Performance Analysis of Mantid for ESS

Simon Heybrock

simon.heybrock@esss.se

August 17, 2015

Contents

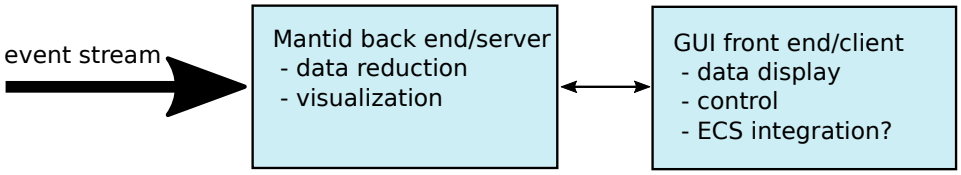


Figure 1: The big picture.

1 Introduction and overview

1.1 Big picture

The big picture is sketched in Fig. 1. This document is mainly concerned with the Mandid-based back end, in particular how to store the stream of events and run the instrument-specific data reduction on them, within a time frame that can reasonably be called "live".

When reading this document, keep in mind that it was created during my experimentation and process of understanding the current status and therefore may in some places go into more detail than required at the present time. It may therefore be a good idea to skip over some sections during a first read of these notes.

1.2 Optimization overview

Roughly speaking, there are two types of optimizations that we can do. The first, general performance optimizations comprises improved code structure, improved algorithms, reduced overhead, etc.. The second, parallelism, has several levels:

1. multi-node (MPI)
2. multi-core nodes
3. hyper-threading
4. cores with several execution ports
5. SIMD (vectorization)[1]

[1]SIMD ("Single instruction, multiple data") CPU instructions — such as SSE or AVX — apply the same instruction to multiple elements in a vector register at once (in parallel).

The compiler respectively the CPU itself usually deal very well with execution-port-based parallelism, so we ignore it in our discussion. Multiple cores and hyper-threading can in first approximation be considered as equivalent, so we will not consider them a separate issues for the time being. This leaves us with three levels of parallelism to consider, (1) MPI, (2) threading, and (3) SIMD. The boundary between MPI and threading is flexible — instead of threading within a node we can also use several MPI ranks per node. This will be discussed at an appropriate later time.

1.3 Optimizing data-reduction performance in Mantid

1.3.1 General remarks

1. Many algorithms used in data reduction treat all spectra independently and thus exploiting parallelism via MPI and threading is in principle trivial.[2]

2. Optimizations that do not rely on parallelism are certainly possible, but there are a few things to keep in mind:

   - Data reduction uses a whole set of algorithms, so a speedup would need to be obtained in all (or most) of them, which is time consuming.
   - The amount of speedup that can be obtained is always quite limited. In contrast, with parallelism, there may be no speedup in some cases where parallelism cannot be exploited, whereas in many other cases (such as in algorithms treating spectra independently) the speedup can be linear in the number of nodes.[3]
   - Computing power nowadays grows almost exclusively via increased parallelism, so we cannot rely on having "trivially" faster CPUs by the time we need it for ESS.

   Many algorithms do something comparatively cheap, and might thus be bound by memory bandwidth. Reduction work flows can (and many probably will) have a series a algorithms applied that all treat spectra independently. In that case we can circumvent the memory bandwidth limit by running a whole series of algorithms on a small set of spectra at a time (*cache blocking*).

In summary, we should probably resort to employing parallelism where we can, and focus other optimization efforts on parts that to not parallelize well.

1.3.2 Data flow

Our task can be summarized in two sentences: An incoming event stream is written into a workspace. The workspace is passed through a series of algorithms that in combination are called *data reduction*.

The streaming itself is considered in Sec. 2. We show there that a single TCP sockets seems to be capable to dealing with the data rate to be expected form ESS.

The next step is writing that stream into a workspace, which is not as trivial as it may appear to be. The crucial point here is that the stream of events is in general not ordered, so for now

[2] Since spectra are not equal, SIMD with data from *multiple* spectra is not trivial and in many cases probably impossible. As a consequence, in many cases the only option for using SIMD is within operations on a single spectrum.

[3] With threading in principle as well — unless the algorithm is bound by memory access, which is the case for most without further optimizations such as cache-blocking methods.

we have to consider it as completely random. A workspace keeps a vector of events for each spectrum, i.e., in some sense it is sorted by the spectrum number (or detector number) of the events. Writing the stream of events (which correspond to a random series of spectrum IDs) into a workspace causes writes to random memory locations. With the parameters required for ESS this seems to lead to very high cache-miss rates and a performance that does not seem to be sufficient. This problem and potential solutions are discussed in detail in Sec. 3.

Once data is a workspace the actual data reduction can start. Many algorithms can now employ parallelism.[4] There are differences in the reduction for every instrument, so a definite analysis and conclusion can only be given once we have gathered information for all of them. Individual algorithms are studied in Sec. 4 and reduction work flows for individual instruments in Sec. 5.

Aspects that affect the Mantid framework in general, i.e., things that affect most algorithms, are discussed in Sec. 7. Things that do not fit in any other section can be found in Sec. 8, and the current to-do list in Sec. 9.

2 Event streaming

- We need to stream event data from Tobias to Mantid
- At a rate of 10^8 events/s with 16 Byte per event, the data rate is around 1600 MB/s.
- We model a test in analogy to the existing streamers `FakeISISEventDAE`, `FakeISISHistoDAE` and the corresponding listener `ISISLiveEventDataListener`.

2.1 Stream without extra work

Goal: establish performance data for the blocking TCP sockets that are used by the streamer and listener.

- Implementation is based on the `Poco` library — this is what is used in Mantid.
- Various things slow down the tests and were usually removed for this micro benchmark:
  - writing to `stdout`
  - allocating and initializing data with random numbers before sending
  - allocating the output vector on the receiver side (`std::vector.resize()`)
- Figure 2 shows the dependence on the packet size for streaming locally (via localhost). An event would need, e.g., 16 Byte for time-of-flight (`double`) and detector ID (`int64_t`).
- Observations:
  - Top speed is around 3 GB/s for large packets (above 32 MB).
  - This top speed is reached only when we do not re-allocate the output vector every time (tuning may still fix that)

[4] They *can* in principle, but it is yet to be determined in each case whether or not they actually *do* in the current Mantid.

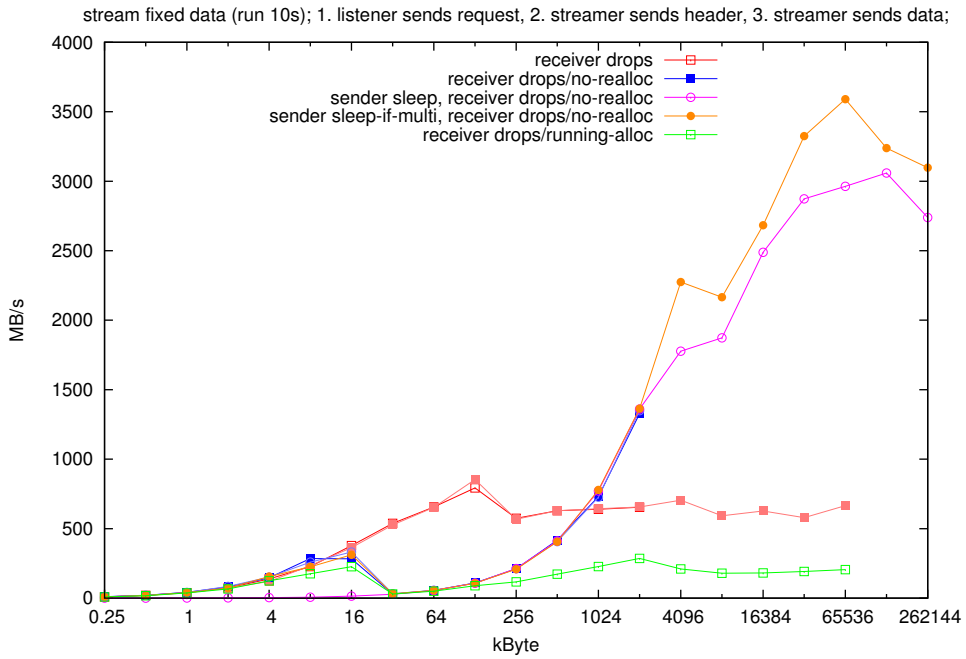


Figure 2: Streaming via localhost, data dropped by receiver.

  - There is a peculiar drop of the bandwidth around 64 kByte packet size in the cases without re-allocation. With re-allocation there is good performance in this region, but it flattens out afterwards. I do not understand what is happening, it might be related to the pauses caused be the re-allocation before reading from the socket...?

- Potential optimizations:
  - The handshake (listener sends request, streamer sends separate header) could be simplified to reduce the overall number of packets and potential latency impacts. This would most likely not improve the peak bandwidth, but we might be able to reach peak already for (slightly) smaller packet sizes.
  - multiple sockets
  - multiple threads in streamer and/or listener (in combination with multiple sockets!?)

To conclude, the socket performance does not to seem to be a big issue, under the assumption that we deal with large packets. Probably there is a whole series of optimizations that can improve the situation further.

2.2 Stream and append to workspace

Raw event data received via the stream must be inserted into a workspace by the receiver. Here we quickly demonstrate that the insertion into the workspace is a bottleneck in the current implementation. We then study this problem in isolation in Sec. 3. Figure 3 gives the resulting streaming bandwidth.[5]

- The dip around 64 kByte packet size is still there.

[5]This test seems to hang for event count 4194304 and above, find out why.

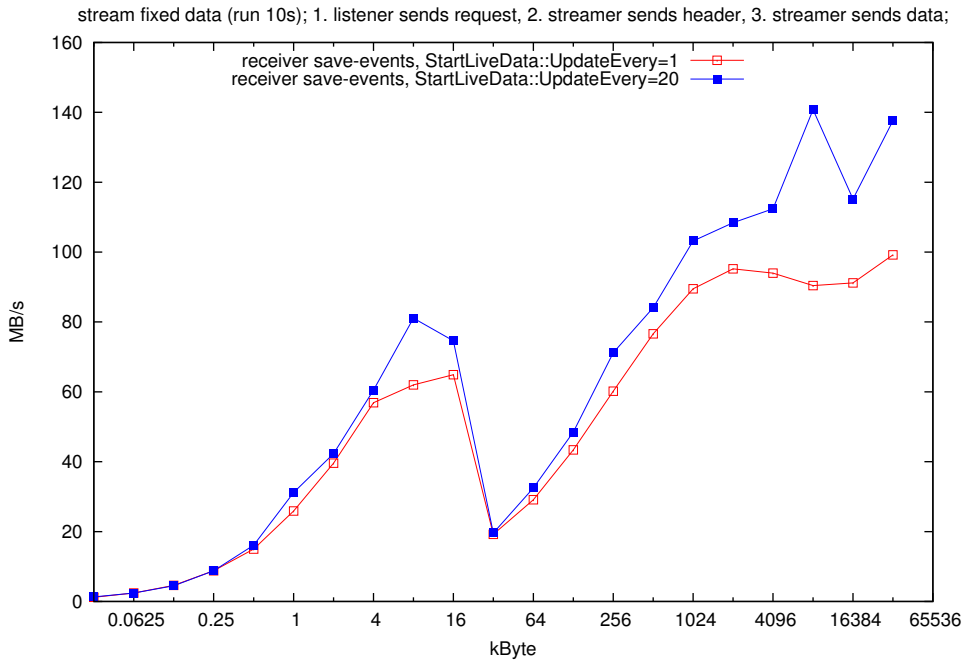


Figure 3: Streaming via localhost, data saved to workspace by receiver.

- Top speed is 100 MB/s for large packets.
- A quick profiling run with `perf record -g` indicates that a large fraction of time is spent in saving events into the workspace, and some reasons are obvious:
  - insertion corresponds to `std::vector::push_back`, i.e., may involve memory allocation and copy
  - unless the stream is sorted by detector ID (spectrum), the insertion randomly accesses spectra in the workspace, i.e., we will suffer from many cache misses.
  - this part is (currently) not threaded; note that we cannot thread over the events, since they need write access to the same spectrum, i.e., we would need to thread of the spectrum (detector ID), and this would be easier with a sorted event list
- Over the course of the run (10 s) `MonitorLiveData` repeatedly extracts the workspace from the listener. For `StartLiveData::UpdateEvery=1` this happens 10 times, and we see a slight performance decrease. No post-processing algorithm was specified.

In Fig. 4 we show a simple threaded version of `saveEvents()` where all threads loop over the event list and pick out a subset with `detectorID % threadNumber == threadID`. We observe a speedup, but it is much smaller than the increase in thread count.

3 Inserting stream data into a workspace

3.1 Current status

As we have seen in Sec. 2.2 the speed of adding events to a workspace is lower than what we get from the socket. It is also considerable below the speed needed for the expected event rate

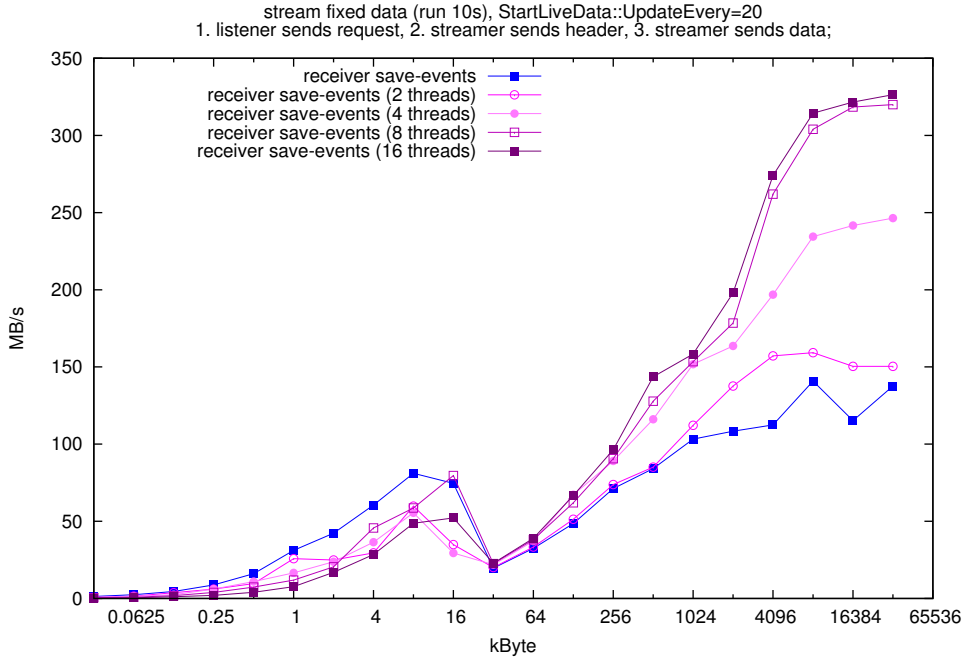


Figure 4: As Fig. 3, but with threading over detector/spectrum ID. There are 100 detectors/spectra. CPU has 12 cores / 24 hardware threads.

(under the assumption that all events are streamed to the same CPU/node). We thus study it in isolation, to understand the performance limits better. This test is the interface between the incoming event stream and a workspace. We ran a small number of tests based on the existing code for inserting events, using `EventWorkspace.getEventList(detector).addEventsQuickly()`. In particular, Fig. 5 we show how many events we can add to an `EventWorkspace` per second, depending on the pixel (detector/spectrum) count of the workspace. In Fig. 6 we show how many events we can add to an `EventWorkspace` per second, depending on the number of events to add. In Fig. 7 we show how many events we can add to an `EventWorkspace` per second, depending on the number of events already in the workspace.

Notes and observations:

- The detector IDs were drawn from a uniform random distribution. In a real test we will thus most likely see a different behavior, for example, fewer cache misses if many events are in a small set of detectors.
- Large pixel counts make things much worse (factor 5x to 10x):
  - If there are more pixels than events, basically every event will cause an (expensive) `alloc` (since in this test our input workspace was empty).
  - Due to random access, we should see a considerable drop once we exhaust the cache size. For sorted input we should be able to get rid of this limitation (in the limit where pixel-count $\ll$ event-count[6] — otherwise we have just few new events per spectrum, and thus cannot benefit from cache reuse).

[6]For real data the limit should not be written like that, the condition is: there are detectors with many new events, and ideally many detectors with no new events.

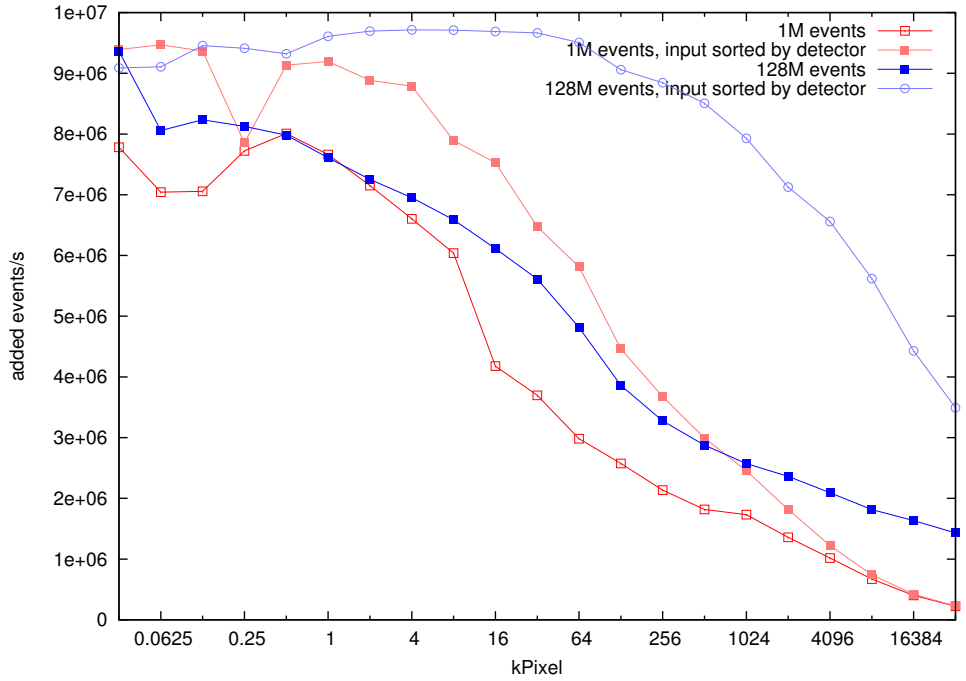


Figure 5: Pixel-count dependence of adding events to an event workspace. The (large) time for sorting is *not* included in the data.

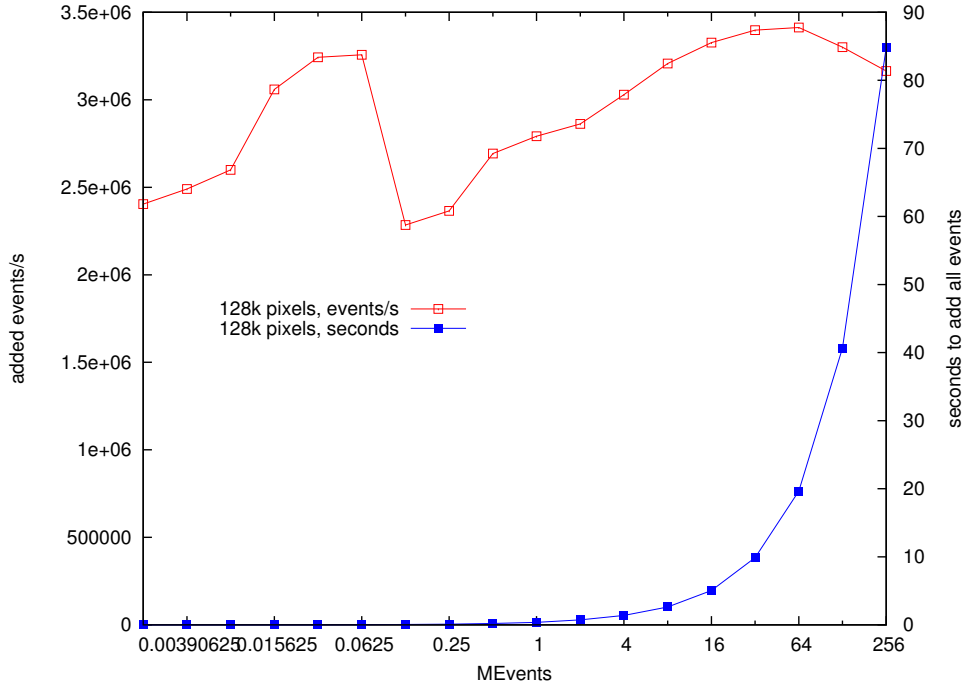


Figure 6: Event-count dependence of adding events to an event workspace.

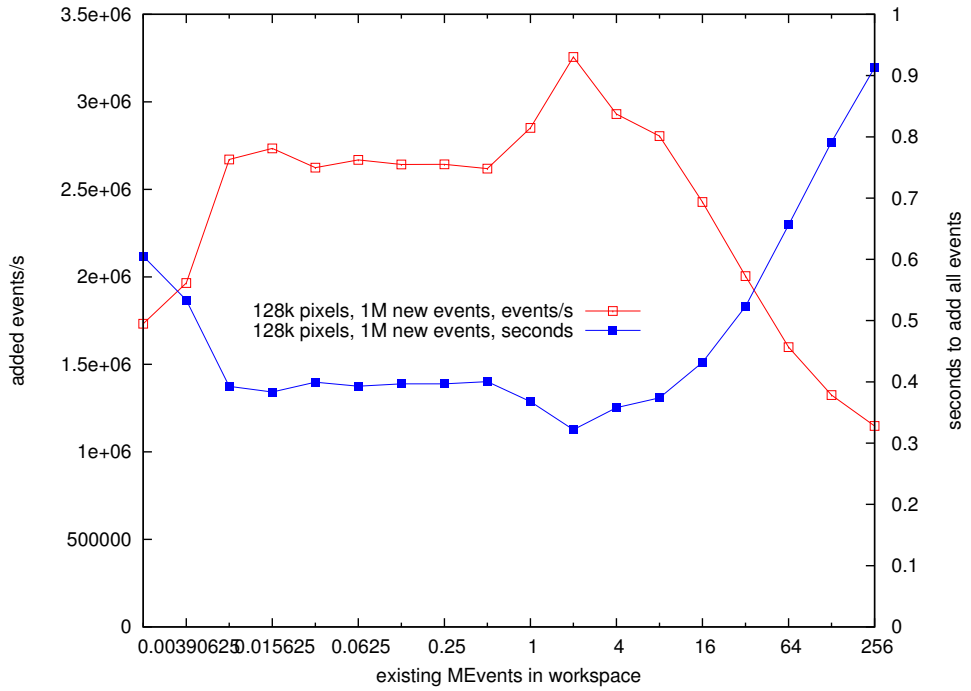


Figure 7: Dependence of the add-rate on the number of existing events in the workspace.

- large number of events to add does not make things much worse
- when more than around a few million events are already in a workspace, adding new ones takes considerably longer (more than 2x at 256 million events)

None of these results is surprising. To establish more detailed estimates, we study the problem outside the Mantid context in the next section.

3.2 Micro benchmarks

The spectra in a Mantid workspace are stored in a vector of vectors. The size of the outer vector is basically fixed and represents the detectors. Each element (corresponding to a detector) is a vector of events. New events will be appended to the event vectors. Here we study this problem in isolation: appending small amounts of data (such as the floating point value used to store the time-of-flight of an event) to a large number of vectors, where the vector to append to is chosen at random. The main performance limitation comes from the random access to main memory, which, in combination with the large amount of data, implies a huge number of cache misses.

The benchmark of the basic unoptimized case is given in Fig.8. We observe a rapid drop of the throughput (events/s) with the number of spectra. It levels out for 256 k spectra and above at just above $5 \cdot 10^6$ events/s.[7]

The first potential optimization is multi-threading, which we show in Fig. 9. Performance improves, but not proportional to the number of threads. We obtain a throughput of $2 \cdot 10^7$ to $3 \cdot 10^7$ events/s (for Haswell). It is noteworthy that the throughput is starting its decline

[7] For only 32 k events the result is better, but then there are fewer events than spectra, so this should probably be ignored.

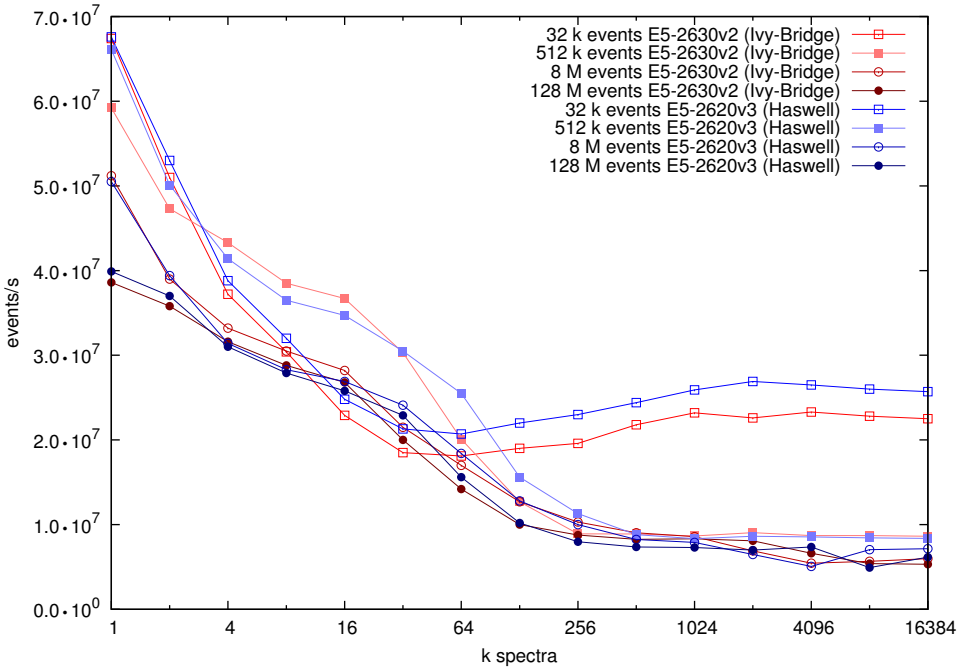


Figure 8: Default write of stream into spectra, using `std::vector::push_back`.

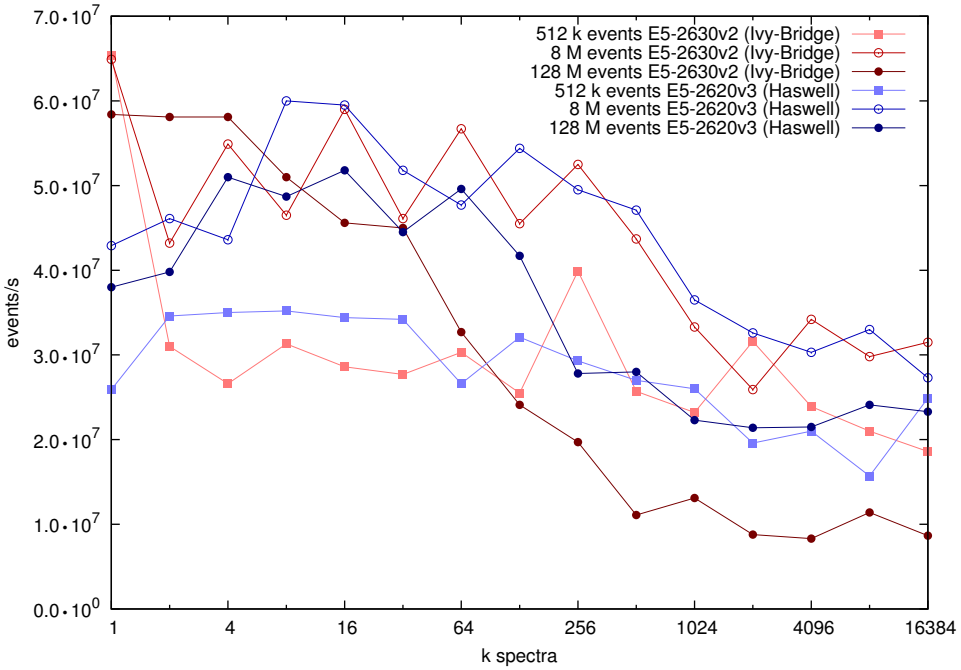


Figure 9: As Fig. 8 but with OpenMP threading (12 cores / 24 threads). All threads parse the complete stream, but write only for their subset of spectra.

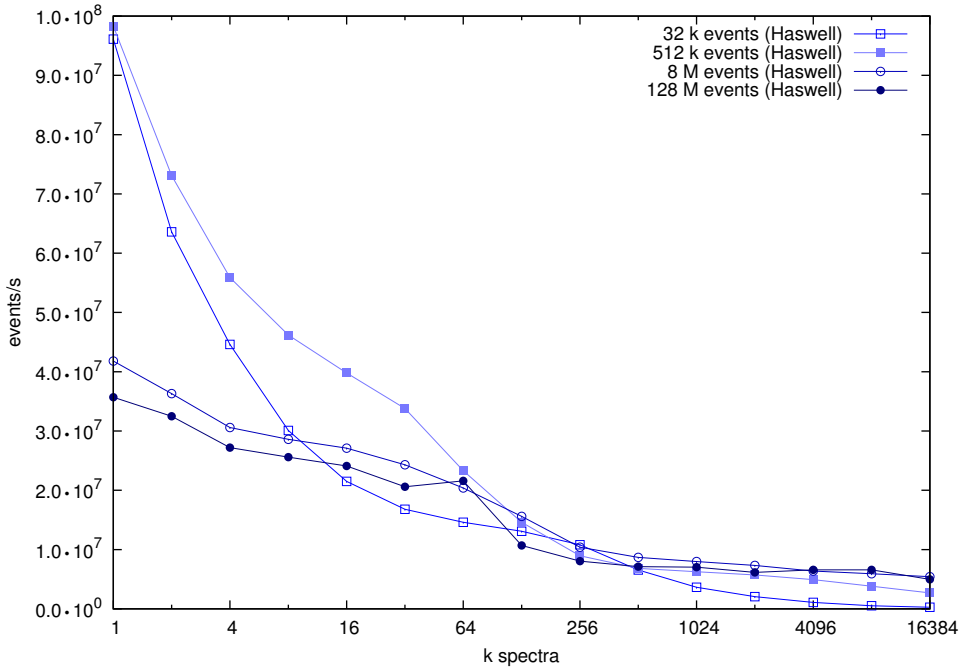


Figure 10: Writing stream into spectra, using a single resize after counting the occurrences of each spectrum in the stream. Data is then written using `operator[]`.

at a later point than the single-threaded case in Fig. 8. Furthermore there is a big difference between 8 M and 128 M events in the Ivy-Bridge case, which has vanished on Haswell. Both of these observations might be due to the TLB (which I think has been improved on Haswell).

Appending data to a vector will eventually cause a re-allocation and a copy of from the old to the new vector. To check whether or not this has a significant impact we test and alternative approach in Fig. 10: First we count the number of new events per spectrum, then resize the accordingly, and finally write the new data into the vector. As a consequence, there is at most one re-allocation/copy per spectrum. On the downside, we need to work through the full set of events twice. We do not observe a big difference to the implementation based on `push_back`. Our current conclusion/explanation is that the `std::vector` implementation is "good", in the sense that it over-allocates in a reasonable way and thus manages to keep the number of re-allocations due to a `push_back` small.

3.3 Considerations for an improved design

From the micro-benchmarks in the previous section we can draw some intermediate conclusions:

- For any number of spectra, the throughput is below the maximal expected event rate of 10^8 events/s.
- For a large (but realistic for ESS) number of spectra the performance degrades significantly.
- Unless we can improve the cache re-use, threading may not help much, because the memory system seems to be the limiting factor.

What can we do? Possible (single-node) optimizations of the existing code might include:

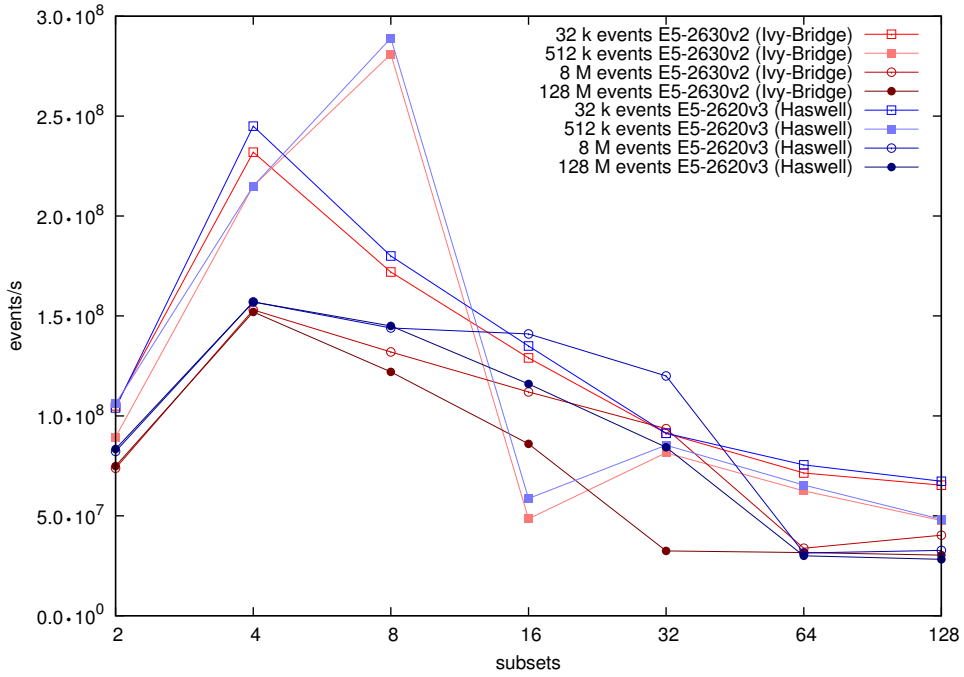


Figure 11: Splitting of an event stream into subsets based on spectrum, for purpose re-streaming to several nodes, each dealing with a specific subset. The number of pixels was set to 100k, but the actual number is almost irrelevant for this benchmark, since the subset is simply computed by a modulo operation.

- Better approach to threading. The current approach requires parsing of the full event stream by all threads and "expensive" modulo computation.
- Sort the event list. If the number of (new) events per spectrum is larger than 1 this will decrease the pressure on the memory system, due to caching effects. However, sorting is probably too expensive for this to pay off.
- Optimize and thread the approach taken for the test in Fig. 10.
- Use `emplace_back` instead of `push_back` — but most likely the compiler does that anyways in an efficient manner.
- Use vectors with a fixed minimum size to avoid many reallocations and fragmentation.
- To fix the cache-miss issue when writing to many locations, consider a small buffer that gets flushed when full. For many detectors the cache is probably too small — say 10^7 detectors, one event has 8 Byte, say we want to have buffer for 8 (one cache-line), gives 64 MByte. Furthermore this method might add many branches at a low level, which limit the performance.

Since none of the above appears to be obviously capable of providing a large speedup (please prove me wrong if you can), we will need to resort to a multi-node implementation. Here, the options are:

1. Each node gets assigned a subset of detectors. Multiplex the event stream, each node picks/stores those events that correspond to its subset of detectors.

2. Split event stream into frames, assign to nodes (round-robin), gather from all nodes (i.e., each node has data for all detectors).

3. Split stream into subsets (according to subsets of detectors) on a master node, re-stream to child nodes (no gather necessary).

Each of these multi-node options has some drawbacks and limitations:

1. Multiplexing the full stream puts more load on the network.[8] Picking out a potentially small fraction on each node will limit scaling, since in any case the full stream has to be read and parsed.

2. Splitting according to frames leads to a final gather operation. The extra load on the network is not huge, since it is basically just the same data again, in a different layout. However, the broadcast and in particular merging data from all receivers for each detector will have a significant cost. This merge would be on ordered data, but would still hit many memory locations. Is it faster than a merge from an unordered stream as we would get it in option three? This cost is still to be determined (TODO).

3. The whole thing hinges on whether or not splitting the stream into detector subsets can be performed fast enough, and whether the number of subsets can be made large enough such that insertion into workspaces is also fast. A test of the split into subsets is shown if Fig. 11. The obtained throughput for a small number of subsets (4 to 16) is at or above the ESS maximal event rate. However, since this benchmark shows only the first step of the process, the resulting latency might not be negligible.

A further option is a combination of the second and third option.

There is one aspect of instruments that will help us: According to Jon, detectors are arranged in banks, and readout is per-bank. If this is preserved in the event stream, we have a natural (coarse) pre-split that can be used to trivially distribute the whole computation. This will drastically improve the feasibility of several options discussed in this section. The factor that we can gain (i.e., the number of banks) depends on the instrument. See Sec. 5 and in particular Tab. 4.

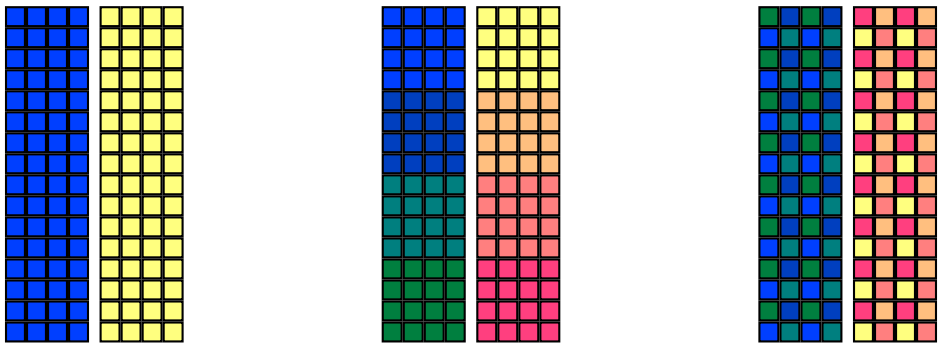
Furthermore, as also Mark pointed out, it may be acceptable to drop parts of the event stream during *live* reduction, since the purpose is only to see what is going on (is it?).

4 Mantid algorithms

4.1 Preliminary remark

Mark has (had) in mind to introduce parallelism based on events. This would rid us of the cost considered in Sec. 3 (but in turn increase the cost when building histograms). Apart from that I can see only disadvantages — reduced memory locality for all further operations on the events, each rank needs to hold data for *all* detectors, etc. This document thus considers only the case of parallelism based on pixels/detectors.

[8] Mark mentioned that Tobias plans to use IPv6 multicast for event streaming, would that also help here?



(a) Parallelism based on banks. **(b)** Parallelism based on blocks within banks. **(c)** Parallelism based on interleaved blocks within banks.

Figure 12: Several options for splitting up the detectors of an instrument for parallel data reduction. Colors denote different workers (threads, MPI ranks). Interleaving the pixels assigned to a specific works with those of other workers may be an option for improving the load balance.

4.2 Overview

The first key property that we have to understand for each algorithm that will be used is: How well can it be run in parallel, and what are the limits to the parallel efficiency. We can identify two classes of algorithms:

Algorithms that treat all pixels independently: The current impression is that this may be the majority. The property of interest for these algorithms is the amount of overhead or amount of work that is always done — the *sequential fraction*, which is independent of the number of spectra. This obviously limits the speedup we can get from parallelization.[9] We need to run tests with all algorithms in question and determine the sequential fraction. Results are gathered in Tab. 2. We can use this to determine whether or not the maximal parallel speedup is sufficient and then decide on improving the sequential fraction.

Algorithms with dependencies between pixels: There may be a wide variety of such dependencies, and a detailed algorithm-by-algorithm analysis is inevitable. An overview is given in Tab. 3.

The second key property is the amount of required memory bandwidth. This will influence two things:

1. If an algorithm relies heavily on memory access, parallelism *within a node* (i.e., threading, or multiple MPI ranks on a single node) will be inefficient and quickly reach a limit were more parallelism does not increase performance.

2. It may put a hard limit on the performance, unless we can improve cache reuse by pipelining several algorithms. This in turn is limited by the sequential fraction, as each algorithm will be called many times for small subsets of the detectors.

To get a realistic picture, we need a reasonable overview of which algorithms will be used for data reductions at ESS instruments. This information is being gathered Tab. 1 and the

[9] This is known as Amdahl's law, but this is really just the obvious behavior. Furthermore, there are usually many other effects that influence the performance depending on the parallelism and thus you hardly ever see the curve that Amdahl's law describes.

Algorithm	Bound by	Remarks
`SortEvents`	compute	See Fig. 15. Should be compute bound if spectrum fits in cache.
`Rebin`	compute?	Moderately expensive and frequently used. Performance issues unclear for now.
`ConvertUnits`		Moderately expensive and frequently used. Performance issues partially due to instrument/detector parameter implementation, see Sec. 7.
`RemoveBackground`		
`CorrectKiKf`		
`Divide`		
`ScaleX`		
`MaskDetectors`		
`GroupDetectors`		Performance issues mainly due to instrument/detector parameter implementation, see Sec. 7.
`NormaliseToMonitor`		
`ExtractMask`		Performance issues mainly due to instrument/detector parameter implementation, see Sec. 7.
`DetectorEfficiencyCor`	compute	Expensive iterative polynomial approximation. May be suitable for SIMD.

Table 1: Algorithms used during data reduction.

remainder of Sec. 4 studies individual algorithms in detail. We are still in the early phase of gathering this information and we do not have a reasonable estimate of which algorithms will be required. Therefore there are currently very few algorithms in this section and the algorithm-specific information is still very incomplete and imprecise. Likewise, the algorithms that are currently studied in this section are not necessarily the most important ones. The volume of this section is therefore currently not representative for its importance — it will eventually grow considerably in length. Note that we are currently omitting any algorithms that load files — under the assumption that we are running a live reduction, this should in principle be irrelevant for this performance analysis. Furthermore, this section is currently not covering any algorithms dealing with `MDWorkspace` — see Sec. 7.4 for a discussion.

An example of how a parallel reduction in Mantid might work, we show how an instrument might be split in Fig. 12 and a more complete picture in Fig. 13. Note that these pictures do not include important subtleties like monitors, which in many cases must be available on each worker in the Mantid back end — and must thus be duplicated somehow.[10] Furthermore, let us note that the division into banks depicted here is not meant to suggest a specialization of Mantid to such a case. Parallelism should be understood more general: It deals with subsets of detectors. The "specialization" of the subsets to banks is merely important for the stream-receiving parts that deal with very high event rates — to ameliorate issues discussed in Sec. 2. Other parts of Mantid, in particular the large majority of algorithms, would not necessarily be aware of this.

4.3 `LoadLiveData` interaction with workspaces

What does `LoadLiveData` do when it has loaded new data? There are at least two options:

[10] Also keep in mind that this might be detectors with very high count rates — how does this affect our performance analysis of the event handling?

Algorithm	Overhead	... negligible for	Bound by	Remarks
`SortEvents`	0.002 s	> 10k events/thread	compute	See Fig. 15. Should be compute bound if spectrum fits in cache.

Table 2: Performance parameters of algorithms used during data reduction.

Algorithm	parallel operation	description
`Q1DWeighted`	global sum	Compute 0-dimensional $I(Q)$, requires integration along angle. Used by `EQSANSAzimuthalAverage1D`.
voxel detector trajectory	?	find matching Bragg peaks of different frequencies that will be spread out across detectors due to nature of voxel detectors, remove them, etc.

Table 3: Overview of algorithms that are non-local, i.e., work on more than one pixel/spectrum/detector at a time and thus require communication.

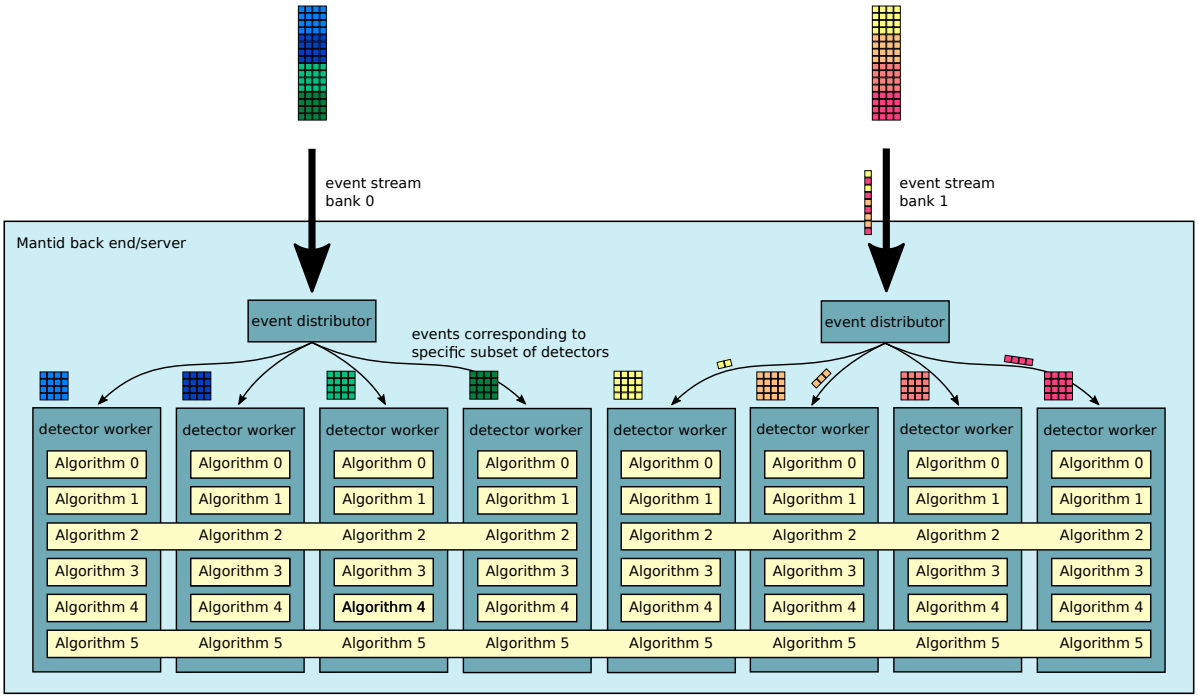


Figure 13: Overview of parallelism model in the Mantid back end. Events from different banks arrive in separate stream or packets. An event distributor splits this stream according to the number of workers (threads and/or MPI ranks) per bank. This may involve re-streaming of the data. In the workers the event data is passed through a series of algorithms. Many algorithms (0, 1, 3, and 4 in this example) can run independently on each worker, i.e., for each detector block. Some algorithms may involve data from a full bank, and thus interaction and communication between the corresponding workers (algorithm 2) or even the complete instrument (algorithm 5).

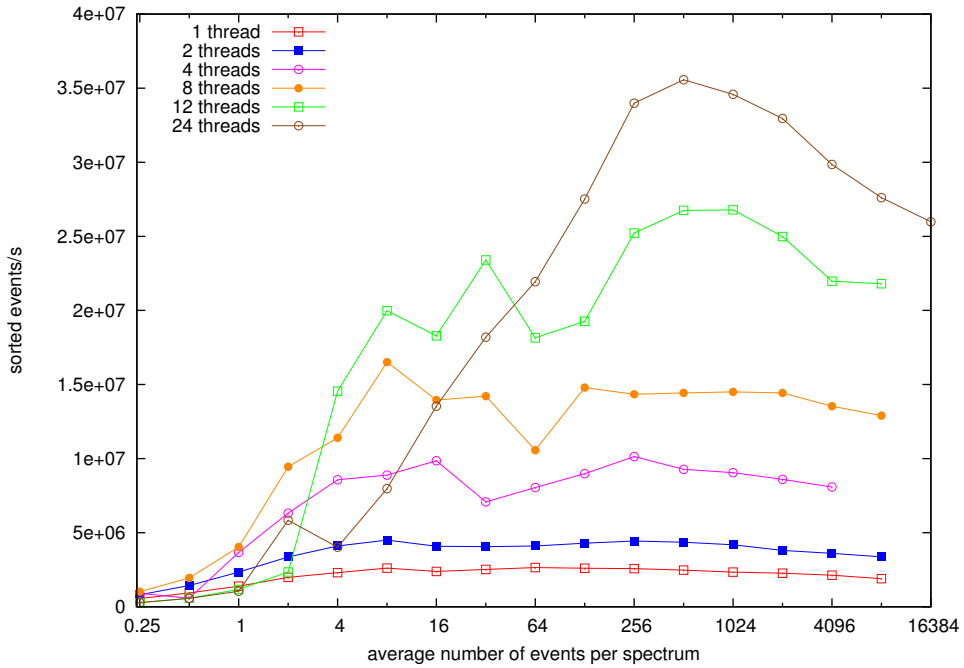


Figure 14: Sorting speed depending on the number of events per spectrum and the number of threads.

- `AccumulationMethod=Replace` assigns data to a (temporary) workspace and then calls the `SortEvents` algorithm
- `AccumulationMethod=Add` calls the `PlusMD` algorithm

Others (like `Append`) can be specified, but would that be what we need?

4.3.1 `SortEvents`

Called with `SortBy = "X Value"`, which causes sorting by TOF. The algorithm simply calls `EventWorkspace::sortAll()`.[11] See Fig. 14 for the performance of the existing sort method and Fig. 15 for the performance of the `SortEvents` algorithm, which calls the same sort method.

Observations and questions:

- Cannot quite see the $n \log n$ behavior, maybe that would hit us only for larger spectra?
- What is giving the bound? Instruction? Memory?
- There is a mutex/lock in `EventList::sortTof()`, does it cause trouble?
- Is there overhead from the thread pool etc.?

Possible optimizations:

- When adding new events to a sorted spectrum: Sort new events, merge sort.
- Use sort algorithm with SIMD support (quite a bit of work, but there are some papers that show how to do it).

[11] Note that this uses all available hardware threads per default, even if the environment variable `OMP_NUM_THREADS` is set to something lower. For the tests here the code was modified to be able to influence the number of threads in use.

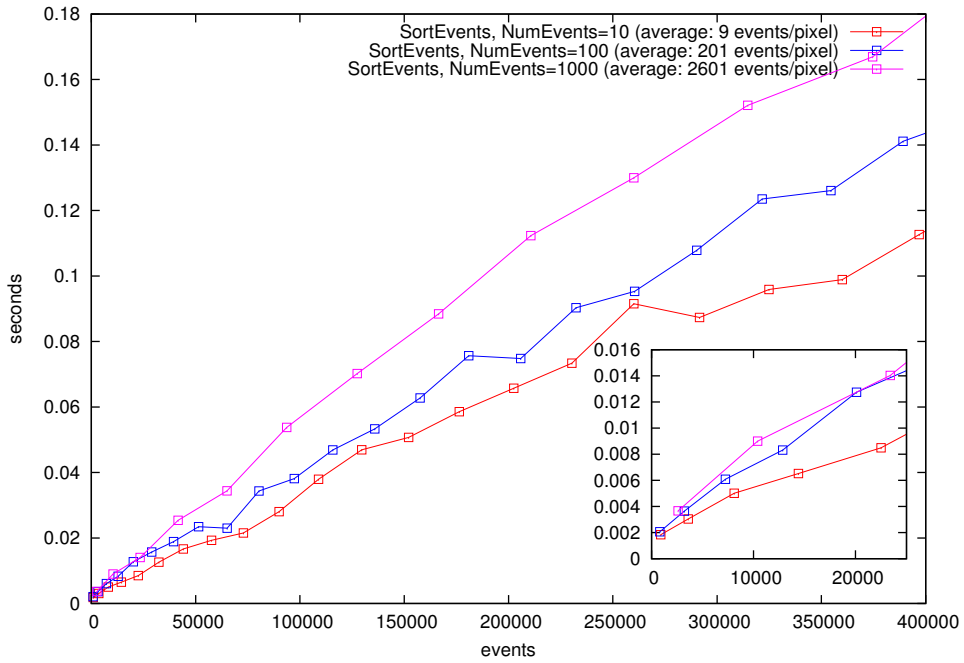


Figure 15: Sorting time with a single thread, depending on the number of pixels and events. The algorithm was called via its Python interface.

4.4 Algorithm status

This is a random collection of notes and TODOs for existing algorithms that I encountered during various tests.

- `SortEvents` uses a two-level threading, usually based on spectra, but within spectrum if there are few spectra. This may interfere with other threading models or multiple MPI ranks on a node. TODO: check if this makes problems, add control functionality.

5 Data reduction

5.1 Overview

Ultimately we have to make sure that data reduction is sufficiently fast for every single instrument. After the first few instruments things probably get easier, since I expect that most reductions use a similar set of algorithms. The purpose of this section is to get an overview of how reductions work, what algorithms they are using, and what the specific performance issues are. Regarding performance issues there are two aspects: (1) each individual algorithm, as studied in Sec. 4 and (2) aspects of the specific combination of algorithms and how they are used in a particular reduction work flow.

The latter aspect may turn out to be very important for optimization: Knowledge of how the algorithms are used may allow for removing redundancies, such as an input parameter that is known to change rarely, an can thus allow for the use of pre-computed results and lookup tables. Furthermore, given sufficiently simple algorithms, a whole series of algorithms may be merged into one transformation, such as two consecutive scale operations that can be rewritten as one. These optimizations might often require knowledge of the instrument and the physics.

Instrument	Detectors	Banks	Max. detectors per bank
DUMMY	10.000.000	10	1.000.000

Table 4: Overview of instrument parameters.

An overview of instrument parameters is given in Tab. 4. The purpose is to give a feeling of the expected event rates and pixel counts, which in turn is crucial for interpreting performance considerations in the rest of this document.

Furthermore, we must determine the memory consumption of each reduction work flow. This includes measuring the actual memory consumption of the current implementation, as well as estimating the theoretical minimum. For the users it is desired to keep events in memory for as long as possible. To achieve this (and/or to reduce hardware costs) we must then decide whether or not it is necessary and feasible to reduce the memory consumption. This is a separate issue for each instrument. Probably this also involves understanding how the instrument users will work and use the reduction. For example they may do several similar but different reductions on the same data set, keep old results open, etc., which in all cases will increase the memory consumption. One option might be to forbid this and just provide a single view into the live data, i.e., one reduction that is running on the Mantid back end that can be configured, but not the completely generic work flow from the Mantid GUI.

- Is it reasonable to start benchmarking at this level?
- Do we have realistic input data? Get some from instruments at other facilities.
- Tests of scaling with number of spectra might be a good test — it can tell us whether or not a simple multi-node parallelization can get us anywhere, without further work. That is, we want to be able to run a reduction for just a part of the instrument. Is that possible currently, will that break something, or not give a proportional amount of computation?
- Find a way to disable threading everywhere in Mantid, it just messes up these tests. Threading would be the second step.

Most of the remainder of Sec. 5 studies individual data reductions in detail. Note that there are currently very few data reductions in this section, because we do not have any detailed enough information on that. The volume of this section is therefore currently not representative for its importance — it will eventually grow considerably in length. Likewise, the data reductions that are currently studied in this section are not necessarily the most important ones.

5.2 Retaining events

For how long do we have to or do we want to preserve individual events? For live data reduction, the primary purpose seems to be visualization, so the main purpose may be plotting some histograms. As a consequence, a transition from event-based to histogram-based computations is necessary at some point in the data reduction. This may be the last or one of the last steps. So why do we keep the events? Reasons may be:

- Changing the bin size without loss of precision. To make use of this in live reduction we must however be able to re-process all buffered events within a reasonable time. Say we change the bin size, the re-binning of *all* events is triggered. As a consequence this is required to be significantly faster that what is required for the live event stream, otherwise the re-binning will not catch up with the live stream within reasonable time.
  - If re-computation is only required for one of the last parts of the reduction chain we could keep the event data from an intermediate point and reduce the cost of re-computation — at the expense of larger memory requirements. This might lead to a mess and lots of manual intervention for each reduction script, so for now it is unclear whether this is feasible or not.
  - The simplest option is to keep the original events but do the reduction in histogram mode.
  - If we want to run reductions in event mode, we must determine for each instrument whether all algorithms used by the respective reduction script support event mode and preserve events.

Effects of keeping all events:

- Running algorithms on the workspace holding the events will become more and more expensive, unless...
- ...algorithms are aware of newly added events and can restrict themselves to processing those.
- Alternatively (or in combination), we can have workspaces that hold only the new events, run reduction of them, and merge only in the end. This may however not make sense for all algorithms (say for example for a normalization).

Can we give a reasonable estimate on how long we might be able to keep data? The biggest unknown is the amount of actual memory consumption by Mantid when running a reduction, for a given number of events. How many copies and other overheads will there be? A very rough estimation:

- What do we need to store? Just time of flight? Then 8 Byte per event. How big is the meta data?
- Assume 8 Byte per event, and a factor 2× overhead (meta data, over-allocation by `std::vector`), so 16 Byte per event.
- Worst case: 10^8 events/s, i.e., 1.6 GByte/s, so this might give us $\mathcal{O}(10)$ seconds on a single node,[12] $\mathcal{O}(1)$ minute on 10 nodes, and $\mathcal{O}(1)$ hour on >100 nodes. More realistically (at least short-term), if we assume a 10 times lower rate, we can accumulate for an hour if we have more than 10 nodes.

What if this is not enough? Apart from hardware solutions that give us more storage (such as an SSD buffer or more nodes with more memory), what can we do in software?

[12] 16 GByte of "available" memory may or may not be too optimistic, given that we will typically need several copies.

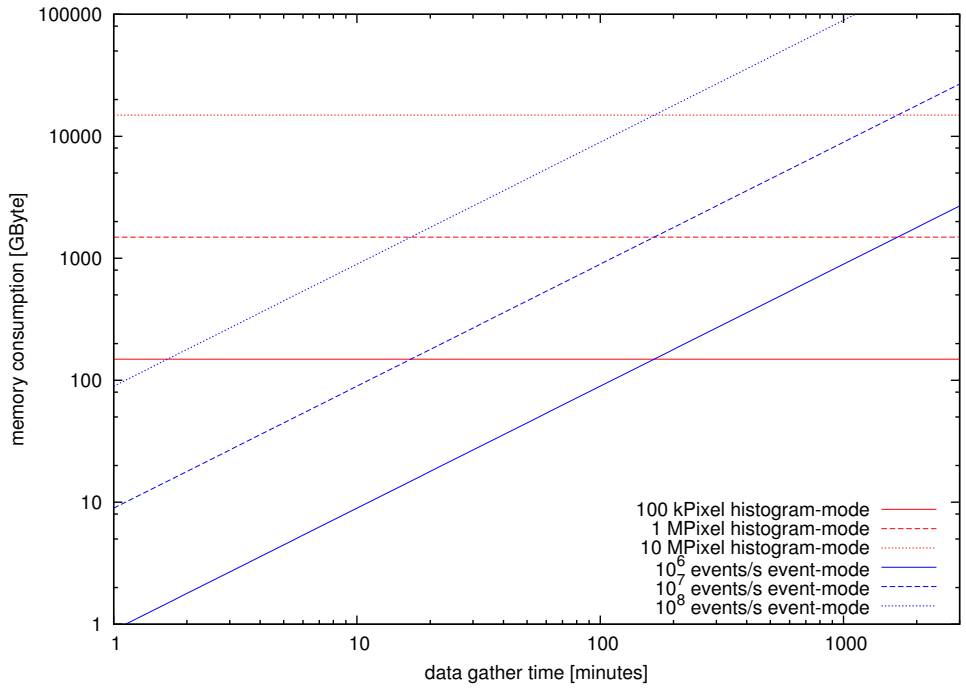


Figure 16: Comparison of memory consumption in histogram and event mode, depending on the time data is gathered. Histogram mode assumes a bin width of 1 μs and a range of 100 ms, i.e., 100000 bins. The memory consumption is computed based on the assumption that storing an event takes 16 Byte and storing data for one bin takes 16 Byte.

1. Drop old data and have something like a running window that keeps only events from the last X seconds.

2. Keep old data, but keep only every nth event (or chunk of events) over the whole history, i.e., thin out the data.

3. Switch to using histogram mode.

Each of this methods has some limitations:

1. Dropping old data means that we either lose statistics (if the parameters of the run were constant over the whole run), or may lose, e.g., parts of a temperature sweep.

2. Thinning out data means that we lose statistics.

3. In histogram mode certain modifications of the reduction algorithm may become impossible, and would require to start gathering data from scratch. For example we cannot make the bin size smaller than what is defined by our existing histogram. Furthermore, a histogram uses the same amount of memory for each pixel, the memory consumption is given by the number of bins multiplied by the number of pixels.

Regardless of the exact implementation, any reduction run that keeps event data must somehow be able to keep track of the current memory consumption. We should not just keep storing more events until we are terminated by the OOM killer.

Could we in principle afford to keep data in histogram mode, with bins small enough for all practical purposes? The lower practical limit for the bin size is the actual TOF resolution of the instrument. According to Jon this may be around 0.1 to 1 μs. We show an estimate of the memory consumption in Fig. 16, where we compare histogram and event mode. For this "fine-grained" histogram mode we will need at least in the range of a TByte of main memory, even for an instrument with only 100 kPixel. In comparison with the memory consumption of the event mode this is not so shocking, however. The memory consumption of the event mode surpasses that of histogram mode within minutes, hours, or latest after about a day — depending on the pixel count and event rate. For instruments with very high event rates but low pixel counts histogram mode might pay off basically immediately.

5.3 Instrument requirements

- Instruments should probably be obliged to send events "per-bank", i.e., should not mix events from different banks in the same frame. This would make splitting the event stream for parallelizations much easier or even feasible.
- Assuming that the Mantid instrument code is redesigned, there is a chance that internally detectors will be addressed by linear and contiguous index instead of a detector ID. Thus, we must either do an expensive translation when writing the event stream data into a workspace, or require from the instruments that they label detectors with a linear and contiguous index.

5.4 `SANSReduction`

We are considering this algorithm since it might be very similar to what we need for LoKI. See `http://docs.mantidproject.org/nightly/algorithms/SANSReduction-v1.html` for the official documentation.

Without having looked too closely, the following questions arise:

- Given fixed parameters, some part seem to do always the same and could even be combined, e.g., `SANSAbsoluteScale` and `NormaliseByThickness`, which both do a scale.
  - When would the scale factors change?
  - Can we just pre-compute and combine them into one?
- Identify parts of the algorithm that are independent of variable parameters, i.e., depend only one the events for a given detector and parameter values known at the beginning of a run.
  - These parts, i.e., a series of sub-algorithms, may be combined into one simpler conversion (think JIT?)
  - If some parameters *do* change infrequently, can we reenter the algorithm at an appropriate point, without redoing the full reduction? To do this, we would need to keep/store events at every possible reentry point!
- To what extent can we just pass through the algorithm without actual event data, and determine a function that corrects input event data?
  - Binning may interfere, but can this be the last step?

Algorithm	
EQSANSDarkCurrentSubtraction	Loads dark current workspace if it does not exist. What happens if called repeatedly, does it get reused?
→ RebinToWorkspace	Such that bins of dark current match data. Will this be done again if it still matches? Does this imply that we cannot work in event mode from here onwards?
→ Scale	
→ Minus	
EQSANSNormalise	Sub-algorithms apply when normalizing to monitor, other options are available.
→ EQSANSMonitorTOF	works with MatrixWorkspace
→ ConvertUnits	
→ SANSBeamFluxCorrection	works with MatrixWorkspace
→ ConvertToHistogram	
→ RebinToWorkspace	
→ Divide	
SANSMask	
→ MaskDetectors	
SANSSolidAngleCorrection	EventWorkspace supported explicitly. Trigonometric functions for each detector, may avoid re-computation, just keep the resulting factor, then scale spectrum or event list.
SANSSensitivityCorrection	looks messy, but maybe it isn't?
EQSANSDirectBeamTransmission	
→ CalculateTransmission	
→ RebinToWorkspace	
→ ApplyTransmissionCorrection	
Minus	
SANSAbsoluteScale	other cases with other sub-algorithms exist
→ Scale	
NormaliseByThickness	
→ Scale	
EQSANSAzimuthalAverage1D	
→ SANSAzimuthalAverage1D	
→ ConvertToMatrixWorkspace	convert from EventWorkspace
→ Q1DWeighted	non-local!
→ ReplaceSpecialValues	
→ ReactorSANSResolution	
→ ReplaceSpecialValues	
→ EQSANSResolution	
→ TOFSANSResolution	
EQSANSQ2D	
→ Qxy	
→ ReplaceSpecialValues	

Table 5: Overview of algorithms called by SANSReduction. We are not listing algorithms that are probably called only infrequently, i.e., not with every new frame of events from the live stream.

6 Visualization

After the data reduction the resulting data needs to be plotted or visualized. This should pose no problems for simple cases, like plotting $I(Q)$ after an integration. Often, however, the data to plot may be larger and more expensive to process. It is not clear whether or not it is feasible to run this on a single node (such as the Mantid front end, that gets the reduced data from the back end). This may depend on both, computing cost and memory consumption. What are the scenarios?

- "Show instrument": Requires event counts from all detectors.
- "Slice viewer": Requires all data from all detectors.
- . . . ?

7 Framework

7.1 Overview

There may be performance issues that are not specific to a certain algorithm or reduction work flow, but instead are a generic problem of the Mantid framework.

The following potential issues have been observed so far:

- `getDetector()` is very expensive. It always makes a copy of the instrument, but in 99% of the cases we do not need that — we just want to query the detector for something, e.g., whether it is masked.
- `getPos()` (and other access routines for component parameters) called on some detector components is very expensive. It seems to search some map based in a string identifier.
- `shared_ptr` everywhere. The ensuing atomic instruction may influence especially the multi threaded performance.

The first two are related to the Mantid instrument code and are discussed in Sec. 7.2. The influence of `shared_ptr` is discussed in Sec. 7.3.

7.2 Mantid instruments

Running some of the "reduction" system tests in a profiler (`perf`), I observe between 5% and more than 30% of time spent in code related to accessing detector/instrument parameters. Two options for solving the `getDetector()` and `getPos()` issues are:

1. During live reduction, we will repeatedly call these functions when new events arrive. We could thus buffer the results and request and update only when things move. The disadvantage is that we would need to change every algorithm that should make use of the buffered values.
2. Change the internal design. This seems like the cleaner solution, but it requires a complete understanding of all internals.

The first suggestion may appear to be a viable solution at first since it is less invasive and may be restricted to algorithms where it is important. However, this will limit the performance gain we can obtain, and it will keep haunting us each time a new algorithm needs optimization, making the code less readable. After a more detailed look at the internal design of the instrument and parameter implementation, we have identified the following potential "quirks" in the `Instrument` design (these range from things I do not understand properly or potentially inefficient design choices to flaws or bugs):

- Detectors are identified via `detid_t`, which seems to be arbitrary (to some extent). As a consequence, when accessing a specific `Detector` of an `Instrument`, an expensive lookup in a `std::map` is done.
- Calling `MatrixWorkspace::getDetector()` will create a copy of the underlying `Instrument` (this does not copy the base instrument or `ParameterMap` but is quite expensive nevertheless). The only purpose of this is to get a *parametrized* `Instrument`, which is used to hand out a *parametrized* `Detector`. This is actually easy to fix, but shows how hard to understand the current design is.
- Calling `Instrument::getDetector()` will — apart from the search in a `std::map` as mentioned above — create a copy of the `Detector`. The purpose of this is to get a *parametrized* `Detector`. Since we actually just want a `const` reference to a `Detector`, since seems wasteful.
- Accessing parameters of a `Detector`, such as the position, looks it up in the `ParameterMap`. The map is for the complete instrument, not for each single detectors, so it is potentially big. It uses the pointer to the detector as a key for an (expensive) lookup. Once the parameters for a specific detector ID are found, the search continues, based on `std::string` comparisons to find the actual parameter.
- To speed up read access, caches have been added. However, accessing the caches again involves a search in a map, which is expensive. Basically the caches are an inefficient attempt to workaround the problem that the instrument design is optimized for writes instead of reads — re-computation of parameters is done when reading instead of when writing. This is inefficient and wasteful, since in a typical data reduction, read access should be much more frequent than write access.
- Why the "base instrument" vs. "parametrized instrument" structure? The reason that Martyn Gigg gives is memory consumption. There are two ways in which the current design attempts to save memory:
  1. There is a single base instrument which may be "modified" via a `ParameterMap`. This map stores only modified values, and is thus small (such as when a detector bank is rotated — only the component corresponding to the bank needs to be modified, all children will implicitly get rotated by means of the tree structure of the instrument). However, this is partially deteriorated by the caching mechanism. As soon as the map is accessed it will start caching the parameters (e.g., positions) of components and thus the memory consumption will grow. As far as I can tell only the positions of parents are cached, so if the instrument is setup with pixels as leaves that all share a few parents the extra memory consumption may not be huge.

2. The `ParameterMap` is shared among workspaces as long as it is not written to, so copying a workspace will not copy the instrument. However, this would also be a non-issue when storing complete instruments: There is no need to have an instrument *for each workspace* — instruments and workspaces could exist independently.

- Is it necessary to have critical regions in `ParameterMap` read and write access *in addition* to the mutex/lock in `Kernel::Cache`?

- Why are there critical regions in the *read* access to `ParameterMap` in the first place? Reading while writing will anyways lead to random outcomes (race conditions), so must be forbidden (consider rotating a detector bank while an algorithm is working through a list of spectra). Similarely, `ParameterMap::contains()` does not have a critical section, but `get()` does — but it is useless anyways, because the map may have been modified between calling `contains()` and `get()`. Is there any realistic scenario where we modify the map from multiple threads?

- `ParameterMap` is designed to be copy-on-write, however the way it is implemented the copy will be a *shallow* copy and the actual underlying `Parameter` entries are not copied (I created issue #13258 in the bug tracker regarding this problem).

What could a new design do differently?

- Do not use `detid_t` but a linear index for access. This removes the need for maps, and we can just return a vector element. Note that this linear index cannot be the actual detector ID, since the latter in non-contiguous. Rather, we may need to introduce a new internal index for detectors. As a consequence, there will be a translation between the detector ID and the internal index at certain points (such as when receiving the stream, or when the user interface requests information about a specific detector). The crucial point is that this translation would happen rarely compared to the current design, where a kind of translation (from the detector ID to a pointer, via lookup in a map) happens for *every* access. In other words: Most algorithms do not care about the actual value of the detector ID, they just need to access all detectors — the underlying data structure should reflect this.

- Workspaces and instruments should exist independently:
  - There may be many workspaces referring to the same instrument.
  - There will probably not be more than a few different parametrizations of the same instrument.
  - Workspaces can keep shared pointers to instruments, instruments may implement a copy-on-write behavior when parameters are modified. So we can just create a new copy of an instrument when a parameter is changed (and the reference count to that instrument is larger than 1).

  As a consequence, we could just directly store the actual parameters, without distinguishing between the base one and the modified one. However, it may be more efficient to have a base instrument for parameters that are not modified. Basically what is now happening on a per-detector basis (picking either a modified parameter or the base parameter) might be implemented on a per-parameter basis. That is, if one detector is moved, probably many are, so we could just have one big vector of new positions that we

directly access with an index, instead of the current lookup in maps and re-computation of position. Note that this would still consume more memory than the current implementation, where the parameter map will contain only the position change of a parent component, whereas the new positions of all leaves would be stored according to the new suggestion.

- Keeping a "base" instrument could be an advantage for read-only parameters, such as component names. This would reduce the memory consumption.
- Likewise, keeping a "base" detector could be an advantage for parameters like the shape of a pixel, to avoid storing it redundantly.
- What about the tree structure? We will want to keep the principle, but the implementation might be more efficient when inverting the logics: Right now, if a parent component is moved, nothing happens to the children. Instead the component tree is walked every time a child parameter is accessed. However, probably accessing parameters is more frequent than modifying parameters. Thus, when modifying a parameter this should be propagated down through the component tree. So instead of the current caching of positions, which makes the design messy, we just update all values right away.
- Keep in mind avoiding pointer chasing... workspace down to detector
- Improve locality when accessing detector parameters? Put everything in one vector instead of a separate allocation for each detector? Consider using SOA instead of AOS (for SIMD). Implications:
  - Speeds up modifying an instrument (such as moving a bank).
  - It is unclear if we gain anything when working on events or spectra. Right now I think it is quite unlikely that we would use SIMD based on multiple detectors, so we would gain nothing there, or actually reduce the locality.

  How are detector parameters typically used? "All/many parameters of one detector, then next detector" or "one parameter of all detectors"?
- How much memory would an Instrument use? Say each detector has position, orientation, mask, monitor? 6 `double`, 2 bool, so in the order of 64 Byte. With 10 million pixels this gives 640 MByte, which is a lot, but it would probably rarely be all on a single node. In any case, unless we have many different parametrizations of an instrument at the same time, this sounds manageable.
- What are the subtleties that were ignored in the discussion so far?
  - grouping of detectors (i.e., one spectrum may refer to several detectors)

The redesign discussed here would/should probably go hand-in-hand with a masking redesign, see issues #9757 and #10116.

7.3 `shared_ptr`

So far I have had limited success trying to quantify the impact of the ubiquitous use of `shared_ptr`. In a CPU counter based profiler (`perf`), I see a considerable amount of samples involving `shared_ptr`, however many of those actually include costs of constructors and

destructors. Overhead from atomic operations is definitely visible, but in the cases I have seen so far it is not a huge effect. Since this is most important in parts that are threaded, it is however unclear if such a simple test and interpretation is meaningful. This might also depend a lot on the architecture: With a shared last-level cache the atomics might be quite cheap, so there is little impact.

My current conclusion is that this is not the most pressing matter.[13] A redesign of the instrument code would probably anyways eliminate many or most of the `shared_ptr` occurrences in lower-level loops, and the others will mostly not be significant. So with some luck this might turn out to be a non-issue.

7.4 `MDWorkspace`

- Basically this does not distinguish between pixel ID and TOF (or the resulting physical coordinates) — pixel positions and TOF are just coordinates of points in space.
- Parallelization along dimensions representing pixel IDs should be simple. But what if we don't have that? We can always split up some of the dimensions, but what happens when we transform, say from pixel positions and TOF to Q_x, Q_y, and Q_z?
  - Redistribute — probably difficult and messy to implement.
  - Just keep things on the node where they were before transformation — as a consequence, the same coordinates might be present on different nodes, i.e., the workspace subsets overlap. Furthermore, the subsets would most certainly not be cubic anymore — or rather, the design of `MDWorkspace` probably makes them cubic, due to the recursive subdivision, but many boxes would be empty.

  The latter option might be fine in principle.[14] However, if events for the same box are distributed to several nodes, subdivision threshold will not work as expected, and the result we see in the end will slightly depend on the number of nodes used for computation.
- `MDEvent` uses considerably more memory than a "normal" event since each event stores also its position in addition to the value and error.
- In memory, an `MDWorkspace` consists of many small vectors. If TOF (or some related unit) is one of the dimensions, we basically have a binning in TOF, even in event mode, and each "bin" (called box in this context) will be represented by a vector. As a consequence, we may get many vectors in comparison to a "normal" (non-MD) workspace, where is pixel corresponds to a vector. On the other hand, the box size is adaptive, and with a reasonable threshold ($\mathcal{O}(100)$) the size of vectors is probably reasonable.

For some initial tests, two system tests that involve `MDWorkspace` were profiled: `SXDAnalysis` and `TOPAZPeakFinding`. So far we observe the following:

- Threading gives very little speedup.
  - It looks like the actual insertion of events into the workspace is not threaded?
  - If it were, there are many locks which cause slowdown.

[13] As long as no one has the idea of using the Intel Xeon Phi — and for Mantid in the current state that would probably be a bad idea.

[14] There is an algorithm that could be used to combine the results from different nodes at the end: `MergeMD`.

  - After initial insertion, the boxes are recursively split until a threshold number of events is reached. This is threaded, and will also hit the locks in `MDBox::addEvent` (but no two threads will ever attempt to add to the same box).
  - Removing the lock does not really improve things.

- Most of the time is spent in `ConvertToDiffractionMDWorkspace`. The cost is mostly from adding events to the workspace.
  - There are many page faults. They seem to be responsible for 20% or more of the total runtime.
  - A lot of time is spent in locking/unlocking a mutex and spin locks. The reason is probably `MDBox::addEvent` which was designed to be thread safe.
  - Significant amount of time spent in memory allocation and deallocation. In particular, unless the workspace already has settled to a particular number of boxes, there will be *several* allocations and deallocations per event: If a large number of events is added to a workspace with few boxes, those will contain many events. In a call to `MDEventWorkspace::splitAllIfNeeded`, this is then subdivided. Subdivision takes the events from the `MDBox` and puts it into a `MDGridBox`. All boxes in the grid box are then subdivided again, if necessary, and so on. The number of reallocations is related to the maximum number of levels in the workspace, and will also contain an *additional* factor due to reallocation by `std::vector`, which happens when adding the events one by one.
  - A division in `MDGridBox::addEvent` may also cause slowdown (unclear) — this is trivial to eliminate.

Possible improvements:

- To avoid the horrendous amount of memory allocation and deallocation, implement a two-pass version of `MDEventWorkspace::splitAllIfNeeded`. The first pass should determine the new box layout. The second pass should (maybe) pre-allocate the event vectors in each box, then copy the events to their new location, and finally deallocate the old storage.[15] How would this be implemented? For each `MDBox` in the current layout:
  - If we need to split, create `MDGridBox`.
  - Instead of copying events into the new `MDGridBox`, make a vector of indices which gives for each event the target `MDBox` inside the new `MDGridBox`.
  - Recurse the creation of `MDGridBox` and update the indices when necessary. Do not make new lists, just update the to level one (just figure out a simple indexing scheme to assign indices to boxes at various levels, or use pointers).
  - When no sub-divisions happen anymore, loop over the list and add each event to its target box.
  - It *may* pay off to keep track of the number of events for each box and reserve the corresponding space in the vector of the target box before adding events. This must be tested — `std::vector` is quite efficient so this does not necessarily pay off.[16]

[15] This will also drastically reduce the number of calls to `MDBox::addEvent` and maybe make the mutex in there less relevant.

[16] It might pay off for the memory consumption, however — reserving the correct amount of memory beforehand will prevent large over-allocation that happens when the vector is growing.

8 General topics

8.1 SIMD (vectorization)

- The `cmake` build is currently not setup properly for optimal results. For example, on my Ubuntu machine `gcc` 4.9 produces SSE code, whereas the hardware supports AVX2.[17]
  ⇒ modify build flags to specify `-march=native` (or equivalents for other compilers).
  Note that we do not obverse any speedup of a full Mantid run (for a few system tests). This would probably only be relevant in case we introduce code that relies heavily on SIMD.

8.2 Threading

This presentation by Steven Hahn covers the current threading in Mantid: `https://github.com/mantidproject/documents/blob/master/Presentations/Parallel%20Macros.pptx`. Maybe the most significant result is that in the majority of system tests there is no speedup when enabling threading. Without further analysis it is hard to tell why, but so far I have encountered the following things:

- In many of the system tests a lot of files are loaded, which is typically slow on the systems we are running the tests on.
- The timings in the tests sometimes include the verification of the result, which in some cases takes longer than the actual test.
- Some algorithms are bound by memory bandwidth, and thus do not benefit from the additional compute power by adding more threads. This might be improved by things like cache-blocking that I mention in my document (but I don't think it will be easy).
- Some other performance issues that I described (such as the `Instrument` code and `shared_ptr`).

These issue must definitely be understood in more detail. The first step should probably be to pick a more representative set of system tests.

8.3 Hybrid parallelization (threading vs. MPI)

Should we use a single MPI rank per node (with many threads), or several MPI ranks per node? I have heard/read (more than once!) that people found the hybrid approach (threading + MPI) to be inferior to a pure MPI-based parallelization. I do not think this would be the case for Mantid, given that a lot of the threading is basically trivial based on splitting according to spectra. Furthermore, Mantid has lots of "overhead", and duplicating this many times if there are many ranks on a node seems inefficient. So my current feeling is that we should run one MPI rank per node — under the assumption that we threads all algorithms that loop over all spectra (this is probably already the case for most relevant algorithms).

[17] This implies a 4× lower theoretical peak floating point performance.

8.4 Parallel Mantid

- What is the status and current usage model of MPI in Mantid?
- Is there any existing code that supports the parallelization model discussed earlier?
- Is there any existing MPI based code in Mantid that might interfere with the parallelization model discussed earlier?

We want to run a data reduction with the instrument distributed to MPI ranks. How do we do this?

- An instrument must be split up into *sub-instruments*.
- Source and sample position must be part of each sub-instrument.
- Monitors should probably be part of each sub-instrument.
- Normal detectors should be contained in exactly one sub-instrument.

What needs to be done?

- Code for automatically determining which detectors should be part of which sub-instrument.
- All relevant `Load` methods or algorithms must be capable of loading data for sub-instruments. This could either mean that each rank opens a file and reads data of relevance or only the master rank reads and then redistributes the data by means of MPI.
- All relevant `Store` methods or algorithms must be capable of storing data for sub-instruments.
- Algorithms that require information from other spectra need do be modified to include MPI calls, etc.
- Algorithms that are not aware of other spectra need no changes.
- Data for monitors in the event stream must somehow be duplicated or redistributed, such that all sub-instruments get the data.

9 TODOs

- Gather information on all required algorithms.
  - Determine scaling behavior and current performance limits.
  - Are there a few that dominate the run-time?
    ⇒ Targeted optimization.
- Figure out what changes will be necessary for a detector-based parallelization with MPI throughout a reduction run.
  - Proper “sub-instruments” have to be set up on each rank.
  - Figure out what subtleties might create problems.

    - Monitors need to be available on all ranks.
  - Certain algorithms that do not treat all detectors independently need to be modified for MPI support.

- Instrument code redesign.

- Figure out potential further performance issues that are not specific to particular algorithms.