Execution Performance Analysis of the ABySS Genome Sequence Assembler using Scalasca on the K Computer

Itaru KITAYAMA [a], Brian J. N. WYLIE [a,b,1], Toshiyuki MAEDA [a]

[a] *RIKEN Advanced Institute for Computational Science, Kobe, Japan*

[b] *Jülich Supercomputing Centre, Forschungszentrum Jülich, Germany*

Abstract.

Performance analysis of the ABySS genome sequence assembler (ABYSS-P) executing on the K computer with up to 8192 compute nodes is described which identified issues that limited scalability to less than 1024 compute nodes and required prohibitive message buffer memory with 16384 or more compute nodes. The open-source Scalasca toolset was employed to analyse executions, revealing the impact of massive amounts of MPI point-to-point communication used particularly for master/worker process coordination, and inefficient parallel file operations that manifest as waiting time at later MPI collective synchronisations and communications. Initial remediation via use of collective communication operations and alternate strategies for parallel file handling show large performance and scalability improvements, with partial executions validated on the full 82,944 compute nodes of the K computer.

Keywords. Scalasca, Score-P, Vampir, K computer, bioinformatics.
Topic area. Software and Architectures

Introduction

Understanding the performance of the MPI library through real-life applications on petaflops machines such as the K computer and other Top500 supercomputers is key to successful transition to foreseeable Exascale computing. MPI applications developed and optimized for high-end servers often fail to take full advantage of the capability that today's supercomputers provide in part due to node-to-node communication latencies. Evaluation of the ABYSS-P [1,2] sequence assembler on the K computer demonstrated the issue in scaling; linear scaling stops early at 200 compute nodes and adding more nodes only resulted in moderate improvement until 800 nodes (only 1% use of the K computer resources). Performance decrease was observed when running with 1024 and more compute nodes, and executions with 16384 or more compute nodes were not possible since they exceeded available node memory.

[1]Corresponding author: b.wylie@fz-juelich.de

In the face of the very limited application performance scalability, a tool was required that best suits our needs to find performance bottlenecks. The Scalasca toolset [3] was selected as it provides profiling and tracing of MPI events to quantify bytes transferred, elapsed time in functions or distinct application call paths, and execution traces which can be viewed as timelines with Vampir [4]. Scalasca is used on many of the largest HPC systems, and was recently ported to the K computer to enable large-scale application profiling for the first time.

We discovered frequent MPI point-to-point communications used to exchange data stored in local buffers between nodes at various stages of ABYSS-P execution, and rapidly growing time for MPI point-to-point and collective communication at large scale (more than 1024 compute nodes). In the master/worker parallelisation paradigm employed by ABySS, the master needs to collect all the checkpointing messages sent from workers to transit to other processing phases. Additional serious performance and scalability inhibitors are ultimately determined to arise primarily from ineffective parallel file reading and writing.

1. Test Platform: The K Computer

A detailed description of the K computer can be found elsewhere [5]. The K computer has 82,944 compute nodes connected through the Tofu 6D mesh/torus interconnection network and attached to the Lustre-based Fujitsu Exascale Filesystem (FEFS). Each compute node has SPARC64 VIIIfx 8-core processors, 16 GB of memory, and 4 Tofu network interfaces whose maximum aggregate throughput is 20 GB/s [5–7].

A custom Linux kernel runs on each compute node and for program development Fujitsu Fortran, C, and C++ compilers and MPI library optimized for the K computer are provided [8, 9].

2. Target Application: ABySS

ABySS is a parallel sequence assembler for small to large mammalian-size genomes. ABYSS-P written in C++ is the target MPI application of this paper; the program finds overlapping sequences from the distributed directed graph across the computing nodes and stitches those sequences together to obtain long sequences.

ABYSS-P uses the eager protocol for small, 4 kB messages in the master/worker programming paradigm. During execution the master process (MPI rank 0) is responsible for controlling the entire workflow and lets worker processes transit to different processing phases. The main phases studied in this paper are *LoadSequences* (each process reading data from disk and building a distributed hash table), *GenerateAdjacency* (building a distributed graph), *Erode*, *Trim*, *DiscoverBubbles* and *PopBubble* (removing bad data for quality control), and final *NetworkAssembly* (creating long sequences from the distributed graph).

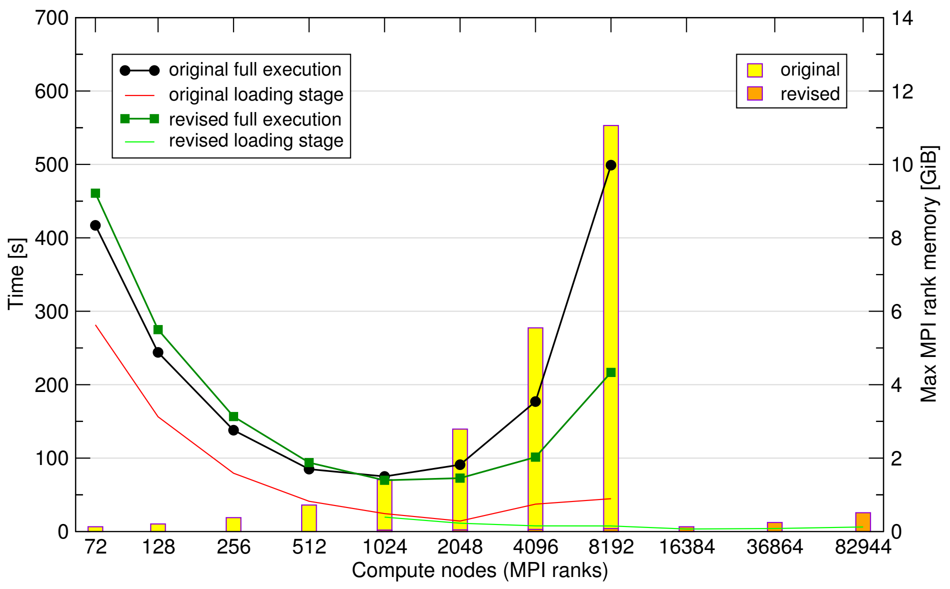


Figure 1. Original and revised ABYSS-P execution scalability on the K computer: bars of maximum MPI rank memory requirement, plot with time of best full execution and loading stage.

2.1. ABYSS-P Configuration

Configuration is done using GNU Autotools cross-compilation to build a SPARC64 binary for the K computer on the x86_64 front-ends using the Fujitsu C++ compiler FCC.

Two MCA parameters of the MPI library specific to K computer must be set prior to launching ABYSS-P: `OMPI_MCA_common_tofu_fastmode_threshold=0` and `OMPI_MCA_common_tofu_max_fastmode_procs=-1`, the former needed to always exchange messages in eager mode to avoid deadlock, and the latter to remove the limit on the number of processes (default 1024) that can be used in eager mode. Although the K computer compute nodes have 8 cores, the job configuration is always set to allocate one process per node to allocate all the usable 14 GB of memory (i.e., excluding that required for system libraries).

Publicly-available data from the *E.coli* MG1655 laboratory strain sequenced on Illumina MiSeq system [10] is used for the experiments described in this paper. The total data read from disk is 6.1 GB, staged in to FEFS local disks.

2.2. ABYSS-P Execution Time Evaluation

Execution timings graphed as circles in Figure 1 show that the best data processing performance of the application (90 seconds) was obtained with around 768 nodes. Adding more nodes up to 1024 did not improve the performance and beyond that execution was slower. Memory required for MPI message buffers also grows rapidly and become prohibitive for executions with 16384 or more nodes. Reports of time elapsed in the key phases and user functions distinguished the most time-consuming *LoadSequences* stage but were insufficient to understand the ABYSS-P execution on the K computer, and therefore comprehensive tools for in-depth parallel performance analysis were required.

3. Profiling and Tracing with Scalasca

The open-source Scalasca toolset was developed to support scalable performance analysis of large-scale parallel applications using MPI and OpenMP on current high-performance computer systems [3]. The latest version uses the community-developed Score-P instrumentation and measurement infrastructure [11]. MPI library interposition combined with automatic instrumentation of application source routines and OpenMP parallel constructs is used to generate detailed call-path profiles or event traces from application processes and threads. Event traces are automatically analysed in parallel to identify and quantify communication and synchronization inefficiencies, and trace visualisation tools such as Vampir can be directed to show the severest instances.

These capabilities proved very valuable for the analysis of ABYSS-P, which has a complex execution structure and uses MPI extensively throughout.

3.1. Methodology

Initial measurements only of MPI events already resulted in large execution traces which hinted at distinct execution phases with very different performance characteristics. Context was provided with manual source annotations of regions (i.e., stages) of particular interest, and found to be essential to understand the complex fragmented nature of the ABYSS-P task-queuing and master-worker paradigm.

Enabling compiler instrumentation of user routines helped clarify the context for this communication, but suffered from significant measurement overheads both in execution dilation and trace buffer memory and file size requirements. To produce execution traces of managable size and reduce dilation, extensive filtering of frequently-executed short-running routines was necessary. While producing a filter for all routines coming from standard C++ libraries is straightforward, more care is required with user-level source routines. Generally those routines which are purely local computation are readily identified via scoring of analysis reports as they are not found on a call-path to MPI communication or synchronization, however, ABySS has a number of routines which have a dual nature that are used in deeply recursing call-paths. For example, `pumpNetwork` is frequently used to process outstanding communication but may also complete computations or busy-wait, so it provides valuable context but at the cost of high measurement overheads.

Following this iterative process of both augmenting and refining the instrumentation and measurement configuration, it was possible to produce rich analyses with insight into ABYSS-P parallel execution inefficiences, via summary profiles and traces. Initial summary measurements were scored to identify appropriate measurement filters that were necessary to avoid extreme measurement dilation and buffer/disk requirements for subsequent trace experiments, producing around 1 GB of event trace data per process, which were automatically analysed in parallel by Scalasca and interactively examined with Vampir.

3.2. Analysis of ABYSS-P Execution

A variety of measurements of the original version of ABYSS-P were done on the K computer with up to 8192 MPI processes.

Scalasca trace analysis of a 1024-node ABySS execution of five minutes on the K computer is concisely presented in Figure 2. Expanding and selecting nodes of the tree for metric *Time* from the left panel reveals that 8.0% of the total execution time [301,000 CPU seconds readable from the scale at the bottom] is used by MPI: 1.3% [3872s] by `MPI_Init`, 1.6% [4872s] by `MPI_Barrier` collective synchronization, 2.1% [6398s] for point-to-point communication and the remaining 3.0% [8915s] in various collective communications (mainly `MPI_Allreduce`). Selecting the *Point-to-point communication time* metric updates panels to its right identifying that it is predominantly [3832s, 60%] for `MPI_Send` in the *LoadSequences* stage shown in the centre panel, and then the right panel shows that this time is broadly distributed across (non-master) ranks: 3.74 ± 0.59 seconds.

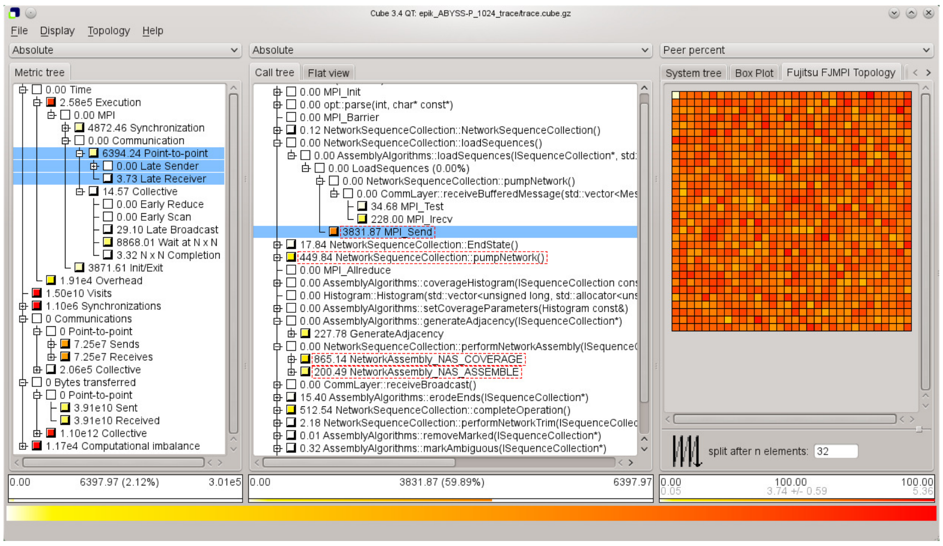
Vampir timeline visualisation of the 1024-node ABySS trace in Figure 3 provides an overview of the execution phases for each process. The *LoadSequences* stage (in yellow) dominates the first half of the execution and reveals eight “waves” as sequence data is read from disk by each process and transferred when the local buffer is full. Although there are the same number of `MPI_Send` events and the same amount of data transferred in each stage, only those in the first wave are clearly distinguished. The *GenerateAdjacency* stage (pale blue) follows, then *Erode* dominated by `MPI_Allreduce` (orange) and *NetworkAssembly* containing a period of extensive `MPI_Barrier` (magenta) usage before the final network assembly with its huge imbalance. The `pumpNetwork` routine which polls for available messages is evident in all stages when point-to-point message transfers may occur, but is often characteristically unproductive “busy-waiting.”

The Scalasca trace analysis quantified point-to-point communication *Late Sender* and *Late Receiver* blocking time sub-metrics as a negligible 3.7s, as receiving buffers are posted early by the application. Much more serious is the blocking time in `MPI_Barrier` and associated with synchronising collective communication. Figure 4 shows that `MPI_Allreduce` which is used at various points in the *Erode* stage (all of which have been selected from the call tree in the central panel) badly affects all of the processes. A popup window reports the severest single inefficiency instance on rank 271 with global severity of 1330 CPU seconds.

When Vampir is directed to zoom in to show the corresponding execution interval, as seen in Figure 5, the third `MPI_Allreduce` instance in *Erode* stands out with rank 271 waiting more than 2.0 seconds for the last of its peers to be ready to proceed. Imbalanced computation between the numerous synchronising reductions results in low efficiency and poor scaling for this ABySS-P execution stage.

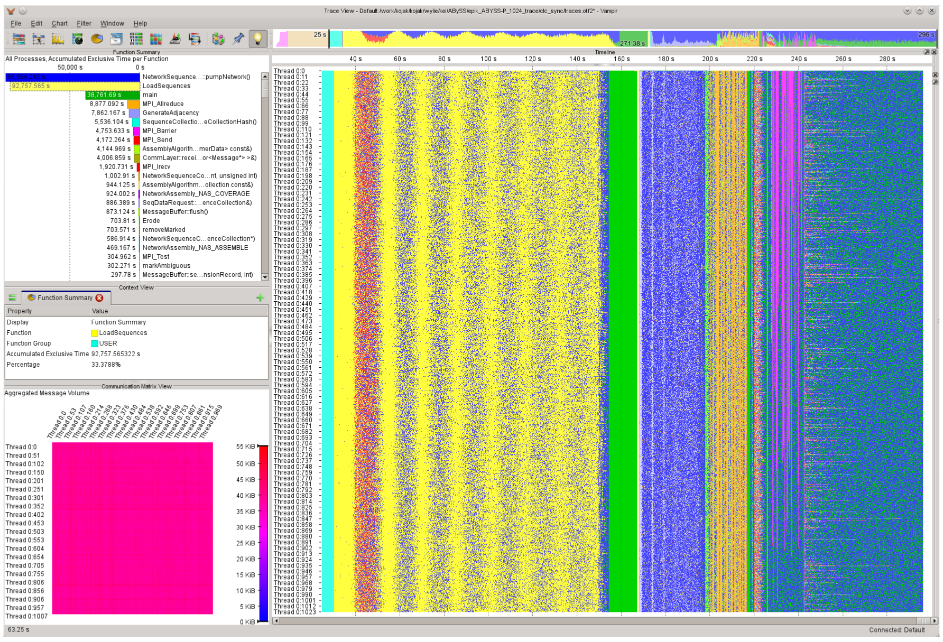
3.3. Revised Instrumentation and Implementation of ABYSS-P

To avoid the measurement overhead arising from automatic instrumentation of routines by the compiler, manual annotation of ABYSS-P execution stages was employed instead, resulting in improved profiles as shown in Figure 6. Particular



```
Cube 3.4 QT: epik_ABYSS-P_1024_trace/trace.cube.gz
File  Display  Topology  Help
Absolute                    Absolute                    Peer percent

Metric tree
0.00 Time
  2.58e5 Execution
    0.00 MPI
      4872.46 Synchronization
      0.00 Communication
        6394.24 Point-to-point
          0.00 Late Sender
          3.73 Late Receiver
        14.57 Collective
          0.00 Early Reduce
          0.00 Early Scan
          29.10 Late Broadcast
          8868.01 Wait at N x N
          3.32 N x N Completion
      3871.61 Init/Exit
    1.91e4 Overhead
1.50e10 Visits
1.10e6 Synchronizations
0 Communications
  0 Point-to-point
    7.25e7 Sends
    7.25e7 Receives
  2.06e5 Collective
0 Bytes transferred
  0 Point-to-point
    3.91e10 Sent
    3.91e10 Received
  1.10e12 Collective
1.17e4 Computational imbalance

Call tree
0.00 MPI_Init
0.00 opt::parse(int, char* const*)
0.00 MPI_Barrier
0.12 NetworkSequenceCollection::NetworkSequenceCollection()
0.00 NetworkSequenceCollection::loadSequences()
  0.00 AssemblyAlgorithms::loadSequences(ISequenceCollection*, std:
    0.00 LoadSequences (0.00%)
      0.00 NetworkSequenceCollection::pumpNetwork()
        0.00 CommLayer::receiveBufferedMessage(std::vector<Mes
          34.68 MPI_Test
          228.00 MPI_Irecv
      3831.87 MPI_Send
17.84 NetworkSequenceCollection::EndState()
449.84 NetworkSequenceCollection::pumpNetwork()
0.00 MPI_Allreduce
0.00 AssemblyAlgorithms::coverageHistogram(ISequenceCollection cons
0.00 Histogram::Histogram(std::vector<unsigned long, std::allocator<uns
0.00 AssemblyAlgorithms::setCoverageParameters(Histogram const&)
0.00 AssemblyAlgorithms::generateAdjacency(ISequenceCollection*)
  227.78 GenerateAdjacency
0.00 NetworkSequenceCollection::performNetworkAssembly(ISequence
  865.14 NetworkAssembly_NAS_COVERAGE
  200.49 NetworkAssembly_NAS_ASSEMBLE
0.00 CommLayer::receiveBroadcast()
15.40 AssemblyAlgorithms::erodeEnds(ISequenceCollection*)
512.54 NetworkSequenceCollection::completeOperation()
2.18 NetworkSequenceCollection::performNetworkTrim(ISequenceCollec
0.01 AssemblyAlgorithms::removeMarked(ISequenceCollection*)
0.32 AssemblyAlgorithms::markAmbiguous(ISequenceCollection*)

System tree   Box Plot   Fujitsu FJMPI Topology
split after n elements: 32

0.00    6397.97 (2.12%)    3.01e5 | 0.00    3831.87 (59.89%)    6397.97 | 0.00  100.00  100.00
                                                                          0.05  3.74 +/- 0.59  5.36
```

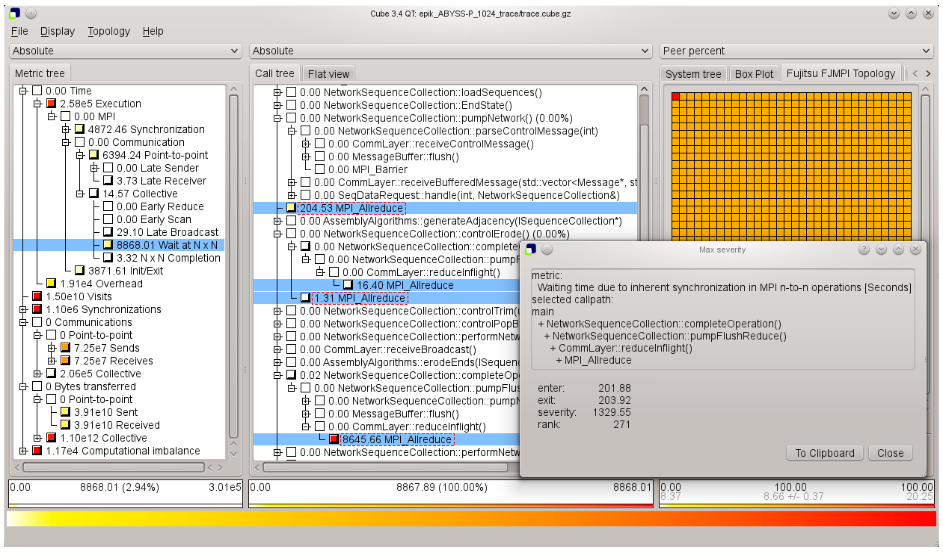
Figure 2. Scalasca trace analysis of original 1024-node ABySS execution on the K computer showing distribution of *Point-to-point communication time* metric (left pane) for the selected `MPI_Send` call-path in the *LoadSequences* stage (centre) for each of the processes (right pane). Values in each panel are colour-coded using the scale along the window footer, and in the various trees represent exclusive metric values for open nodes and inclusive values for closed nodes.



Trace View - Default:/work/kojak/kojak/wylie/kei/ABySS/epik_ABYSS-P_1024_trace/clc_synchtraces.otf2* - Vampir
File Edit Chart Filter Window Help
Function Summary
Timeline
Context View
Communication Matrix View
Aggregated Message Volume
Connected: Default

```
All Processes, Accumulated Exclusive Time per Function
50,000 s                      0 s
                 NetworkSequence...::pumpNetwork()
92,757.565 s     LoadSequences
38,761.69 s      main
8,877.092 s      MPI_Allreduce
7,862.167 s      GenerateAdjacency
5,536.104 s      SequenceCollectio...eCollectionHash()
4,753.633 s      MPI_Barrier
4,172.264 s      MPI_Send
4,144.969 s      AssemblyAlgorith...merData> const&)
4,006.859 s      CommLayer::recei...or<Message*> >&)
1,920.731 s      MPI_Irecv
1,002.91 s       NetworkSequenceCo...nt, unsigned int)
944.125 s        AssemblyAlgorithm...ollection const&)
924.002 s        NetworkAssembly_NAS_COVERAGE
886.389 s        SeqDataRequest::...enceCollection&)
873.124 s        MessageBuffer::flush()
703.81 s         Erode
703.571 s        removeMarked
586.914 s        NetworkSequenceC...enceCollection*)
469.167 s        NetworkAssembly_NAS_ASSEMBLE
304.962 s        MPI_Test
302.271 s        markAmbiguous
297.78 s         MessageBuffer::se...nsionRecord, int)

Property                    Value
Display                     Function Summary
Function                    LoadSequences
Function Group              USER
Accumulated Exclusive Time  92,757.565322 s
Percentage                  33.3788%
```

Figure 3. Vampir presentation of the same trace of original 1024-node ABySS execution on the K computer with horizontal timeline for each MPI process showing ‘waves’ during the *LoadSequences* stage followed by *GenerateAdjacency*, *Trim/Erode*, *PopBubble/Discover_Bubbles* and *NetworkAssembly* stages. The aggregate chart in the window head is provided for navigation when zooming, with profile of function execution times and communication matrix on left side.

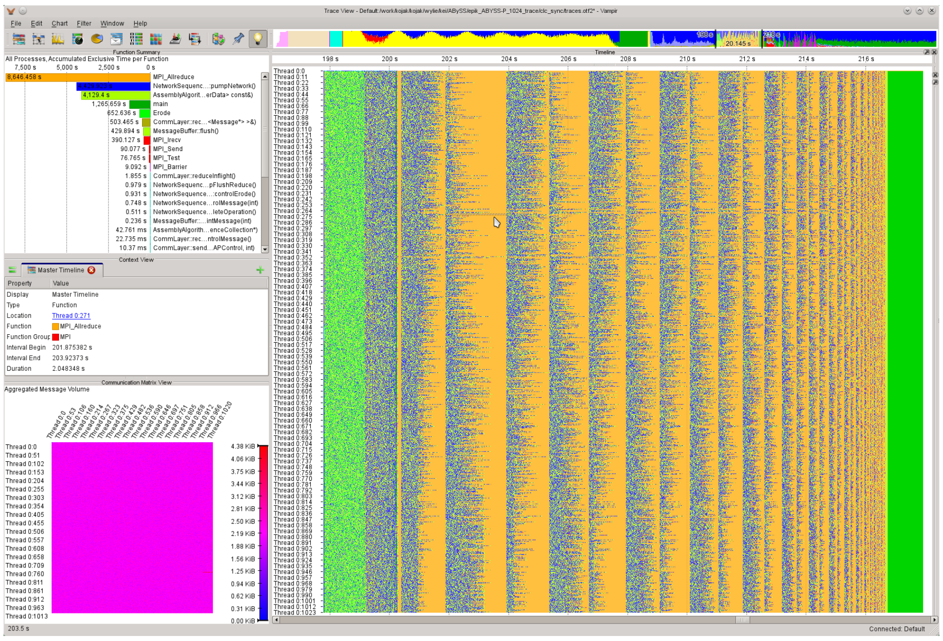


```
Metric tree
0.00 Time
  2.58e5 Execution
    0.00 MPI
      4872.46 Synchronization
      0.00 Communication
        6394.24 Point-to-point
          0.00 Late Sender
          3.73 Late Receiver
        14.57 Collective
          0.00 Early Reduce
          0.00 Early Scan
          29.10 Late Broadcast
          8868.01 Wait at N x N
          3.32 N x N Completion
      3871.61 Init/Exit
    1.91e4 Overhead
1.50e10 Visits
1.10e6 Synchronizations
0 Communications
  0 Point-to-point
    7.25e7 Sends
    7.25e7 Receives
  2.06e5 Collective
0 Bytes transferred
  0 Point-to-point
    3.91e10 Sent
    3.91e10 Received
  1.10e12 Collective
1.17e4 Computational imbalance

0.00    8868.01 (2.94%)    3.01e5
```

```
Call tree
0.00 NetworkSequenceCollection::loadSequences()
0.00 NetworkSequenceCollection::EndState()
0.00 NetworkSequenceCollection::pumpNetwork() (0.00%)
  0.00 NetworkSequenceCollection::parseControlMessage(int)
    0.00 CommLayer::receiveControlMessage()
    0.00 MessageBuffer::flush()
    0.00 MPI_Barrier
  0.00 CommLayer::receiveBufferedMessage(std::vector<Message*, st
  0.00 SeqDataRequest::handle(int, NetworkSequenceCollection&)
  204.53 MPI_Allreduce
0.00 AssemblyAlgorithms::generateAdjacency(ISequenceCollection*)
0.00 NetworkSequenceCollection::controlErode() (0.00%)
  0.00 NetworkSequenceCollection::complete
    0.00 NetworkSequenceCollection::pumpF
      0.00 CommLayer::reduceInflight()
        16.40 MPI_Allreduce
  1.31 MPI_Allreduce
0.00 NetworkSequenceCollection::controlTrim(
0.00 NetworkSequenceCollection::controlPopB
0.00 NetworkSequenceCollection::performNetw
0.00 CommLayer::receiveBroadcast()
0.00 AssemblyAlgorithms::erodeEnds(ISequen
0.02 NetworkSequenceCollection::completeOp
  0.00 NetworkSequenceCollection::pumpFlus
    0.00 NetworkSequenceCollection::pumpN
    0.00 MessageBuffer::flush()
    0.00 CommLayer::reduceInflight()
      8645.66 MPI_Allreduce
0.00 NetworkSequenceCollection::performNetw

0.00    8867.89 (100.00%)    8868.01
```

```
Max severity
metric:
  Waiting time due to inherent synchronization in MPI n-to-n operations [Seconds]
selected callpath:
main
 + NetworkSequenceCollection::completeOperation()
  + NetworkSequenceCollection::pumpFlushReduce()
   + CommLayer::reduceInflight()
    + MPI_Allreduce

enter:      201.88
exit:       203.92
severity:  1329.55
rank:          271
```

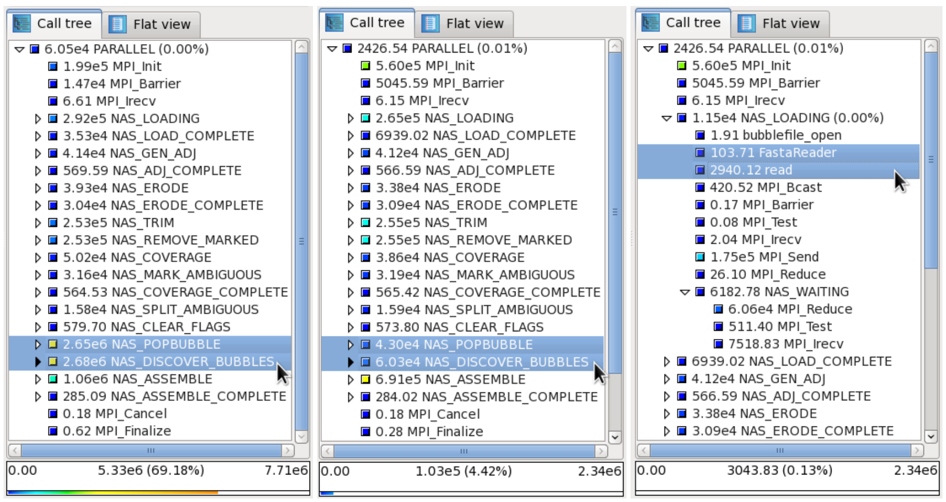
Figure 4. Scalasca trace analysis of original 1024-node ABySS execution on the K computer with severest instance of automatically identified *Wait at NxN* inefficiency in MPI_Allreduce during *Erode* stage detailed in popup window.



```
Function Summary
All Processes, Accumulated Exclusive Time per Function
  8,646.458 s   MPI_Allreduce
  4,429.923 s   NetworkSequenc...:pumpNetwork()
    4,129.4 s   AssemblyAlgorit...erData> const&)
  1,265.659 s   main
    652.636 s   Erode
    503.465 s   CommLayer::rec...<Message*> >&)
    429.894 s   MessageBuffer::flush()
    390.127 s   MPI_Irecv
     90.077 s   MPI_Send
     76.765 s   MPI_Test
      9.092 s   MPI_Barrier
      1.855 s   CommLayer::reduceInflight()
      0.979 s   NetworkSequenc...pFlushReduce()
      0.931 s   NetworkSequence...:controlErode()
      0.748 s   NetworkSequence...rolMessage(int)
      0.511 s   NetworkSequence...leteOperation()
      0.236 s   MessageBuffer::...intMessage(int)
    42.761 ms   AssemblyAlgorith...enceCollection*)
    22.735 ms   CommLayer::rec...ntrolMessage()
     10.37 ms   CommLayer::send...APControl, int)
```

Property	Value
Display	Master Timeline
Type	Function
Location	Thread 0:271
Function	🟧 MPI_Allreduce
Function Group	🟥 MPI
Interval Begin	201.875382 s
Interval End	203.92373 s
Duration	2.048348 s

Figure 5. Vampir analysis zoomed on time interval for *Erode* stage of original 1024-node ABySS execution on the K computer with severest MPI_Allreduce inefficiency instance by rank 271 selected and details of selection in middle pane on left side. Dominance of sequence of reductions is evident in the (zoomed) timeline view and associated execution time profile.



```
Call tree   Flat view
▽ 6.05e4 PARALLEL (0.00%)
    1.99e5 MPI_Init
    1.47e4 MPI_Barrier
    6.61 MPI_Irecv
  ▷ 2.92e5 NAS_LOADING
  ▷ 3.53e4 NAS_LOAD_COMPLETE
  ▷ 4.14e4 NAS_GEN_ADJ
  ▷ 569.59 NAS_ADJ_COMPLETE
  ▷ 3.93e4 NAS_ERODE
  ▷ 3.04e4 NAS_ERODE_COMPLETE
  ▷ 2.53e5 NAS_TRIM
  ▷ 2.53e5 NAS_REMOVE_MARKED
  ▷ 5.02e4 NAS_COVERAGE
  ▷ 3.16e4 NAS_MARK_AMBIGUOUS
  ▷ 564.53 NAS_COVERAGE_COMPLETE
  ▷ 1.58e4 NAS_SPLIT_AMBIGUOUS
  ▷ 579.70 NAS_CLEAR_FLAGS
  ▷ 2.65e6 NAS_POPBUBBLE
  ▶ 2.68e6 NAS_DISCOVER_BUBBLES
  ▷ 1.06e6 NAS_ASSEMBLE
  ▷ 285.09 NAS_ASSEMBLE_COMPLETE
    0.18 MPI_Cancel
    0.62 MPI_Finalize
0.00          5.33e6 (69.18%)          7.71e6
```

(a) Original

```
Call tree   Flat view
▽ 2426.54 PARALLEL (0.01%)
    5.60e5 MPI_Init
    5045.59 MPI_Barrier
    6.15 MPI_Irecv
  ▷ 2.65e5 NAS_LOADING
  ▷ 6939.02 NAS_LOAD_COMPLETE
  ▷ 4.12e4 NAS_GEN_ADJ
  ▷ 566.59 NAS_ADJ_COMPLETE
  ▷ 3.38e4 NAS_ERODE
  ▷ 3.09e4 NAS_ERODE_COMPLETE
  ▷ 2.55e5 NAS_TRIM
  ▷ 2.55e5 NAS_REMOVE_MARKED
  ▷ 3.86e4 NAS_COVERAGE
  ▷ 3.19e4 NAS_MARK_AMBIGUOUS
  ▷ 565.42 NAS_COVERAGE_COMPLETE
  ▷ 1.59e4 NAS_SPLIT_AMBIGUOUS
  ▷ 573.80 NAS_CLEAR_FLAGS
  ▷ 4.30e4 NAS_POPBUBBLE
  ▶ 6.03e4 NAS_DISCOVER_BUBBLES
  ▷ 6.91e5 NAS_ASSEMBLE
  ▷ 284.02 NAS_ASSEMBLE_COMPLETE
    0.18 MPI_Cancel
    0.28 MPI_Finalize
0.00          1.03e5 (4.42%)          2.34e6
```

(b) Improved

```
Call tree   Flat view
▽ 2426.54 PARALLEL (0.01%)
    5.60e5 MPI_Init
    5045.59 MPI_Barrier
    6.15 MPI_Irecv
  ▽ 1.15e4 NAS_LOADING (0.00%)
      1.91 bubblefile_open
      103.71 FastaReader
      2940.12 read
      420.52 MPI_Bcast
      0.17 MPI_Barrier
      0.08 MPI_Test
      2.04 MPI_Irecv
      1.75e5 MPI_Send
      26.10 MPI_Reduce
    ▽ 6182.78 NAS_WAITING
        6.06e4 MPI_Reduce
        511.40 MPI_Test
        7518.83 MPI_Irecv
  ▷ 6939.02 NAS_LOAD_COMPLETE
  ▷ 4.12e4 NAS_GEN_ADJ
  ▷ 566.59 NAS_ADJ_COMPLETE
  ▷ 3.38e4 NAS_ERODE
  ▷ 3.09e4 NAS_ERODE_COMPLETE
0.00          3043.83 (0.13%)          2.34e6
```

(c) Detail of improved reading

Figure 6. Total execution time profile extracts from 8192 compute nodes of ABYSS-P with annotated execution stages and file operations.

care was required to annotate stages of master and workers consistently, such that they combine in the resulting profiles. Furthermore, the *Waiting* stage of workers between completing one computation phase and receiving direction to start the next was also distinguished and associated with the completed phase, as this helped with presenting idling arising from load imbalances.

For an execution with 8192 compute nodes shown in Figure 6(a), 69% of time (650 seconds) is attributed to coupled *PopBubble* and *Discover_Bubbles* stages which are serialised in rank order, resulting in all but one worker process idling while each rank in turn processes its local data and then creates a file. Computational load imbalances reported in ABYSS-P profiling result from the master/worker paradigm employed and associated ineffective parallelisation of file reading and writing, therefore additional manual annotations were incorporated for file operations (shown in Figure 6(c)) which helped identify that these were particularly costly and suffered from large variability.

Configuring executions on the K computer to use files in separate rank-local directories helped reduce filesystem performance variablity, while serialisation costs could be reduced by opening files in parallel, such that time for *PopBubble/Discover_Bubbles* was reduced more than fifty-fold to 12.6 seconds, only 4.4% of the much quicker overall execution time in Figure 6(b). For the *Assemble* stage, originally taking 14% of time (129 seconds), the benefit was a more modest 35% since only 4.5 MB of reconstructed contigs are written while the total length of sequences to put together varies from rank to rank, and the assignment is very imbalanced. The line with square points in Figure 1 shows how the performance of this revised version of ABYSS-P is improved at larger scale.

The MPI point-to-point communication used by ABYSS-P for coordination between master and worker processes requires an eager transfer protocol using prohibitive amounts of message buffer memory for 16,384 or more compute nodes (bars in Figure 1). A first step to avoid this substitutes Fujitsu's efficient

`MPI_Reduce` and `MPI_Alltoall` collectives in the sequence *Loading* stage, and has been validated with successful execution of *Loading* in 6.2 seconds on the full 82,944 compute nodes of the K computer. Applying similar changes to the remaining ABYSS-P stages is expected to permit complete executions at this unprecedented scale, ready for processing much larger sequence datasets.

4. Related Work

Lin [12] compared the capabilities and performance of ABySS with other *de novo* assembly tools, and found that long execution times and large memory requirements were serious constraints. ABySS execution performance with up to 128 processes has been analysed on a Dell/AMD/Mellanox cluster comparing Ethernet and Infiniband DDR networks [13], identifying that MPI communication overhead increases significantly with scale for `Send/Irecv` and `Allreduce`. Georganas et al [14] compare the performance on up to 960 cores of a Cray XC30 of their own parallel genome assembler with ABySS, where scaling parallel file reading and writing is identified as the key bottleneck that they avoided by implementing their own file format.

Call-path profiling of C++ parallel applications is often prohibitive for instrumentation-based tools [15], however, combining MPI tracing with sampling based on interrupt timers has been demonstrated to be effective in such cases [16].

5. Conclusions

ABYSS-P executions on the K computer with 1024 or more MPI processes suffered from a variety of critical performance and scalability issues, which were investigated using the Scalasca toolset. Execution traces from up to 1024 processes were collected, automatically analysed to quantify inefficiency patterns and direct Vampir timeline visualisations. Since automatic routine instrumentation by the C++ compiler and associated filtering of measurement events proved to be costly and awkward, subsequent measurements instead exploited manual source instrumentation to clearly distinguish ABYSS-P execution stages and file I/O operations, carefully matching stages executed by master and worker processes. File handling found to constitute the most serious inefficiencies was remedied by using rank-local directories on the K computer as well as code restructuring to significantly reduce serialisation costs. Scalability beyond 8192 MPI processes additionally required substituting point-to-point messages with efficient MPI collective routines for master/worker coordination, such that initial ABYSS-P execution stages (and associated Scalasca measurement experiments) have now been possible with the full 82,944 compute nodes of the K computer.

Acknowledgments

This research was made possible by using resources of the K computer at the RIKEN Advanced Institute for Computational Science. We are grateful for the dedicated effort from the Operations and Computer Technologies Division that supports the K computer and its users.

References

[1] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J. M. Jones, and İnanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[2] ABySS GitHub Repository, `https://github.com/bcgsc/abyss`.

[3] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[4] Wolfgang E. Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.

[5] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, and Tadashi Watanabe. Overview of the K computer System. *Fujitsu Sci. Tech. J.*, 48(3):255–265, 2012.

[6] Takumi Maruyama, Toshio Yoshida, Ryuji Kan, Iwao Yamazaki, Shuji Yamamura, Noriyuki Takahashi, Mikio Hondou, and Hiroshi Okano. SPARC64 VIIIfx: A New-Generation Octocore Processor for Petascale Computing. *IEEE Micro*, 30(2):30–40, 2010.

[7] Yuichiro Ajima, Tomohiro Inoue, Shinya Hiramoto, Toshiyuki Shimizu, and Yuzo Takagi. The Tofu Interconnect. *IEEE Micro.*, 32(1):21–31, 2012.

[8] Jun Moroo, Masahiko Yamada, and Takeharu Kato. Operating System for the K computer. *Fujitsu Sci. Tech. J.*, 48(3):295–301, 2012.

[9] Naoyuki Shida, Shinji Sumimoto, and Atsuya Uno. MPI Library and Low-Level Communication on the K computer. *Fujitsu Sci. Tech. J.*, 48(3):324–330, 2012.

[10] Illumina TruSeq Data Sets,
`http://www.illumina.com/truseq/tru_resources/datasets.ilmn`.

[11] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. 5th Parallel Tools Workshop, (Sept. 2011, Dresden, Germany)*, pages 79–91. Springer Berlin Heidelberg, 2012.

[12] Yong Lin, Jian Li, Hui Shen, Lei Zhang, Christopher J. Papasian, and HongWen Deng. Comparative studies of *de novo* assembly tools for next-generation sequencing technologies. *Bioinformatics*, 27(15):2031–2037, 2011.

[13] HPC Advisory Council. ABySS performance benchmark and profiling, May 2010.

[14] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yellick. Parallel de Bruijn graph construction and traversal for *de novo* genome assembly. In *Proc. ACM/IEEE Conference on Supercomputing (SC14, New Orleans, LA, USA)*, pages 437–448. IEEE Press, November 2014.

[15] Christian Iwainsky and Dieter an Mey. Comparing the usability of performance analysis tools. In *Proc. Euro-Par 2008 Workshops*, volume 5415 of *Lecture Notes in Computer Science*, pages 315–325. Springer, 2009.

[16] Zoltán Szebenyi, Todd Gamblin, Martin Schulz, Bronis R. de Supinski, Felix Wolf, and Brian J. N. Wylie. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of MPI programs. In *Proc. 25th Int'l Parallel & Distributed Processing Symposium*, pages 640–648. IEEE Computer Society, May 2011.