart, which is also shown in Figure 4.1. This pie chart shows how much time was spent doing different types of operations at runtime over all nodes in the system. Of particular interest here is the fact that more than 60% of time was spent doing notify operations (part of a barrier), get operations, and global heap management operations.

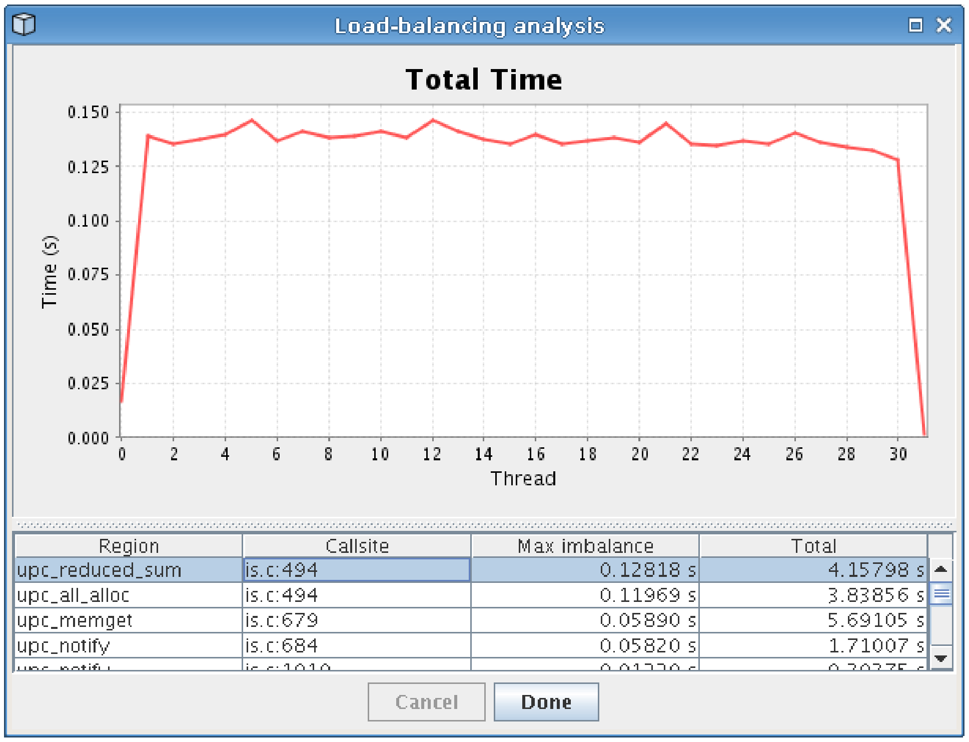


Property	Value
Run date	3/31/07
Threads	32
Trace ?	None
Build date	3/31/07

Now viewing data file is-pragma.B.par

Figure 4.2: Profile metrics pie chart for the IS benchmark

If you switch to the profile metrics pie chart by clicking on the bottom row of tabs, you should see a display similar to that in Figure 4.2. This shows us similar information to the operation types pie chart, except that it displays data in terms of particular language operations rather than generic classes of operations. From the figure we see that the '`rank`' function comprises most computation time for our application, but calls to '`upc_notify`', '`upc_all_alloc`', and '`upc_memget`' account for a large fraction of execution time. We need to keep this in mind when hunting through our data set to identify performance issues.

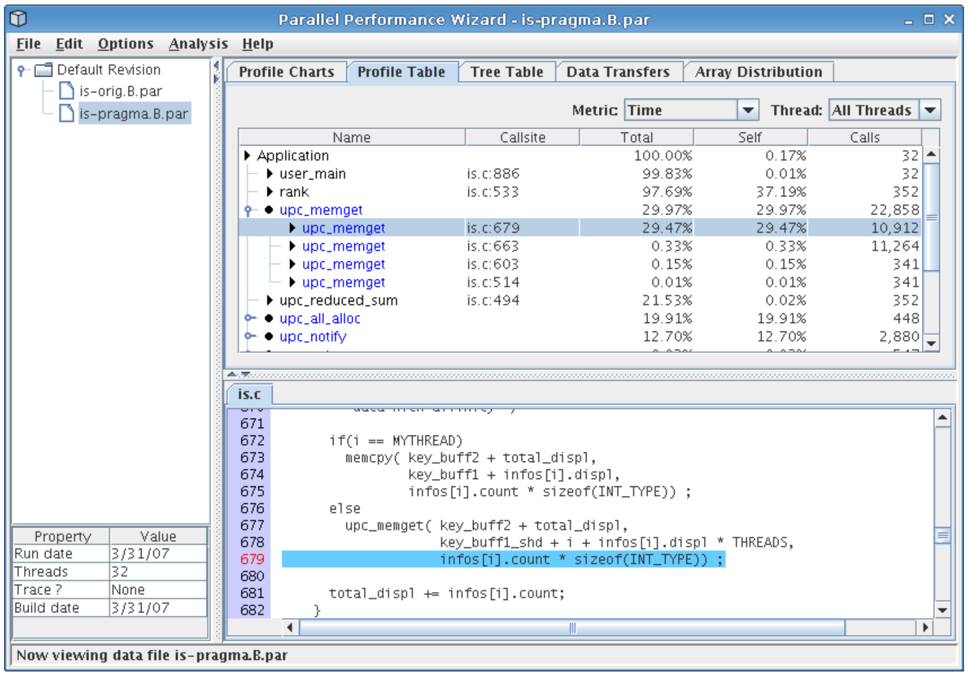


Region	Callsite	Max imbalance	Total
upc_reduced_sum	is.c:494	0.12818 s	4.15798 s
upc_all_alloc	is.c:494	0.11969 s	3.83856 s
upc_memget	is.c:679	0.05890 s	5.69105 s
upc_notify	is.c:684	0.05820 s	1.71007 s
upc_notify	is.c:1010	[illegible]	[illegible]

Figure 4.3: Load-balancing analysis on the IS benchmark

One novel feature provided by PPW is its load-balancing analysis, which you can run by accessing the "Analysis" menu bar. After the analysis runs, it will display a dialog similar to that shown in Figure 4.3. The analysis picks out lines of code where the time spent executing varies a lot from node to node, then displays these lines (sorted by severity) alongside a graph showing how time for that particular line of code was distributed among all nodes in the system. Looking at the figure, we can easily tell that calls to '`upc_reduced_sum`' end up taking longer on threads 1-30 but not much time on thread 0 and 31.

In addition to the load-balancing issues with '`upc_reduced_sum`' function itself, a call to '`upc_all_alloc`' within that particular function also had load-balancing problems. The '`upc_all_alloc`' function, as its name implies, is a collective routine requiring participation from all threads before the call will complete. Since the first and last thread spend the least amount of time executing this function, this means that all of the other threads wasted time waiting for these two threads to catch up to them. This explains the large slice for '`upc_all_alloc`' we saw in the profile metrics pie chart visualization.



Metric: Time Thread: All Threads

Name	Callsite	Total	Self	Calls
Application		100.00%	0.17%	32
user_main	is.c:886	99.83%	0.01%	32
rank	is.c:533	97.69%	37.19%	352
upc_memget		29.97%	29.97%	22,858
upc_memget	is.c:679	29.47%	29.47%	10,912
upc_memget	is.c:663	0.33%	0.33%	11,264
upc_memget	is.c:603	0.15%	0.15%	341
upc_memget	is.c:514	0.01%	0.01%	341
upc_reduced_sum	is.c:494	21.53%	0.02%	352
upc_all_alloc		19.91%	19.91%	448
upc_notify		12.70%	12.70%	2,880

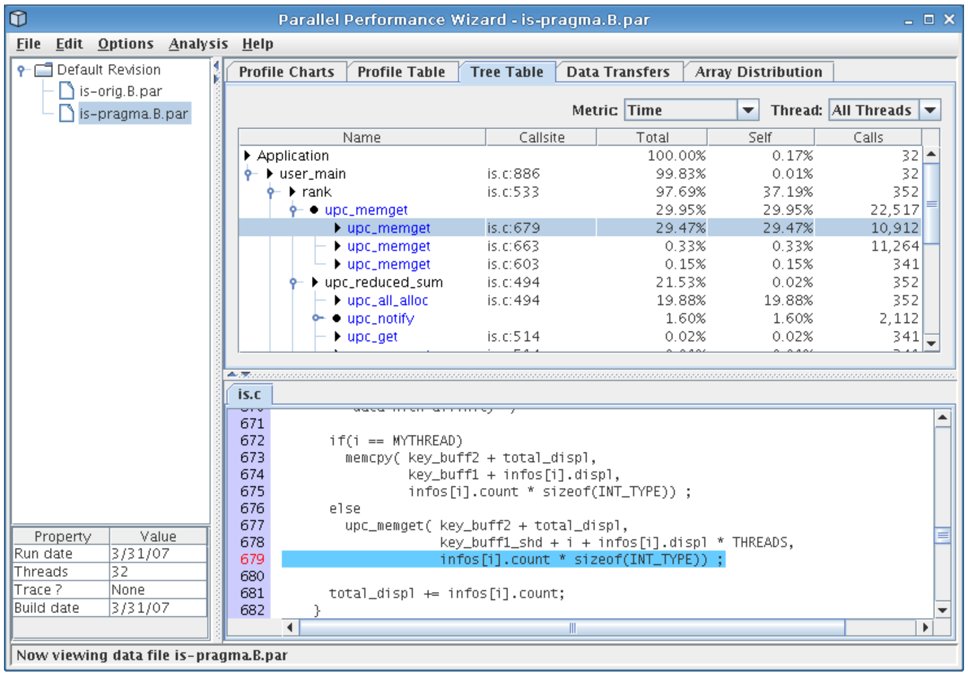
Property	Value
Run date	3/31/07
Threads	32
Trace ?	None
Build date	3/31/07

```
671
672         if(i == MYTHREAD)
673           memcpy( key_buff2 + total_displ,
674                   key_buff1 + infos[i].displ,
675                   infos[i].count * sizeof(INT_TYPE)) ;
676         else
677           upc_memget( key_buff2 + total_displ,
678                       key_buff1_shd + i + infos[i].displ * THREADS,
679                       infos[i].count * sizeof(INT_TYPE)) ;
680
681         total_displ += infos[i].count;
682       }
```

Figure 4.4: Profile table visualization for the IS benchmark

Moving on to some of the other visualizations, selecting the profile table visualization (in the tab row on the top of the GUI) should bring up a screen similar to Figure 4.4. The profile table reports flat profile information, similar to what you'd expect to see from a tool like `gprof`. However, unlike sequential tools, PPW aggregates data from all threads in the run together to show you an overall summary of how time was spent in your program, and will group similar operations together instead of showing a flat table.

From the figure, we see that the call to '`upc_memget`' on line 679 accounted for nearly 30% of overall execution time, which explains the large slice for '`Get`' operations we saw in the operation types pie chart. (You can switch on the percentages display by choosing it from the "Options" menu at the top of the GUI window.) Inspection of the code around the '`upc_memget`' callsite shows it is from an all-to-all operation implemented with several calls to '`upc_memget`'. Furthermore, this callsite also shows up on our load-balancing analysis as having an uneven distribution on all nodes, so this particular all-to-all operation could benefit from further tuning.



Property	Value
Run date	3/31/07
Threads	32
Trace ?	None
Build date	3/31/07

Name	Callsite	Total	Self	Calls
Application		100.00%	0.17%	32
user_main	is.c:886	99.83%	0.01%	32
rank	is.c:533	97.69%	37.19%	352
upc_memget		29.95%	29.95%	22,517
upc_memget	is.c:679	29.47%	29.47%	10,912
upc_memget	is.c:663	0.33%	0.33%	11,264
upc_memget	is.c:603	0.15%	0.15%	341
upc_reduced_sum	is.c:494	21.53%	0.02%	352
upc_all_alloc	is.c:494	19.88%	19.88%	352
upc_notify		1.60%	1.60%	2,112
upc_get	is.c:514	0.02%	0.02%	341

```

        if(i == MYTHREAD)
          memcpy( key_buff2 + total_displ,
                  key_buff1 + infos[i].displ,
                  infos[i].count * sizeof(INT_TYPE)) ;
        else
          upc_memget( key_buff2 + total_displ,
                      key_buff1_shd + i + infos[i].displ * THREADS,
                      infos[i].count * sizeof(INT_TYPE)) ;

        total_displ += infos[i].count;
      }
```

Now viewing data file is-pragma.B.par

Figure 4.5: Tree table visualization for the IS benchmark

Finally, the last stop on our whirlwind tour of PPW's main visualizations is the tree table, which you can access by clicking on the tab row at the top of the GUI named "Tree Table." For the IS benchmark, you should see a screen similar to the one shown in Figure 4.5. The tree table is very similar to the profile table we discussed above, except that performance information is shown in the context of function calls rather than as a flat list. This visualization allows you to "drill down" through your application's call tree, adding contextual information to the performance data.

Examining Figure 4.5, we see that the rank function accounts for nearly all of the benchmark's execution time, of which nearly 30% of time can be attributed to the problematic call to '`upc_memget`' that we previously identified. Additionally, we see that the call to '`upc_reduced_sum`' is indeed being hindered by the collective call to '`upc_all_alloc`', as we previously hypothesized.

One useful feature of both the profile table and tree table is that you are able to double-click on any of the rows to bring up a window that shows how time for that particular row was distributed among all threads. This can be useful for eyeballing how a particular operation was distributed among all nodes in your system.

5 Generating Trace Data

While the profile data visualizations from the previous section are very useful in finding general performance problems in your application, trace data can be used to troubleshoot detailed performance problems that profile statistics may gloss over. Trace data can be thought of as a "log" of all program activity, in that events are recorded at runtime and are annotated with globally-synchronized timestamps. One of the more useful things you can do with trace data is export it to a timeline viewer such as Jumpshot or Vampir, so you can see *exactly* what is going on at a particular point in time during execution.

In order to get PPW to generate trace data, give the '`--trace`' to `ppwrun`, as in:

```
$ ppwrun --trace --output=is-trace.B.par srun -N 32 is.B.32
```

If you are collecting trace data to view with the Jumpshot timeline viewer, we suggest compiling without using the '`--inst-functions`' flag, as the amount of information being displayed can quickly become overwhelming, and excluding function calls from trace files can significantly reduce their size. This is entirely a personal preference, however, so you might experiment with your particular application to see which you prefer.

It is important to mention that one of the major drawbacks to tracing is that if left unchecked, trace file sizes can grow to become very large and unmanageable. Unfortunately, PPW currently does not have any fancy tricks for dealing with such large data files. We recommend that you trace your application only after identifying specific performance problems with a profile (as in the previous sections), and use the techniques outlined in Chapter 3 [Tweaking Instrumentation and Measurement], page 4 to cut down on the amount of data being collected.

The remainder of this tutorial will focus on analyzing trace data from our IS benchmark using the Jumpshot timeline viewer, which is free and comes bundled with PPW. To view trace data with Jumpshot, you first must open a PAR data file containing trace records (eg, run with the '`--trace`' option) with the PPW GUI, then choose "Export... SLOG-2" from the main menu. Choose a file in which to save the exported data (give it an extension of '`.slog2`'), then open up the newly-created SLOG-2 file with the Jumpshot viewer. If you have a working Java installation on your parallel machine, you can also use the command-line program `par2slog2` to do the conversion, as in

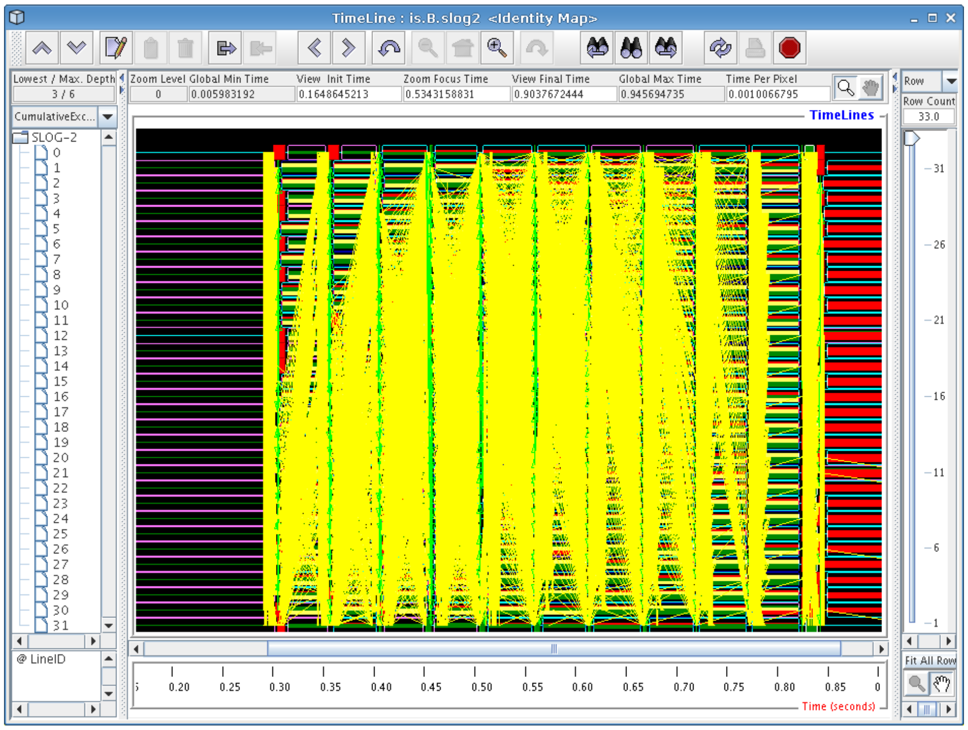
```
$ par2slog2 is-trace.B.par is-trace.slog2
```

This trace conversion process is CPU- and disk-intensive and may take a while for larger trace files.

In addition to the SLOG-2 trace format, PPW also supports exporting trace data to the OTF trace format, which is supported by newer versions of Vampir and the very cool Vampir-NG tool. For more information on these trace viewers, visit the Vampir website.

6 Browsing the Trace with Jumpshot

After you start the Jumpshot viewer, and open up your new SLOG-2 file, you will be presented with a multitude of windows.


TimeLine : is.B.slog2 <Identity Map>

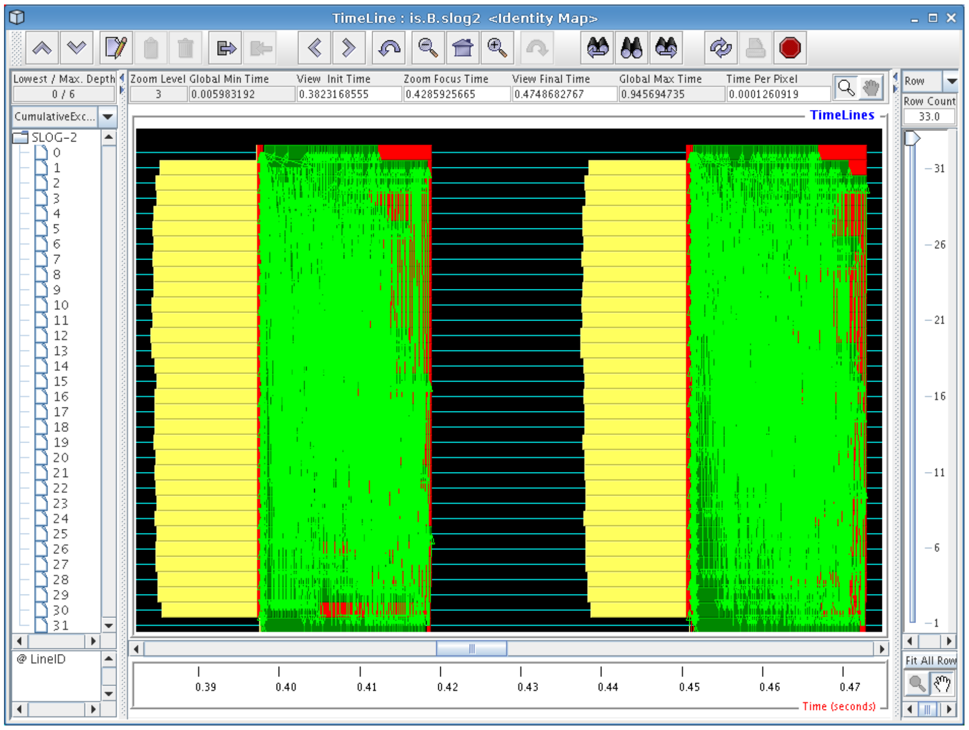
Lowest / Max. Depth: 3 / 6
Zoom Level: 0
Global Min Time: 0.005983192
View Init Time: 0.1648645213
Zoom Focus Time: 0.5343158831
View Final Time: 0.9037672444
Global Max Time: 0.945694735
Time Per Pixel: 0.0010066795
Row Count: 33.0

Figure 6.1: Jumpshot display of a 32-node IS benchmark run

The main window in Jumpshot is the timeline window, which is shown in Figure 6.1. This window is the one that displays the timeline view we discussed before, and is where all the action happens when working with Jumpshot.

When opening a large trace file with Jumpshot, you will notice some interesting things being displayed. In Figure 6.1, we see a bunch of strange-looking boxes and arrows. These correspond to what Jumpshot calls “Preview drawables”; they are simply Jumpshot’s way of summarizing a large amount of trace data for you. The summary boxes can be useful by themselves (very tiny histograms are drawn inside these boxes), but in general it is more useful to zoom in until you start noticing specific performance characteristics.

In our case with the IS benchmark, the overall screen doesn’t really tell us much so we’ll have to zoom in more closely to get an idea of what is going on. To zoom in, choose the magnifying glass tool from Jumpshot’s toolbar and click on the screen, or select a horizontal section of the timeline by selecting it with your mouse (drag with the button down). Try to zoom in on one of the blocks with a bunch of yellow mush around it.


TimeLine : is.B.slog2 <Identity Map>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 6	3	0.005983192	0.3823168555	0.4285925665	0.4748682767	0.945694735	0.0001260919

Row Count 33.0

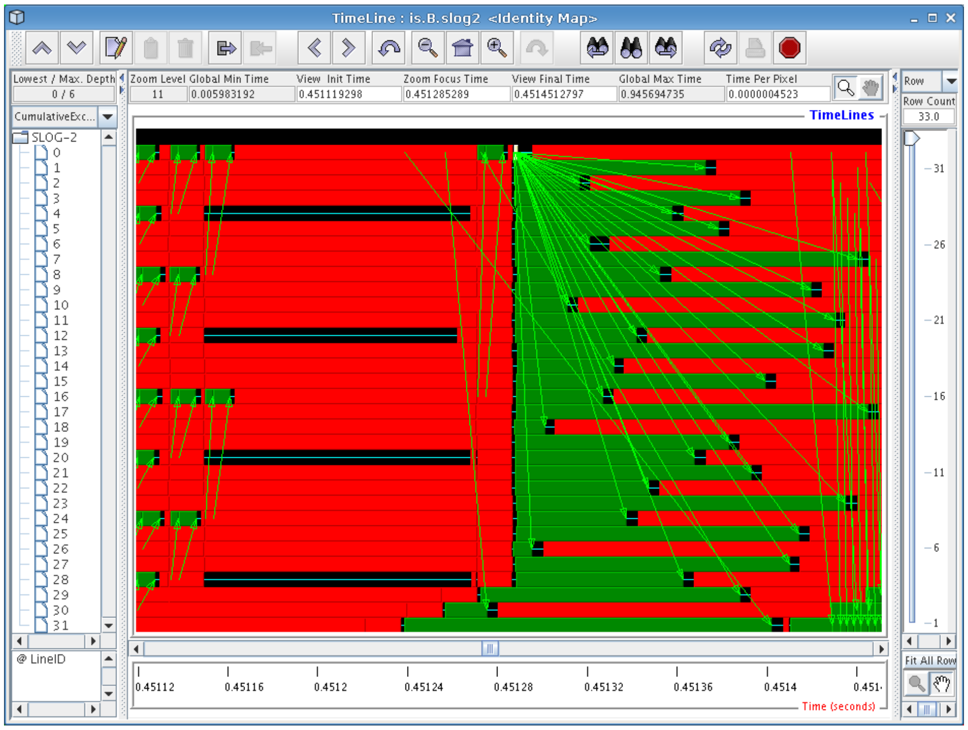
Figure 6.2: Jumpshot view of two iterations of the IS benchmark

If you zoomed in on one of the blocks closer to the center of the screen, you should see a window similar to the one in Figure 6.2. This is starting to look a little more sane, since Jumpshot is now displaying real trace records instead of summary previews.

If you've never worked with timeline diagrams before, then you might be wondering what exactly Jumpshot is trying to tell you with this sort of display. Essentially, what Jumpshot does is draw one line per thread in your trace data file. On these lines, Jumpshot will draw boxes and arrows that represent states and communication from your run. Since we've decided to exclude function information in our trace file, we assume that anything drawn as black area can be attributed to "computation" (roughly speaking), and so any communication or other language-level operations will stick out on the timeline display.

Looking at Figure 6.2, we see two iterations of the IS sort benchmark kernel, which have similar behavior. One interesting thing we immediately notice is the big yellow boxes on the left side of the screen. Looking at the function key (not pictured), we find that these boxes correspond to calls to '`upc_all_alloc`'. As we found before in our profile data, the first and last threads in the system spend less time executing these functions. Since the function is a collective one, we can infer that these two threads were "late", so we should look at the computational operations that immediately preceded these calls to figure out what is causing them to be slower than the other nodes.

One very useful tweak we've made to Jumpshot (by exploiting some interesting parts of its file format) is that by right-clicking on any particular box in the timeline window, a popup dialog will tell you exactly which line of code caused that particular operation. By right-clicking on one of the yellow boxes, we find that our old friend '`upc_all_alloc`' on line 878 is definitely the perpetrator of this performance problem.

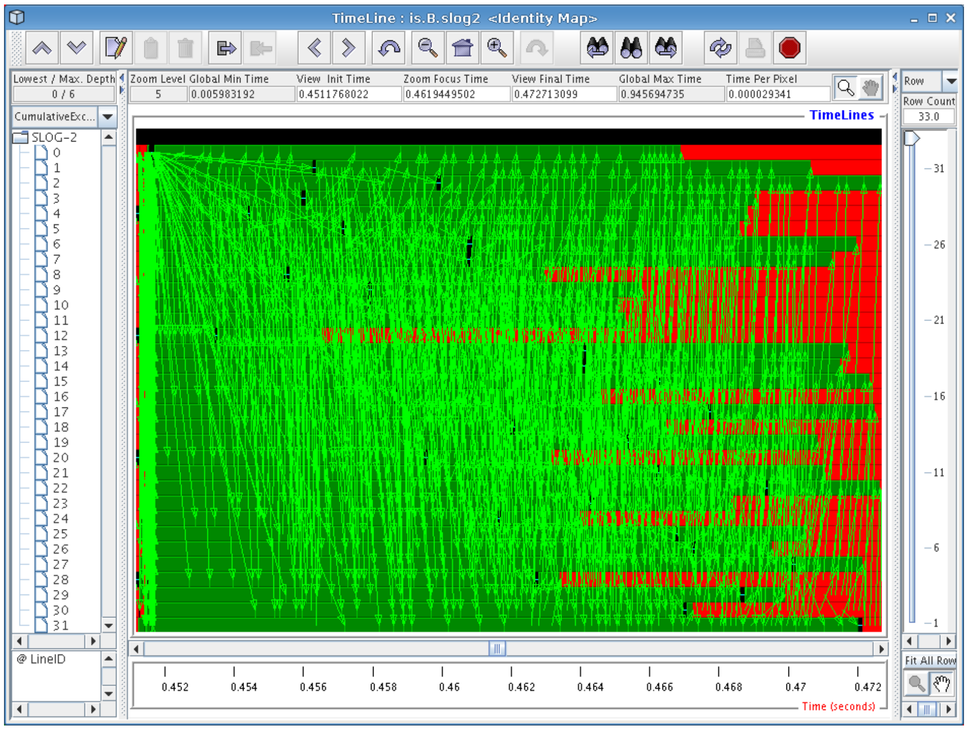


TimeLine : is.B.slog2 <Identity Map>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel	Row Count
0 / 6	11	0.005983192	0.451119298	0.451285289	0.4514512797	0.945694735	0.0000004523	33.0

Figure 6.3: Jumpshot view of a broadcast pattern

If you zoom in much closer by the red part of the trace file just to the right of the yellow '`upc_all_alloc`' boxes, you will see a display similar to that in Figure 6.3. One thing that we immediately see here is a broadcast-style operation being implemented by using a string of '`upc_memget`' operations. This particular benchmark was written before UPC had collective operations in its standard library, so rewriting the code to make use of these new collective operations might yield better performance.



TimeLine : is.B.slog2 <Identity Map>

Lowest / Max. Depth	Zoom Level	Global Min Time	View Init Time	Zoom Focus Time	View Final Time	Global Max Time	Time Per Pixel
0 / 6	5	0.005983192	0.4511768022	0.4619449502	0.472713099	0.945694735	0.000029341

Row Count: 33.0

Figure 6.4: Jumpshot view of an all-to-all pattern

Scrolling over a little to the right of the previous broadcast operation and zooming out a little bit will produce a display similar to the one shown in Figure 6.4. Interestingly, this mass of communication stems from a single line of code, the ‘`upc_memget`’ operation on line 679 of ‘`is.c`’. Our profile data tells us that this single line of code takes up 30% of execution time in our runs, and this diagram provides visual cues as to why this might be the case: implementing an all-to-all communication operation on a Quadrics cluster as a sequence of unscheduled get operations does not yield very good overall performance. As with the previous broadcast, using the new collective functions would undoubtedly squeeze more performance out of our Quadrics cluster.

In this tutorial, we’ve only covered a few of Jumpshot’s features. For more information on Jumpshot, refer to the frontend GUI reference chapter of the PPW user manual, or the Jumpshot user manual available from the Jumpshot website.

7 Summary

In this tutorial, we have shown how to use PPW to analyze the GWU UPC implementation of the NPB2.4 IS benchmark, and how to use PPW to generate trace data that can be viewed with the Jumpshot trace visualization tool. We hope that we have inspired you to try out our tool, which is available from the PPW website. If you found this information useful, you might check out the PPW user manual, which is also available from the aforementioned website.

Conspicuously absent from this tutorial are specific instructions on how to optimize this benchmark based on the information discovered from using PPW. This is left as an exercise for the reader :-)

Finally, we are actively seeking feedback on our tool. If you find PPW useful or annoying, please drop us a line at ppw AT hcs DOT ufl DOT edu.