MPI+OpenMP Performance Analysis tips

This document poses some questions a performance analyst may face when looking at a Paraver trace. We describe how the tool can be used to identify specific behaviors or performance properties.

Paraver is a generic browser of performance data stored in a generic trace format. Different tracing packages are targeted to different programming models or platforms. The actual analysis that can be performed is dependent on the type of information captured and emitted by the tracing package. This document concentrates on the analysis of traces produced by the OMPItrace package for tracing MPI and OpenMP programs. Because traces for real programs and platforms can be very large, a set of utilities have been developed to summarize the information. These utilities generate events with some encoding convention. This document also describes how to use and analyze the scalability features supported by such utilities.

A separate document is devoted to each of the other encoding conventions used by CEPBA-Tools tracing packages and translators such as LTT, AIXtrace, etc.

In all the descriptions in this document it is assumed that you have experience with the navigation features of the Paraver display windows as well as some knowledge of the basic concepts of the filter, semantic, display and 2D/3D analysis modules. If not, please refer to the Paraver navigation and concepts.

The document is structured in two major sections. The first one will give methodological guidance on the basic analysis of parallel programs. The second section describes how to handle large traces deriving form large platforms and/or program runs. The following index will lead you directly the section describing how to answer a relevant question that may arise during a performance analysis. Navigating through the cross reference links you will be able to traverse a broad exploration space in order to better understand the performance of your application. For each of them you will get a reference to one Paraver configuration that should directly address the issue.

Table of Contents

1.Basic analysis

1.1.Global performance and profile

A first set of question when looking at a parallel program is which are the main routines in the application and how much parallelism are we obtaining.

1.1.1.Which are the user routines taking most of the time?

An initial analysis of a trace should be targeted to the identification of routines of the code where a relevant fraction of the time is spent. At the level of user routines this will point to regions where to further focus attention and optimization efforts. To obtain the typical user functions profile use configuration file: General/analysis/uf_profile.cfg.

WARNING: Applicable only if user function events available in the trace. Only functions instrumented appear in the table. Such instrumentation may have been automatically inserted (driven by the file pointed to by environment variable MPTRACE_ADD_FUNCTIONS) with dynamic instrumentation tracing packages. For static tracing packages explictis call to ompitrace_event (60000019, #routine|0) must have been used.

What you will see: A table with one row for each thread and one column for each user function. The value indicates the time inside such function (exclusive times between the instrumented routines). Dark blue indicates a high value, light green a low value.

Some interpretation capabilities:

- You will get the typical information provided by profilers. It is given on a per thread basis. It lets you identify the most relevant functions in the application.

- As the profile is reported per thread, load imbalance may show up in variations across processes for one routine. Care has to be taken to properly identify whether it is a real computational load imbalance.

Further uses of the configuration file:

- This configuration file can be used as reference for computing a bunch of other related statistics such as

1. **#of invocations** to the routine by changing the statistic in the 2D window to "# Bursts"

2. **average duration of those invocations** by changing the statistic in the 2D window to "# Bursts"

3. or to obtain other **advanced hardware counter profile** statistics such as average MIPS or miss ratios within the routine. Load the configuration file for your hardware counter based metric of choice (see section 1.6). Then change the statistic in this 2D window to "Average Value" and choose your desired hardware counter based metric on the data window selector of the 2D window. In this case such an average MIPS would be computed including the MIPS achieved within the MPI calls made by the user routine. If you are only interested in the MIPS in the sequential part that is under the control of the application developer you should use the reference profile of user functions excluding MPI.

- If you are interested in generating a view where only a specific user routine appears you should use the "Open Control Window Zoom 2D" button on the 2D and select the column of the routine you are interested in.

1.1.2. What is the global efficiency of the application?

Configuration file ***General/analysis/avg_procs.cfg*** can be used to get a single number of the performance of the application.

What you will see: a single entry 2D window reporting the **average number of processes performing useful computation** out of the total number of processes.

Some interpretation capabilities:

- The number reflects the efficiency on the application in terns of its parallel execution. Ideally, the value should be as close as possible to the total number of processors for good performance. The higher the number the more CPUs are kept active working on computational parts of the problem. It does not state whether the CPU is used efficiently (relative to its peak performance) so further analysis with hardware counter based metrics would be recommended even if this statistic is good.

Further uses of the configuration file:

- The instantaneous value of such metric will probably vary with time. By clicking on the button by the name of the control window you will see a timeline with the instantaneous parallelism profile of the application. On this view you can identify periodicities of the application and intervals of poor performance.

1.1.3.What is the instantaneous parallelism profile of the application?

Configuration file ***General/views/instantaneous_parallelism.cfg*** displays the total number of processes performing some useful computation at each point in time.

Some interpretation capabilities:

- This view will point out to regions of poor performance. Ideally a user would like its application to have a constant number of active processors equal to the allocated number of processors. Regions of low value of this metric should probably be analyzed.

Further uses of the configuration file:

- This is a first view that may be helpful to identify patterns in the application behavior such that later analyses can be performed on just a subset of the trace. A zoom on a relevant region of trace can be made on this view and then paste the scale to other timelines and statistics. The analyst could also generate based on the structure identified in this view a new trace (Tracefiles->TraceGeneration) on which to perform subsequent analyses.

1.2.Load balance

A first factor that will determine the performance of a parallel application is the balance of the load between processes. The following questions/descriptions can help an analyst identify the nature of such load distribution by looking at the global and local distribution of the computational chunks between communications.

1.2.1.Measuring computational load imbalance

To determine whether different processes do compute for different amounts of time inside one routine use configuration file ***General/analysis/uf_excl_MPI_profile.cfg***.

WARNING: Applicable only if user function events available in the trace. Only functions instrumented appear in the table.

What you will see: A table with one row for each thread and one column for each user function. The value indicates total user level computation time within such function (excluding other traced functions it may call). This profile is similar to the one in section 1.1.1, but without including in the statistics the time in MPI.

Some interpretation capabilities:

- Typically an analyst would expect an application to be load balanced within each major user function. This would show up in this table as all rows of any given column having a roughly equal total computation time. Although typical, this is not mandatory (for example: a well balanced application might have half of its processes performing the computation inside one routine and half of them inside a different routine)

- Even if the routines of an application are globally balanced, the application might experience microscopic load imbalance. This would be the case where different invocations of a routine do show load imbalance but the threads that take longer in one invocation take less in other. A barrier or collective operation at some point between the two invocations (or within the routine) would imply that the individual load imbalances impact the application performance.

Related approaches:

- Load imbalance can often be indirectly detected by looking at collective MPI calls.

- Configuration file General/analysis/uf_profile.cfg is a similar profile of time within each user function, but also includes the time inside MPI. This is also an exclusive profile (does not include time in called functions).

1.2.2.Migrating load imbalance in SPMD patterns

An SPMD iterative program may show a type of load imbalance where at each iteration there is a different process with more load than others, while still at the global level the load is fairly balanced. To try and identify this situtation for a sepcific user funciton you can use configuration file ***General/analysis/load_balance_for_specific_uf.cfg***.

WARNING: Applicable only if user function events available in the trace. Only functions instrumented appear in the table.

WARNING: Assumes an SPMD structure where all processes enter the user routine at about the same time. Skews in the invocation of a routine across processes will actually be considered as load imbalance.

What you will see: A table with two columns. Column 1 represents the specific user routine, while column 0 displays the same metric for the rest of the program. Each entry computes the percentage of time the thread is active within the total span of the parallel function.

To select a specific user function of interest you have to change the value in the “In Stacked Val Parameter” selector of the semantic module of view “Some process in specific user function”. The value should be the identifier of the routine of interest.

Some interpretation capabilities:

- A value less than 0.5 (light yellow) in an entry indicates that such thread has consistently been inactive during at least 50% of the duration of the function.

- Dark blue measn the thread has been consistentily active (executing user code) during the whole duration of the function.

- The average value of a column may be considered as a global indicator of efficiency. The closer to 1 the better.

- The Stdev across threads is reported in the summary row and is a good indicator of global imbalance. The smaller the value the more balanced the global distribution of work across threads is.

- If the column has a value no greater than a threshold it may indicate microscopic load imbalance even if the Stdev of the metric between threads is small. What this actually corresponds is to floating load imbalance, where a different thread takes more time than others each instantiation of the function..

1.2.3.Histograms of computation burst

Configuration file ***General/analysis/3dh_ufduration.cfg*** can be used to obtain a very detailed

WARNING: Applicable only if user function events available in the trace. Only functions instrumented appear in the table.

What you will see: A table showing a histogram of the duration of the different CPU bust between successive events. Typically this will be between entry/exits of user functions and between these and the MPI calls. The histogram you will see corresponds to one specific user function. To change the user function, use the selector at the bottom right of the 2D window.

Some interpretation capabilities:

- A histogram is the ultimate way to present the distribution of computation times. Even if it does not give a single scalar metric that reports the goodness or badness of the property it does easily give a detailed qualitative perception of the nature of the application. For each routine you expect to see vertical stripes (most probably several) representing that all threads take the same time in each computation phase within the routine. Load imbalance will show up as bent lines.

1.3.General Communication Performance

Message passing and global operations are a need to achieve the cooperation of the different processes in a parallel program, but frequently the first point to blame in case of poor performance. The following questions and descriptions should help an analyst to properly identify to what extent is the communication a real bottleneck for the application performance.

1.3.1.Is MPI taking a significant fraction of the time?

To identify potential problems due to communication and synchronization overhead use: ***mpi/analysis/mpi_stats.cfg***.

What you will see: A table with one row for each thread and one column for each MPI routine used by the program. The value indicates percentage of the time inside such MPI call. Additionally, column 0 (End) indicates the fraction of time in user level code. Dark blue indicates a high value, light green a low value. Changing the gradient scale representation may highlight differenced between processes (load imbalance reflected in MPI calls).

Some interpretation capabilities:

- Large values in one MPI call may draw our attention to it. Before thinking of modifying the structure of the source code it might be interesting to further investigate the performance of the MPI calls in their various aspects: overhead per byte, end to end communication performance, ...

- Variations across processes in column 0 probably indicate computational load imbalances.

- If column 0 has values much larger than other columns and you are not interested in it, you might want to eliminate it (modify the Min box in the lower left corner of the window to 1). By fitting the color gradient you will now get a better feeling of relative difference between MPI calls and across processes.

Waits for message reception due to imbalances or externally caused delays (i.e. preemptions) that propagate through the communication dependence chain.

Further uses of the configuration file:

- By clicking on the open button by the control window name you will see the timeline with the actual sequence of MPI calls performed by the application. This can be directly loaded from ***mpi/views/MPI_calls.cfg***.

Directions for further investigation:

- **Link to source code**: Is all the problem in one routine? In many of them? Which one(s)? Use configuration file ***mpi/analysis/uf_fraction_of_MPI.cfg*** which is derived from the uf_profile. This cfg reports for each user function the fraction of time (between 0 and 1) it is inside MPI, not performing useful computation.

- **Point to point or collectives**: Depending on the numbers obtained, you may wish to focus on point to point or collective calls. In the case of point to point, the first issue to look at is probably whether the ratio of time they take to amount of bytes they move is reasonable (section 1.4.1). It may also be interesting to check the transfer bandwidth and network utilization (section 1.4.2). A further evaluation of the collectives is described in section 1.5.2.

- If you are very curious, you may be interested in analyzing **hardware counter information**

within the MPI calls themselves. You should firts load your hardware counter metric of choice (i.e. Loads). Then modify in the above 2D window the statistics selector to your desired choice (i.e. Average value) and the data window selector to the just loaded hardware counter view. Finally click on the Repeat button and you will get the corresponding statistic for each MPI call (i.e. Average number of loads within each MPI call). **WARNING**: your trace must include hardware counter events on entry and exit of MPI calls (OMPItrace option -counters:mpi)

1.4.Point to Point Communication Performance

1.4.1.Are point to point MPI calls performing efficiently from a process point of view?

Even if an MPI routine is taking a lot of time, the question is whether the routine is behaving as expected performance wise. As developers/users, we may be ready to accept the overhead of MPI calls as long as the service we obtain from them falls within the conceptual performance model we have of it. For example, we expect isends to take minimal amount of time, we may accept sends to take a time inversely proportional to the nominal bandwidth of the system, ….

To assess the local performance of an MPI point to point call use configuration file ***mpi/analysis/3dh_bw_per_call.cfg***.

What you will see: A histogram with one row for each thread and one column (bin) for each range of “local cost” of the point to point MPI call. By “local cost” we mean the ratio of microseconds per byte sent/received by the MPI call. This “cost” does not consider the transfer time for that data. It should be seen as a relative measurement of the overhead the call introduced in the sequential execution of the program that called it. The value in an entry indicates the percentage of the time at the corresponding local cost range (computed over the total time inside the MPI call). The histogram you see corresponds to a specific MPI call. Change the selector at the bottom right of the window to analyze the behavior of the MPI call you are interested on.

Some interpretation capabilities:

In general, the histogram should be useful to identify MPI call invocations with different types of behaviour as well as outliers. By clicking the “Open Control Window Zoom 2D” and selecting a range of interest within the 2D histogram table you will generate a new display window where only the calls of the selected local cost will appear. You may thus look at calls with expected behavior, as well as strange (i.e. very poor) behavior regions. By correlating the scales of these selective views to the user function or MPI call vies you may identify where in the source code is the problem or

which communication pattern results in the obtained behavior. The following bullets detail what should be the expected behavior for different MPI calls

- For immediate calls you should expect very small values as they should be of minimal duration irrespective of the message size. In many MPI implementations it is frequent to find that some instances of immediate send or receive calls do take significant amounts of time within the call, the run time does perform the transfer (may even be a transfer for a different MPI call). It is often possible to understand quite a bit about unexpected behaviors and the internals of the MPI implementation by further analyzing the trace (i.e. to find out that an isend gets unexpectedly blocked till the receiving thread leaves the user level computation and calls again MPI) but this is left for an experienced Paraver analyst.

- For normal send calls you may experience very low values possibly indicating blocking send to a late receiver(possibly because it could also be due to preemptions,....), high values (in the order of the inverse of memory bandwidth for memcpy operations within the node) if the send is buffered or values around the inverse of the nominal MPI point to point bandwidth for blocking sends where the receiver has already arrived.

- For normal recv calls you may experience very low values possibly indicating a late sender (possibly because it could also be due to preemptions,....), high values (in the order of the inverse of memory bandwidth for memcpy operations within the node) for buffered sends or values around the inverse of the nominal MPI point to point bandwidth for blocking sends where the sender has already arrived.

- The histogram may also be used to identify poor MPI implementation features? Always difficult to blame the provider. Microbenchmarks?

Further uses of the configuration file:

- If you are interested in the number of calls at each cost range (actual histogram) you can see it by changing the statistics selector to “# Bursts”.

- Sometimes, it is interesting to look at a related histogram using duration of the call instead of bandwidth. This is presented by configuration file ***cfgs/mpi/analysis/3dh_duration_per_call.cfg***. Although the duration of a call does not directly provide an indication of whether it performed properly or not, this histogram typically shows a much more detailed separation between calls performing poorly (very wide ranges of duration that actually map to a very thin/small bandwidth range). Using the

"Open Control Window Zoom 2D" on this 2D can thus be used identify very specific ranges of durations of calls.

- Other interesting analysis based on the previous cfgs is to change the statistic to "average value" and use as data window "bytes betw. Events" (automatically loaded as part of ***mpi/analysis/3dh_bw_per_call.cfg*** or can be directly loaded from ***mpi/views/point2point/p2p_size.cfg***). The table should show the correlation between bandwidth/duration of the call and the message size.

1.4.2.Is the end-to-end performance of point to point MPI calls appropriate?

Configuration file ***mpi/views/point2point/s_r_bandwidths.cfg*** displays the actual amount of communication bandwidth used by each process in the application (one view for send bandwidth, one for the receive bandwidth and one for the sum).

What you will see: Timelines reporting for each thread at each point in time the equivalent bandwidth (incoming/outgoing/total) of all point to point message transfers the thread is involved (you may have to open the incoming and outgoing views from within the "Visualizer Module" window). For each message, the equivalent bandwidth it contributes during the duration of the transfer is equal to the ratio between the message size and the duration of such transfer. Every message contributes with such value both to the sender thread on the "Send Bandwidth" view, to the receiver thread on the "Recv Bandwidth view". The "Process Bandwidth view" is the point wise addition of the other two.

Some interpretation capabilities:

- The send or receive bandwidth are physically limited by the total bandwidth a process can inject/drain from the network. From the knowledge of the architecture we can infer what this value should be and this is a fair reference against which to compare the observed metric.

- The patterns of the send and receive vies may be similar or different. A similar pattern indicates an application where the communication structure is relatively homogeneous (exchanges between pairs of processes, shifts,…) Differences in the pattern will show up in cases with differentiated behavior between processes (some senders/some receivers). If the number of senders of receivers is small this view may point to potential bottleneck processes.

Further uses of the configuration file:

- We can compute a histogram of either send or receive bandwidth and identify how close or far from the theoretical limit is what the application demands/achieves. Using the “Open Control Window Zoom 2D” button we can identify in the timeline regions of a specific range of interest (good or bad performance).

1.4.3.What are the message sizes used by point to point calls in the program?

A typical concern when an MPI program does not scale is that it may be using many small messages. We may for many reasons also be interested in finding which message sizes are used by the application. This is computed by configuration file ***cfgs/mpi/analysis/point2point/3dh_msgsize_per_pt2pt_call.cfg*** can be used.

What you will see: A histogram with one row for each thread and one column (bin) for each range of message sizes of the point to point MPI calls. There is actually one such histogram for each MPI point to point call. The selector “Fixed Value” at the bottom of the 3D window can be used to select the desired MPI call. The value in an entry of the table indicates the total number of messages within that range of sizes sent/received by the selected MPI call.

If a program uses very large message sizes, the bins of the histogram will be very large, and several message sizes may fall in one such bin. If you are interested in differentiating message sizes in a small range you may use the 2D zoom capability. Applications tend not to use many different message sizes, so you expect to see a few vertical stripes in the 2D table. You can use the “Hide null entries” button of the 2D window and small bin ranges (delta) to get a table with just one column for each one of the message sizes used by the program.

Some interpretation capabilities:

- **Overhead of small messages?** A lot of very small message sizes is normally associated a lot of overhead. By changing the statistic to “% Time” you will get in each entry the percentage of the total time that MPI calls of message sizes within the corresponding bin take. If this percentage is not large, you should not expect improvements in performance by packing small messages in larger ones.

- Typically, send and receive MPI calls will match each other and thus the same message sizes histograms are expected.

- Other frequently used communication structure consists of an MPI_waitall completing the

reception of several sends. The use of such pattern can thus be identified if the histograms for send and waitall calls are different.

- For MPI_SendRecv calls the reported size is the total send and receive size.

Further uses of the configuration file:

- **When are specific message sizes used?** You can use the “Open Control Window Zoom 2D” zoomimg capability to identify in a timeline when a specific range of message sizes being used. The view that pops up uses the max draw mode for each pixel so that you can identify such MPI calls even if very few and scattered in time. The view will nevertheless appear totally coloured if you selected small messages and they are frequent in an interval (or the whole trace). In this case you may want to zoom the timeline to differentiate the individual instances. You may be interested in checking out whether they are uniformly distributed along time or mostly clustered within a specific region or routine. Synchronizing such view with the user functions view will show such information. If you want to get a timeline of only MPI calls for messages of the selected size you can apply the “sign” compose function to the new window and derive from the MPI call view by multiplying them.

Directions for further investigation:

- A very closely related configuration file is ***cfgs/mpi/analysis/3dc_msgsize_totbytes.cfg***. In this case each entry in the table corresponds to the total number of bytes sent/received at the specific message size range. It lets you identifies the message sizes at which most of the data is moved.

1.4.4. What is the system bandwidth the application uses?

Configuration file ***mpi/views/point2point/total_bw.cfg*** can be used to visualize the instantaneous amount of communication bandwidth used by point to point calls in the application.

What you will see: A timeline computed by summing at each point in time the equivalent bandwidth of all point to point message transfers taking place at that time. Each point to point transfer thus contributes to such function from the moment the sender invokes the send primitive till the receiver gets the message with a magnitude equal to the ratio between the message size and the duration of such transfer.

Some interpretation capabilities:

- The metric quantifies the equivalent bandwidth a network should have provided not to delay the execution. It is NOT the actual network bandwidth of the real system (for example a load imbalance in the application may result in a low bandwidth as there may be a lot of time available for the transfer, even if the real system performs a very fast transfer between processors and then the message has to wait for the reception to arrive). The metric IS and indication of what the application demands or is able to achieve in the real system. In regions where the demand is very high the metric will saturate if the limit of the network is reached.

- The view is thus useful to identify regions where the network bandwidth may actually be limiting the application performance.

Directions for further investigation:

- To better understand whether the observed bandwidth matches the reasonable expectation the observed metric should be compared to estimated predictions by some modeling tool. Dimemas can be used for this purpose. A dimemas trace of the identified regions where the networks seems to be a bottleneck should be generated. The best way is to first generate a Paraver trace of just the region of interest (Trace Generation menu). Then, the chopped trace should be loaded and a Dimemas trace generated for it (same menu). Finally, a Dimemas configuration file for the target machine be defined and the simulation carried out. By comparing the original trace (chop) and the Dimemas generated trace with the same Paraver cfgs described in this document it will be possible to understand properly the behavior of the application and target platform. It will be typically interesting to vary parameters of the simulated machine such as latency (overhead in the LogP model) bandwidth, contention, … and check which of those generate a better approximation of the original Paraver trace. The comparison can be based on the total duration, performed visually on the timelines or based on the comparison of histograms such as those described throughout sections 1.4 and 1.5

1.5.Collective Communication Performance

1.5.1.Which/how many communicators are used within collective MPI calls?

Some applications only use the COMM_WORLD communicator. Other applications do create and use new communicators. The question often arises for an unknown application whether it is using one or several communicators. In the last case, visualizing which communicators are being used by which group of processes and when is very useful to get a good feeling of the application structure.

Configuration file ***mpi/views/collectives/communicator.cfg*** should be used to address this issue.

What you will see: a timeline that shows the intervals when each process is within a collective MPI call. For each interval, the color represents the identifier of the communicator used by the call.

Related approaches:

- If you are interested in knowing which process is the **root of the collective operations** use configuration file ***mpi/views/collectives/communicator_root.cfg***. Only the root process will show the color corresponding to the communicator. Al others will be set to the default 0 value.

1.5.2.Are collective MPI calls performing efficiently?

Configuration file mpi/views/collectives/collective_bandwidth.cfg can be used to get a good feeling of this issue.

What you will see: the configuration file displays a timeline of the ratio between the size of the data involved in the collective for a process and the tame the process has been in the collective. It is a local measure and does not necessarily measure the actual communication bandwidth used by the collective implementation. It is nevertheless a good view to compare different instances of a given MPI collective call.

Directions for further investigation:

Some issues to investigate are:

- **Load imbalance or preemptions:** MPI being like a perfect gas, it will fill any hole left by real computation. Load imbalance or Preemptions are some of the possible causes of observing apparently inefficient collectives. They often result in collectives taking more time than expected and thus the collective bandwidth view will small values (light green) for some of the threads involved in a collective while the ones experienced higher load or preemptions will arrive late to the collective, will execute it fast thus resulting in a high valued (Dark blue) in this view.

- If you are very curious, you may be interested in analyzing **hardware counter information within the MPI calls themselves**. You should first load your hardware counter metric of choice (i.e. Loads). Then modify in the above 2D window the statistics selector to your desired choice (i.e. Average value) and the data window selector to the just loaded hardware counter view. Finally click on the Repeat button and you will get the corresponding statistic

for each MPI call (i.e. Average number of loads within each MPI call). **WARNING**: your trace must include hardware counter events on entry and exit of MPI calls (OMPItrace option -counters:mpi)

1.6. Hardware counter based metrics

Hardware counter events can be emitted to the trace on entry and exit of user functions and MPI calls (plus direct source code invocation of the trace_event API). From these hardware counter events, a bunch of direct and derived metrics can be computed. Given that the actual hardware counters captured by instrumentation packages are very platform specific, different sets of configuration files have to be provided for each platform. Even for one platform, many combinations of events can be instrumented. As starting point for novice users we provide a set of views that can serve as basic reference.

The views may be directly obtained from a single hardware counter or be derived metrics combining several of them. Each view represents a time varying function typically color encoded

For each platforms or events set, the view are classified in four major groups:

- **Program**: Metrics characteristic of the application itself. Usually absolute measures of events (i.e. instructions, FLOPS,....) not depending on the platform architectural parameters. Possibly ratios between several such absolute metrics.

- **Architecture**: metrics that depend on the application as well as on the specific architectural parameters (cache size, TLB size, replacement algorithms...) of the platform. These metrics do not directly measure performance although they may have a strong impact on it.

- **Performance**: actual performance metrics. Ratios where time (or cycles at least) are used as reference in the computation of the metric.

- **Models**: This section contains configuration files where a simple model of what the IPC (nstructions epr cycle) or elapsed time should be for each interval between to samples as a function of the acquired hardware counts. In general they could/should be compared to the measured IPC or elapsed time also available from the trace.

Specific Hardware Performance Counters: Intel Platforms

On many platforms, one is limited to a set of hardware counters being read at the same time, or the number of counters available. For this, one may have to do several mpitrace runs to get a better picture of the related hardware counters. E.g. the family of ia32 processors are quite diverse, with

respect to the number of hardware counters may be read out, to the overall width per counter changing from the standard Pentium (P5) to Pentium II/III (P6) to Pentium4/Xeon as is currently in use. Through PAPI, which is using perfctr to read out hardware performance counters on the Intel (ia32, ia32_64 and ia64), Athlon (K7 and Opteron) and PowerPC group of processors, mpitrace has a convenient way to access hardware performance counters.

The following counters may be selected through the MPTRACE_COUNTER-environment flag:

Specific Hardware Performance Counters: NEC SX Platforms:

On the NEC SX-8 Vector Systems, one has access to the following Platforms, selectable through the MPTRACE_COUNTER-evironment flag:

EX (execution counter) The execution counter (EX) is 52 bits long and is incremented by one every time a vector or scalar instruction is executed. When EX overflows, it is reset and starts counting from zero again.

VX (vector execution counter) The 48-bit vector execution counter (VX) is incremented by one every time a vector instruction is executed. When VX overflows, it is reset and starts counting from zero again.

VE (vector element counter) The vector element counter (VE) is 56 bits long and counts the vector