Slide 1: Cover

In this paper we present an alternative to deal with the traces scalability problem, from the point of view of performance analysis tools.

Slide 2: Tools scalability

Nowadays, it is not surprising to have applications running in real production systems for hours and using thousands of processes. This keeps on increasing everyday, bringing us closer to the Petascale.

It is clear that, blind tracing in such scenarios produces huge volumes of data that, at best, are difficult to handle. And this problem is not exclusive to tracing tools anymore, because profilers also start to present issues at such scale.

Despite the drawbacks, our interest stays in traces because they provide far much information for a very detailed analysis.

However, the very first handicap these tools face is actually to store the data.

Even if you can, the amount of time required to process it can be very high. So if the tool is not responsive enough… How much time will the user wait for results before he looses all interest?

Slide 3: Work with large traces

So far, the way we've dealt with such big traces has typically consisted in filtering the information. Generally, we would:

1. Generate the full trace
2. Summarize it to get general understanding of the application behavior.
3. Focus on a small, interesting region with all details.
4. Iterate if necessary.

And this process was manually driven by the expert and done post-mortem.

So, what comes next is to automatize this process to be done on-line, so that you directly get the most interesting parts of the trace, getting rid of the tedious task of handling large files.

Slide 4: Objectives

Our objective is to produce traces of 100 Mb. And the number here is not important. What this really means is that we want to balance the trade-off between relevant information and raw data. Or in other words, we want the minimum amount of data that best describes the application.

And how can you obtain that? For this, you need the ability to intelligently select what is relevant, and discard what is not.

To this end, we've built an analysis framework that determines this automatically, and at run-time.

Slide 5: Modules integration

And this framework is structured in 3 components.

(1) First, we use MPItrace, which is a tracing toolkit that attaches to the application to obtain performance measurements. The typical information gathered by this tool is mostly hardware counters data at the entry and exit points of MPI calls. While in this work we've focused on MPI applications, it could be easily extended to other parallel programming models.

(2) Periodically, all data is aggregated in a central process using MRNet. This software interconnects a network of processes in a tree-like topology, and allows data traversing the tree to be summarized upon its way to the root, so as to preserve scalability.

(3) Finally, we use a clustering mechanism to analyze the structure of the application.

Combining these tools, we've built an analysis framework that is able to determine a relevant region of the execution. And for such region, generate a detailed trace plus other performance reports.

This system is generic, in the sense that all pieces are interchangeable. And this starts with different types of analyses that could be computed, but also we could use other tools to extract data from the application, or even use a different communication layer.

Slide 6: Modules interaction

Now let's see in more detail the way these components interact:

1) Instrumentation is local to every task and performance data is stored in memory buffers.
2) For each task, a back-end thread is created to be a leaf of the communication network. These threads stall, so that they do not interfere the execution of the application..
3) Periodically, the front-end pauses the application, wakes up these threads, and starts collecting data.
4) As said, data flowing the network can be reduced in the intermediate processes of the tree thanks to a filter functionality offered by MRNet.
5) This allows data to be aggregated in the front-end, to be analyzed having a global view of the state of all tasks of the application.
6) Results are propagated back to the leaves.
7) There, the tracing tool is instructed in what is the most interesting information to focus, and traces are locally generated.

Slide 7: Clustering analysis

The analysis carried out at the front-end node is based on clustering, which is an algorithm that makes groups of elements with similar characteristics.

Our elements are the computing bursts of the application (i.e. the sequential computations between MPI calls).

For these zones there's hardware counters data associated, and amongst all possible counters, we select 2 in particular to perform the analysis with.

- Instructions completed: Measures the amount of work
- Combined with IPC: Allows to differentiate zones with similar work, but different performance.

We use these 2 because they're good to bring insight into the overall performance, but any other combination of metrics is also possible. The objective, is that they reflect the structure of the different computing trends of the application.

Slide 8: Clustering results

Once the analysis concludes, we obtain several interesting reports:

First, a scatter plot that shows the structure of the computing regions of the application. In the example we can see 7 different types of computations.

Then, you can get the distribution of these clusters in a timeline, so you can see when these computations actually happen. In the example, you can see this application is very SPMD, as all tasks perform the same type of computation at the same time.

Also, you get a list of detailed performance statistics per cluster.

And finally, links to the code to focus on sources when you detect a problem.

Slide 9: Tracking evolution

How we use the clustering? We use it to track the evolution of the application.

At fixed intervals, or whenever a given volume of new performance data is produced, a new analysis step triggers.

In this way, we obtain a sequence of clusterings, that can be compared to see changes in the application behavior.

This is repeated, until we find a representative region of the execution. For this region, a detailed trace is finally produced.

How we decide a representative region?

Slide 10: Select representative

At every analysis step we compare the previous state of the application with the current one, and see if it still behaves the same.

To do that, we check cluster per cluster if they have the same shape, size and position. This can be seen as inscribing them into a rectangle, and match those that overlap, with a margin of variance.

In the example we that the cluster at the top-right would match its partner, but the one on the left would not, as it splits and its shape and position changes.

If the matched clusters represent more than a given threshold of the total execution time, we consider the application stable.

And what we're actually looking for is the application to stay stable N times running.

Once this region is found, we produce a detailed trace for it. At this moment the system can stop, or it can continue monitoring the application, looking for other representative regions with different structure.

If the application has a lot of variability and a stable region can't be found, the requisites are gradually lowered in an attempt to catch the best possible region.

Slide 11: Evolution of MILC

Here we can see an example of the evolution track of the SPECMPI benchmark MILC.

As you can see, the state of the application keeps on changing during the initial phases, while the initialization takes part, until it enters the main computing loop. Then, the application remains stable in the following steps and so on.

So for this region, a trace is generated, which in this case, comprises 6 full iterations with all the details. This includes MPI calls, communications, hardware counters, callstack information, plus the clustering structural information which is also incorporated into the trace. And all this sizes only 60 Mb, down from 5 Gb for the full trace.

Slide 12: Clustering vs. Classification

Since this process is done on-line we need it to be quick, but clustering time grows large the more points to cluster, so we need to keep this under control.

The strategy we follow is to cluster a small subset of points only (use a training set of data). For this we've tried different alternatives:
- Get samples across space: Get the full time sequence of just a few representative processes.
- Take random samples from all processes across time.
- Combine both sampling methods, which provides a better chance to capture variances both in space and time.

The rest of data is classified using a nearest neighbor algorithm, which compared to clustering is much faster. In this way, we can keep the analysis times low and more or less constant.

Slide 13: Clustering vs. Classification

The effect of the different sampling criteria for the classification can be seen in these trace timelines.

Left column shows the results of sampling across space with different numbers of processes. The first picture corresponds to a run where all data was clustered, so that can be considered the "perfect" result.

On the right we have the same, but taking samples from all processes across time.

As we reduce more and more the number of samples, results get less precise. This doesn't mean they're bad, as you can still see structure in there, but with a lower level of finesse.

Anyway, combining both sampling methods you can virtually get the same results as in the reference, while processing much less data. In this example, 75% less data was processed, which took 6s down from 2m.

Slide 14: Experiments

We've tested this system with a variety of benchmarks, real applications and processor counts. This table sums up some of the results. Some interesting facts that worth mention:

- We obtain trace size reductions up to two orders of magnitude. It is the user who specifies the size for the final trace according to his needs. In these examples, we decided to scale the size with the number of processes, but the idea is to have this adjusted automatically.

- Second, we obtain results before the application finishes.

- We get small, manageable results, and we get them relatively fast, and you might wonder, are they representative compared to the full trace?

Obviously, with such reductions there's an inevitable data loss. But In the examples, you can see the final traces comprise a good number of iterations of the application. So, the question is, for example in the case of Gromacs, are these 10 iterations enough to represent its overall behavior?

Slide 15: Quality of the trace segment

The answer is yes.

To ensure that, we have compared our results with an external tool with its own quality checks, so as to reassert our measurements.

We've generated TAU profiles for the whole run, and compared them to the traced region.

What we've seen is that the main user functions detected are all the same. And they have the same average counter values and execution time percentages. Most differences are under 1% (considering different tools' overheads). So in all cases, the trace segment seems to be a fair representative for the rest of the execution.

Slide 16: Example: GROMACS evolution

Now let's see a very short example of how this system can be applied to the analysis of a real application to get better understanding of it and detect interesting performance issues.

Here we see the structure of Gromacs application at the different analysis steps. You can see it is pretty much the same all the time, with the exception of the green cluster that sometimes splits.

This can happen both because small variations of the application, or because the points in the middle that make the subclusters merge are not very frequent and might not get sampled.

Anyway, the application stays very stable the next steps, and finally this region is selected as representative.

Slide 17: Example: GROMACS structure

If we focus on this region, we can easily identify the main computing trends of the application.

Gromacs is a non-SPMD application with 2 types of processes. Half perform heavy computations (high Instructions & IPC), which correspond to clusters Green and Yellow. The rest perform smaller, fast computations and are represented by the other clusters.

While Green seems to stretch well-balanced (the more Instructions, the more IPC), Yellow presents fluctuations in Instructions with constant IPC, which is a sign of potential imbalance.

If we look in detail into the small computations, we can see again both Instructions and IPC variances between different clusters.

If we correlate this plot with the timeline for the subset of processes performing these small computations, we can verify that slower and faster computations take place simultaneously, resulting in a duration variance.

In this case, a recommendation that could be made to the user would be to study the load-balancing characteristics of these particular regions that are affecting performance.

Slide 18: Increasing scale

Our initial development had a few characteristics that made it good enough for runs up to 500, 1000 processes, but would not scale very well at large processor counts.

All data was centralized, and sampling, clustering and classification were done at the analysis node. But this implied too much pressure on the front-end node, in terms of memory consumption and computation.

Thinking of runs with 10k tasks and more, we changed the scheme so that:

(1) Sampling is done at leaves.
(2) Only the data that is actually going to be clustered is put together in the front-end.
(3) Results are broadcasted, and classification is done independently at the leaves.

These changes reduce a lot the pressure on the front-end, and should keep the system scalable for any number of tasks.

Slide 19: Conclusion

In conclusion,

- We've presented an analysis framework that is able to identify the structure of the application as it runs, and see how it evolves.
- Through this mechanism, it automatically selects a representative region of the execution.
- As a result, we generate for this region a detailed yet small trace, plus periodic performance reports. All together, they provide both general and detailed insights about the application behavior.
- The obtained trace size is much smaller compared to the whole, as it comprises just a small sample of the iterative behavior of the application. We could go on reducing the trace size in the space dimension by selecting representative processes, which is the typical use for the clustering.
- Finally, the whole infrastructure is scalable and generic, and enables the use of other analyses towards the intelligent selection of information.

Actually, this is one of our current lines of work:

- We're adding a mechanism based on spectral analysis to delimit much more accurately the traced region.
- Also, we want to develop a parallel version of the clustering algorithm that runs on the intermediate processes of the MRNet tree rather than the root, so we can cluster more points, faster.
- We'd also like to tune a bit more the stability heuristic, as we've seen the sampling process might introduce false negatives that can be detected.