MPI Queue characteristics of large-scale applications

Rainer Keller and Richard Graham
Oak Ridge National Laboratory
Oak Ridge, TN 37831
Email:{keller,rlgraham}@ornl.gov

ABSTRACT: Application codes running at large process counts may exhibit unexpected communication characteristics, hindering scalability. An optimized MPI-implementation must take these into account to better utilize resources. This paper analyzes the communication and MPI-internal queue characteristics of simulation codes important to the future Oak Ridge Leadership Computing Facility (OLCF). The codes represent a wide mix of scientific applications using current production input data. Open MPI's Peruse interface is used to collect various events such as the length of the unexpected message queue, the time of message stay in the unexpected message queue and the amount of time spent searching the unexpected message queue for already received data.
We show, that applications exhibit patterns over process counts, and that applications have characteristic unexpected message queue usage. We also shown, that process zero almost always has a longer unexpected message queue (due to being the root process in collective operations) and that other applications always have a long unexpected message queue on every fourth process. This paper provides useful data for the application and the MPI developer alike. Knowing the unexpected message queue characteristics allows an MPI implementation, e. g. to tune the search algorithm to minimize the overhead.

KEYWORDS: Application performance, Scalability, MPI Queue characteristics, Cray XT5

1 Introduction

The Message Passing Standard is the most widely used parallel programming paradigm for highly-scalable parallel applications. Since its introduction in 1995 various extensions and updates have been incorporated; the latest standard is MPI-2.2 [7]. While MPI is considered to be the "assembler of parallel programming", additions currently being worked on allow application developers to scale to upcoming and future architectures.

A system that fits this programming model is the Cray XT series, which consists of a family of scalable systems comprising of commodity AMD Opteron microprocessors are connected using a custom 2D or 3D torus network. the so-called SeaStar2+ [3] interconnect. The communication stack that is used beneath the vendor's MPI and Open MPI is Portals [8]. Installations of this series range from the midrange Cray XT5m to the Cray XT5 installations at Oak Ridge National Laboratory. The current highest performing system Jaguar at ORNL consists of 18688 compute nodes each using two AMD Opteron hex-core Istanbul processors per node.

To allow parallel applications to scale to such core counts, communication needs to be optimized on several levels to introduce as little overhead as possible: The application must make as few data exchanges as possible, it should try to decouple communication and computation to try hide the communication and the system software and the communication library should not introduce overheads. When trying to adapt the MPI implementation

to requirements of large-scale applications, it is essential to understand the communication characteristics, such as communication pattern, communication amount and distribution. At large process counts various effects may skew a nicely load-balanced application. In this paper, we study the MPI-internal queue characteristics of several applications important to the next Oak Ridge Leadership Compute Facility (OLCF).

In this paper we introduce a lot of results gathered with a tracing library developed using the Peruse introspection interface [9]. Section 2 provides an overview of Peruse and the implementation of this tracing library. In section 3 application's performance characteristics gathered with this library are presented. Section 4 shows some related work. Finally, in section 5 we provide a short conclusion and give an outlook of this work.

2 Measurement Methodology

In order to retrieve information on the message handling characteristics of an application, we used the Peruse [9] specification, which is implemented in Open MPI [10]. In between the MPI library and the application, a library hooks in using the Profiling MPI-Interface (PMPI).

2.1 Peruse Overview

While the standard PMPI-based interface allows gathering of timing information around MPI calls, the Peruse interface is targeted for performance measurement libraries, to gather information on state-changes within the MPI library. This allows fine-grained tracking of messages within the MPI stack. User-level callbacks are registered with Peruse, to be invoked upon a state-change. These may include incoming messages, or entering the search for already received messages. Here we concentrate on evaluating the events concerning the Unexpected Message Queue (UMQ).

Figure 1 shows the currently defined events in Peruse and their sequence along the send- and receive path; please note that the Open MPI implementation offers an additional event `PERUSE_XFER_CONTINUE` not stated in the Peruse specification, which allows detailed timing information for messages split in a rendezvous message protocol. Peruse does not impose any particular message passing method, any fixed set of events, allows additional events and recommends not supporting a particular event, if this would slow down the particular MPI implementation. The interface is portable in design, by allowing applications or performance tracing libraries to query for supported events using defined ASCII strings. For an overview of the semantics of events we refer to [9].

In order to study the behavior of large-scale application's usage on the unexpected message queue, we use the Peruse Events `PERUSE_COMM_MSG_INSERT_IN_UNEX_Q`, `PERUSE_COMM_MSG_REMOVE_FROM_UNEX_Q`, `PERUSE_COMM_SEARCH_UNEX_Q_BEGIN` and `PERUSE_COMM_SEARCH_UNEX_Q_END`. As Open MPI purposefully only allows a single callback to be attached to an event, we demultiplex events in the registered callback-handler within our library.

2.2 Library implementation

For this study, a tracing library has been developed Hooks using the PMPI-interface are provided for any C- or Fortran-application If the application is linked using this wrapper interface, the only output is a header informing of its initialization on process with rank zero. Apart from the the wrapper and an API described below, there is a set of environment variables with which one sets sizes of internal buffers and alike. Additionally, the library offers an API to initialize with different internal buffer sizes, reset counters, print out detailed information, print collectively reduced statistics and collectively write the detailed buffer information to one file per statistics. Care has been taken to take scalability into account in the API and the usage of underlying communication. For example, for jobs with 30,000 processes, while the collective reduce operations provide limited information, the full amount of data may be written collectively using parallel IO: JaguarPF employs Spider, the center-wide Lustre installation, i. e. data is written to disk over the Infiniband network. This exposed a problem with job sizes of 30,000 processes: while care has been taken to write contiguously in only one collective I/O-call, the proper ROMIO info-flags had to be passed upon file-open to allow write aggregation. Furthermore for big message traces ($\geq$ 4096 UMQ measurements per process), one needs to stripe the file to more than the default four Lustre OSTs. Currently the library does not set striping using an `ioctl` or the Lustre User Toolkit library (libLUT), but rather sets the striping prior the application run.

The library currently provides three different analyzes

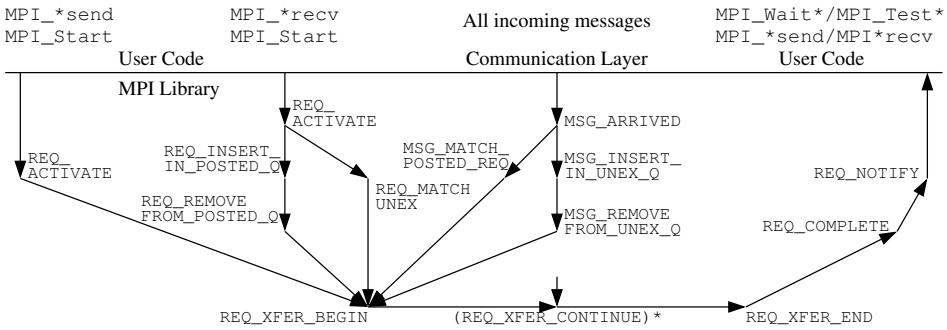

Figure 1: Events and sequence of events during MPI communication as defined in the Peruse specification.

which may enabled separately:

- UMQ maximum and current length, i. e. the maximum length of the process's MPI queue for messages, for which no `MPI_Recv` (or equivalent) has been posted,
- UMQ time, i. e. the time per message staying in the process's MPI queue, which allows analysis, when the actual receive call has been posted,
- UMQ search time, i. e. the time this process has spent searching this MPI process' UMQ.

All the event counters may be reset, or written to disk during the run, to show phases in the application run.

2.3 Measurement

In order not to skew the performance results, all libraries and applications have been adapted to limit the output written to disk as is noted in the corresponding sections. Furthermore all applications and libraries were compiled with the latest default PGI compiler (PGI version 9.0.4) and with the same compilation flags (`-fastsse` to allow vectorization of code, and `-Mipa=fast,inline` for inlining and `-tp=istanbul-64` to optimize the code for the hex-core processor of JaguarPF). The applications were scaled weakly to be able to easily compare communication characteristics. The output files generated above may get large, e. g. in the case of S3D up to one gigabyte. Octave has been used with scripts to load, analyze, post-process and visualize the data.

3 Application Results

3.1 GTC

Gyrokinetic Toroidal (GTC) is a 3-D particle-in-cell code [11]. This code is employed for calculating the turbulence and transport in fusion plasmas such as ITER. GTC is a heavily used code on JaguarPF at ORNL and has been employed as a benchmark for machines at NCCS and NERSC. In [13], the performance of GTC on two Cray XT4 machines installed at NERSC and ORNL are compared.

For this paper, the communication characteristics of up to 8192 processes has been gathered. Similar to [13], we weakly scale the input data set, i. e. the input data set per processor was kept as the same. During the measurement, it turned out that IO in GTC contaminated and randomized the results, i. e. every 5 iterations a diagnosis was performed, which writes the 3D data domain to one file per process. This not only took up a lot of time, but also perturbed the run drastically. Due to the size of the domain per process, on JaguarPF only 4 processes were run per node.

At first GTC is compared using Cray's optimized MPI (Cray MPT 3.5.1) with Open MPI with the Peruse tracing enabled. Figure 2 shows the time of the two dominating functions `charge` (left), `pusher` (middle) and the total time (right). Open MPI does perform equally well in the `pusher` routine, even though microbenchmarks show `MPI_Allreduce` and `MPI_Reduce` employed here to be less performant in Open MPI compared to Cray MPI. Overall the application runs at 8192 processes with Open MPI and Peruse about 2% slower than with Cray's MPI.

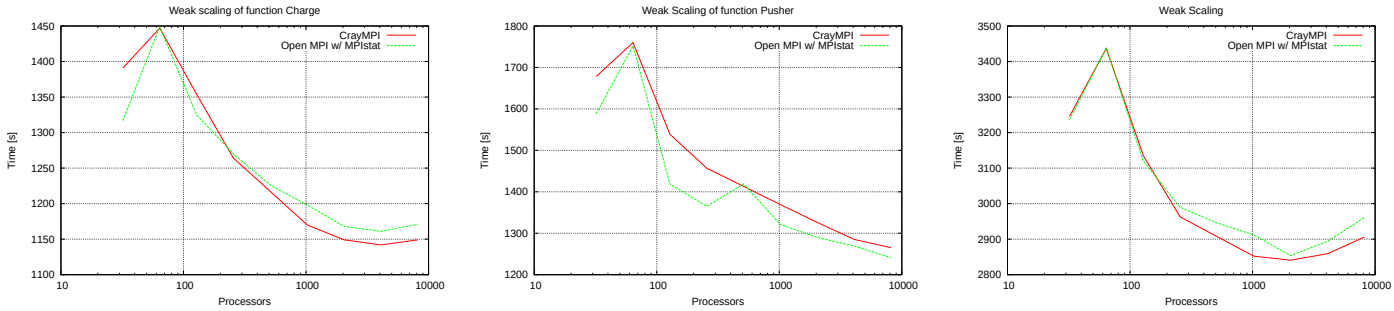

Figure 2: Execution time of GTC routines `charge` (left), `pusher` (middle) and the total time (right).

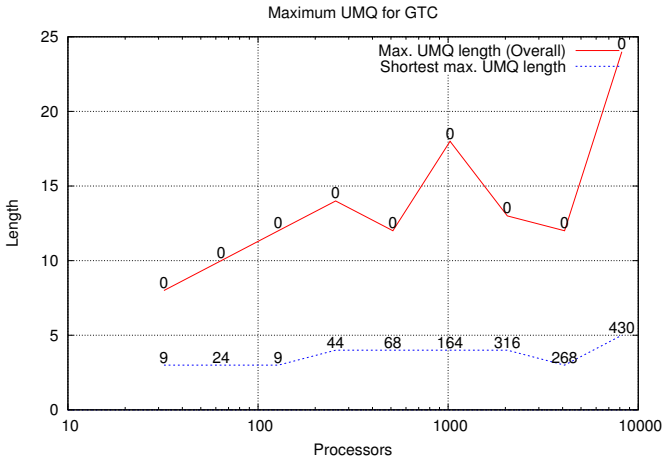

Figure 3: Maximum length of UMQ with rank.

Even though the communication is nearest neighbor between segments of the torus, the results for the unperturbed execution shows that the maximum length of the unexpected message queue (UMQ) slowly increases with process count, as may be see in Fig. 3[1]. Nevertheless, the maximum length of 24 entries in the UMQ on rank zero is very low in comparison with other codes, as in LSMS 3.2. Similarly the average UMQ over all processes and the shortest maximum UMQ stays low as well! The process with rank 0 in this case has the longest UMQ with up to 24 entries in the case with 8192 processes, while the shortest maximum UMQ always stays below 5 entries. When comparing the distribution of the UMQ length, patterns show a correlation between neighbors, as shown in Fig. 4 (left). This may be due to the physical simulation domain, a Torus. The histogram clearly shows that the majority of processes have a short UMQ as may be seen in Fig. 4 (right). In the addendum the histograms and UMQ plots for GTC with intermediate sizes are shown as well in Fig. 9.

As the maximum length UMQ is limited for this code, the time spent searching the UMQ does not offer any surprises. Figure 5 shows the timing information: on the left, the average, minimum and maximum time processes spent searching the UMQ, i. e. in the 32 process case, on average 2.33 ms were overall spent on all processes combined searching the UMQ for messages. This value increases the larger the application scales, i. e. at 8192 processes overall 150.78 ms are spent searching the UMQ. While the average is still low, the maximum accumulated time spent searching the UMQ increases from 3.1 ms at 32 processes to 312.57 ms at 8192 processes, as shown in Fig. 5 (left).

Figure 5 (right) shows the accumulated time, messages spend on the UMQ until the matching receive is posted. Again, the minimum and maximum time increase, except for the last test-case with 8192 processes. Summing up over all messages, and all ranks. In the addendum in Fig. 12, the total search time is shown for 64 to 8192 processes. As may be seen, there exists a pattern of processes, which spend more time searching the UMQ compared to their neighbors. Eventually the histogram of the search times flattens out.

[1]The values above the data point always denote the rank of the process

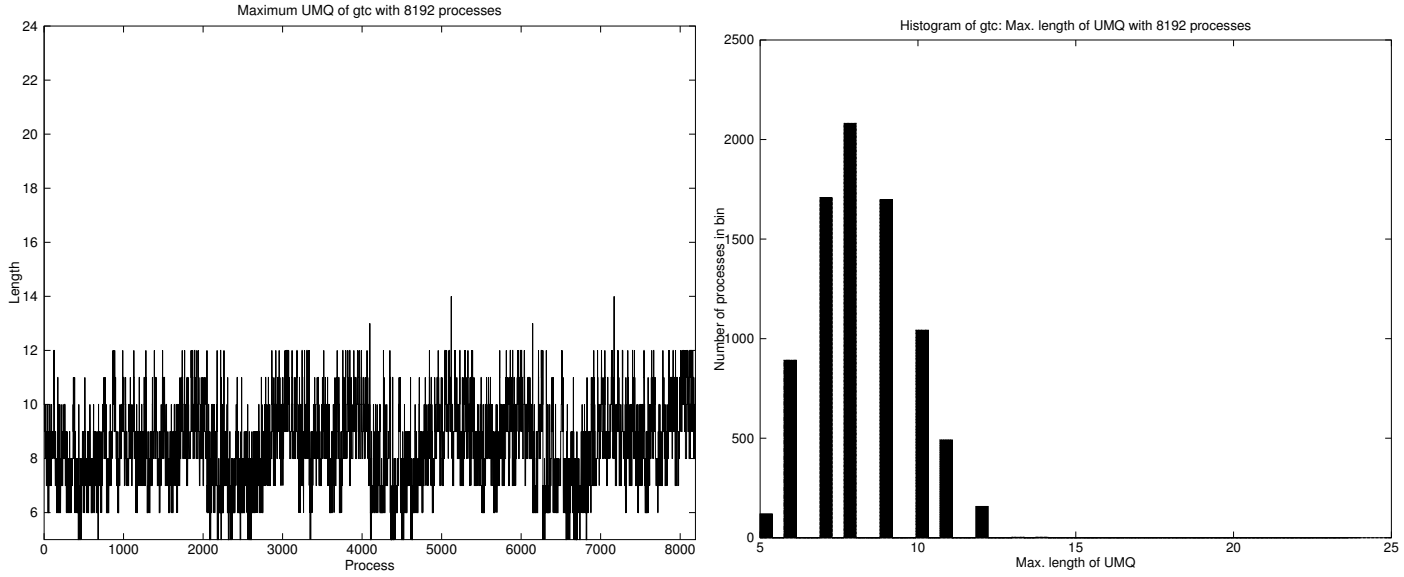

Figure 4: Maximum length of UMQ over ranks (left) and absolute histogram of maximum length of UMQ (right).

3.2 Locally Self-Consistent Multiple-Scattering

The Locally Self-consistent Multiple-Scattering (LSMS) Code [12] simulates the interactions between electrons and atoms in magnetic materials. LSMS is used to perform fundamental studies of the atomistic, electronic, and magnetic microstructure of metals, alloys and semiconductors. Such studies include the description of complex, disordered states of magnetism, and microstructural defects in the materials.

The code has been developed at ORNL since 1994 and was used to study large numbers of atoms, in ensembles of up to 10k atoms. Using the Wang-Langdau (WL-LSMS) scheme, the code may scale up to the whole machine size of JaguarPF; in fact, this work received the Gordon Bell Price at SC 2009 [6]. WL-LSMS distributes chunks of electron configurations to groups of locally optimizing lsms-processes using non-blocking point-to-point communication. However, for the purposes of this paper, this work distribution is not of interest. Rather the communication requirements of the main kernel is of interest, which applies density functional theory for the relativistic wave equation for electron behavior Therefore the `lsms_main` has being employed without the Wang-Langdau method. Here real-world problems require 16 to up to 10k processes.

The first test-case consists of a system of iron atoms in a body-centered cubic crystal structure which is equally spaced at the lattice constant of `Fe` in a body-centered cubic crystal stucture. The system is weakly scaled up to 4096 processes, however to reduce compute time, the number of evaluated energy points and number of iterations is reduced. Without the Wang-Langdau step, the update of potentials is enabled in `lsms_main`. The cut-off radius for interactions is kept at the distance of 12.5 Bohr.

To compare the UMQ of different architectures, `lsms_main` was run with up to 1024 cores on both JaguarPF and Jaguar, the Cray XT5 installation with single-socket quad-code AMD Opteron (Barcelona) processors. Figure 6 shows the maximum and shortest maximum length[2] of the UMQ for Jaguar and JaguarPF in the weak scaling for the iron atom testcase. One may see, the UMQ length is increasing linearly with the number of processes and behaves exactly the same on both machines. To verify these results, the code has been run with larger test-cases as well, as may

[2]That is, the lower bound of all processes' maximum lengths of UMQ in a run.
[3]With Open MPI and MPISTAT at 16 processes 16.41 s per iteration and at 1024 processes 16.31 s per iteration

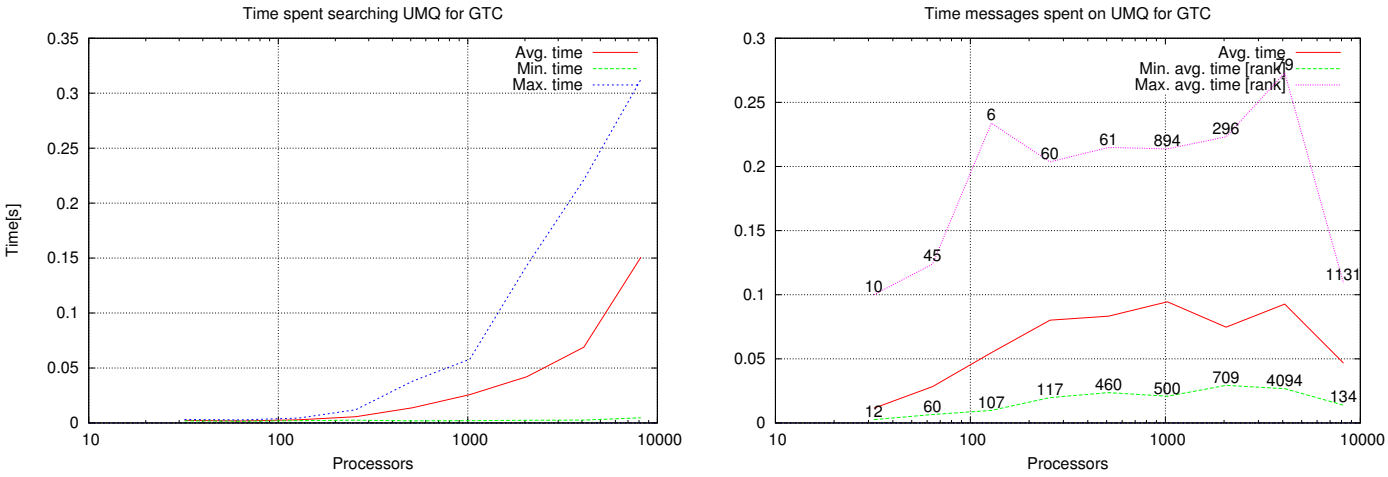

Figure 5: Time spent searching UMQ per process (left). Total time messages spent on UMQ (right).

be seen on the right hand side of Fig. 6. It is noteworthy that the time per iteration, due to the weak scaling stayed approximately the same[3].

Another characteristic may be discovered when looking at the lengths of UMQ. Figure 8 in the addendum shows the normalized histograms of the length of UMQ over several weakly-scaled runs, i. e. the the number of processes within a certain bin of lengths of UMQ are normalized against all processes (aka 100%). Clearly, the normalized histograms reveal three spikes which are particular to this application (and this input data set). The first humps move, e. g. from 50 at 512 processes to 100 at 1024 processes to 195 at 2048 processes (please note, that the X-axis here is clamped at 500 entries).

3.3 S3D

S3D is a massively parallel Direct Numerical Simulation (DNS) solver of fully compressible Navier-Stokes equations including mass and energy conservation laws including detailed chemistry. It is being developed at Sandia National Laboratories [5]. On JaguarPF it has shown to scale to up to the full machine size. For this study, S3D has been scaled from 600 process to up to 18,000 processes

The test cases run with S3D always were set to include 25 grid-points per MPI process on a unit cube, the number of time-steps run were kept at 400 (`i_time_end`), no save fields were output to restart files (`i_time_save`). The simulation included chemical reaction simulation with 4 elements (O, H, C, N) and 22 species.

From an MPI point of view, it uses some collective communication, but most communication is nearest-neighbor point-to-point communication. Of the 25 MPI calls (including `MPI_Init` and `MPI_Finalize`) `MPI_Barrier` is called before `MPI_Allreduce`, which is used to deduce whether a `redo_step` is done. Furthermore for each species, non-blocking point-to-point calls are used to exchange data to compute the gradient. The other calls are used in IO and other service routines, but not the main computational loop.

When analyzing S3D, there are striking patterns in the maximum length of the UMQ at all scales, as may be seen in Fig. 7. These patterns may be made visible when taking the distance between ranks with a UMQ length bigger than a most others, e. g. a maximum UMQ length bigger than the mean. It shows, that for S3D for the 18,000 process case over 98% of the distance between ranks with such a bigger UMQ length is four, the rest is distributed as may be seen in the following table (rank distances of five, six and seven ranks do not occur):

Percentage of occurence				
1	2	3	4	8
0.66%	0.51%	0.35%	98.19%	0.24%

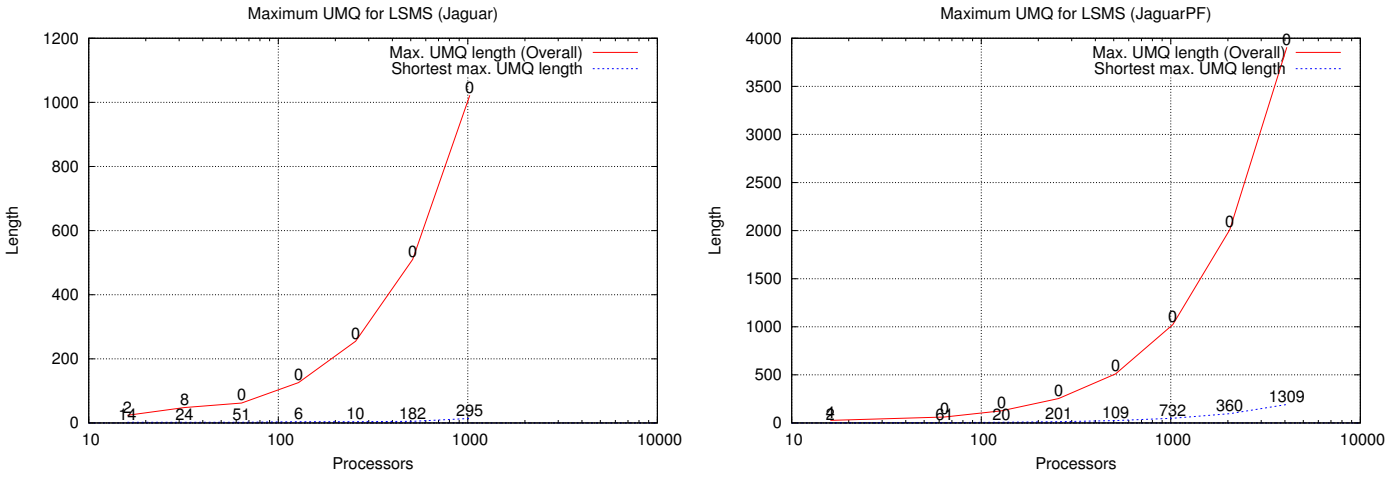

Figure 6: Maximum length of UMQ with rank for Jaguar (left) and JaguarPF (right).

4 Related Work

There does not exist a large body of work with regard to MPI-internal performance analysis. The group with the closest prior work is at Sandia doing similar analysis [1]. In [4] and [2], the authors research the collective and point-to-point communciation requirements of the NAS parallel Benchmarks and three applications (LAMMPS, CTH and ITS) for job sizes of up to 64 processes. Currently, within the MPI3 Forum, the tools working group is discussing new interfaces to collect and expose similar information to application and performance library space. However without further information and lacking implementations, the usefulness, including overheads involved cannot only be speculated.

5 Conclusion

We have shown a new technique to analyze the communication characteristics of large-scale applications based on the Peruse interface. The applications used, stress different characteristics, such as length of the unexpected message queue, time required to search this queue and the time messages stay in this queue. It is shown, that these applications exhibit patterns based on these characteristics. In a future work, applications may be identified using such low-overhead measurement techniques to easily gather finger-prints of applications and possibly inefficient communication characteristics. With this library MPI implementations may tune their message matching and searching algorithm to fit the characteristics of the application (or process rank therein.)

5.1 Acknowledgments

We would like to thank Ramanan Sankaran, Markus Eisenbach and Joshua Ladd for access to source code, input data sets and valuable discussions. This work was supported by the U.S. Department of Energy and performed at ORNL, managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725.

5.2 About the Authors

Rainer Keller is a Postdoctoral research assistant at Oak Ridge National Laboratory. His interest include large-scale parallel applications and parallel tools. At the time of publication, he will heading the Group Applications, Models and Tools at HLRS, Stuttgart. He may be reached at `keller@hlrs.de`.
Richard Graham is a distinguished member of the research staff at Oak Ridge National Laboratory, heading the Application Performance Tools group within the Computer Science and Mathematics Division. He may be contacted at `rlgraham@ornl.gov`.

References

[1] Ron Brightwell, Sue Goudy, and Keith Underwood. A Preliminary Analysis of the MPI Queue Characteristics of Several Applications. In *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 175–183, June 2005.

[2] Ron Brightwell, Kevin Pedretti, and Kurt Ferreira. Instrumentation and analysis of MPI queue times on the SeaStar high-performance network. In *Proceedings of the* 17^{th} *International Conference on Computer Communications and Networks*, St. Thomas, US Virgin Islands, August 2008.

[3] Ron Brightwell, Kevin T. Pedretti, Keith D. Underwood, and Trammell Hudson. Seastar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[4] Ron Brightwell, Rolf Riesen, and Keith D. Underwood. An initial analysis of the impact of overlap and independent progress for MPI. In Dieter Kranzlmüller, Peter Kacsuk, and J.J. Dongarra, editors, *Proceedings of the* 11^{th} *European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science (LNCS)*, pages 370–377, Budapest, Hungary, September 2004. Springer.

[5] Jackie H. Chen, Alok Choudhary, Bronis de Supinski, Matthew DeVries, Evatt R. Hawkes, Scott Klasky, Wei K. Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, Ramanan Sankaran, Shirley Shende, and Chun S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, January 2009.

[6] Markus Eisenbach, C.-G. Zhou, D. M. C. Nicholson, G. Brown, J. M. Larkin, and T. C. Schulthess. A scalable method for *ab initio* computation of free energies in nanoscale systems. In *SC*, Portland, Oregon, USA, November 14-20 2009. ACM.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*, September 2009.

[8] Dan Bonachea Paul H. Hargrove, Michael Welcome, and Katherine Yelick. Porting GASNet to Portals: Partitioned Global Address Space (PGAS) language support for the Cray XT. In *Cray User Group Conference*, May 4–7 2009.

[9] Terry Jones and et al. MPI Peruse – an MPI extension for revealing unexposed implementation information. Internet, May 2006. `http://www.mpi-peruse.org`.

[10] Rainer Keller, George Bosilca, Graham Fagg, Michael M. Resch, and Jack J. Dongarra. Implementation and usage of the PERUSE-interface in Open MPI. In B. Mohr, J. Larsson Träff, J. Worringen, and J.J. Dongarra, editors, *Proceedings of the* 13^{th} *European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science (LNCS)*, pages 347–355, Bonn, Germany, September 2006. Springer.

[11] Zhihong Lin, Taik Soo Hahm, Wei-li W. Lee, William M. Tang, and Roscoe B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281:1835–1837, September 1998.

[12] Yang Wang, G. M. Stocks, W. A. Shelton, D.M.C. Nicholson, W.M. Temmerman, and Z. Szotek. Order-n multiple scattering approach to electronic structure calculations. *Phys. Rev. Lett.*, 75(11):2867–2870, October 1995.

[13] Xingfu Wu and Valerie Taylor. Using processor partitioning to evaluate the performance of mpi, openmp and hybrid parallel applications on dual- and quad-core Cray XT4 systems. In *Cray User Group Conference*, May 4–7 2009.

6 Addendum

6.1 Maximum Queue length of UMQ

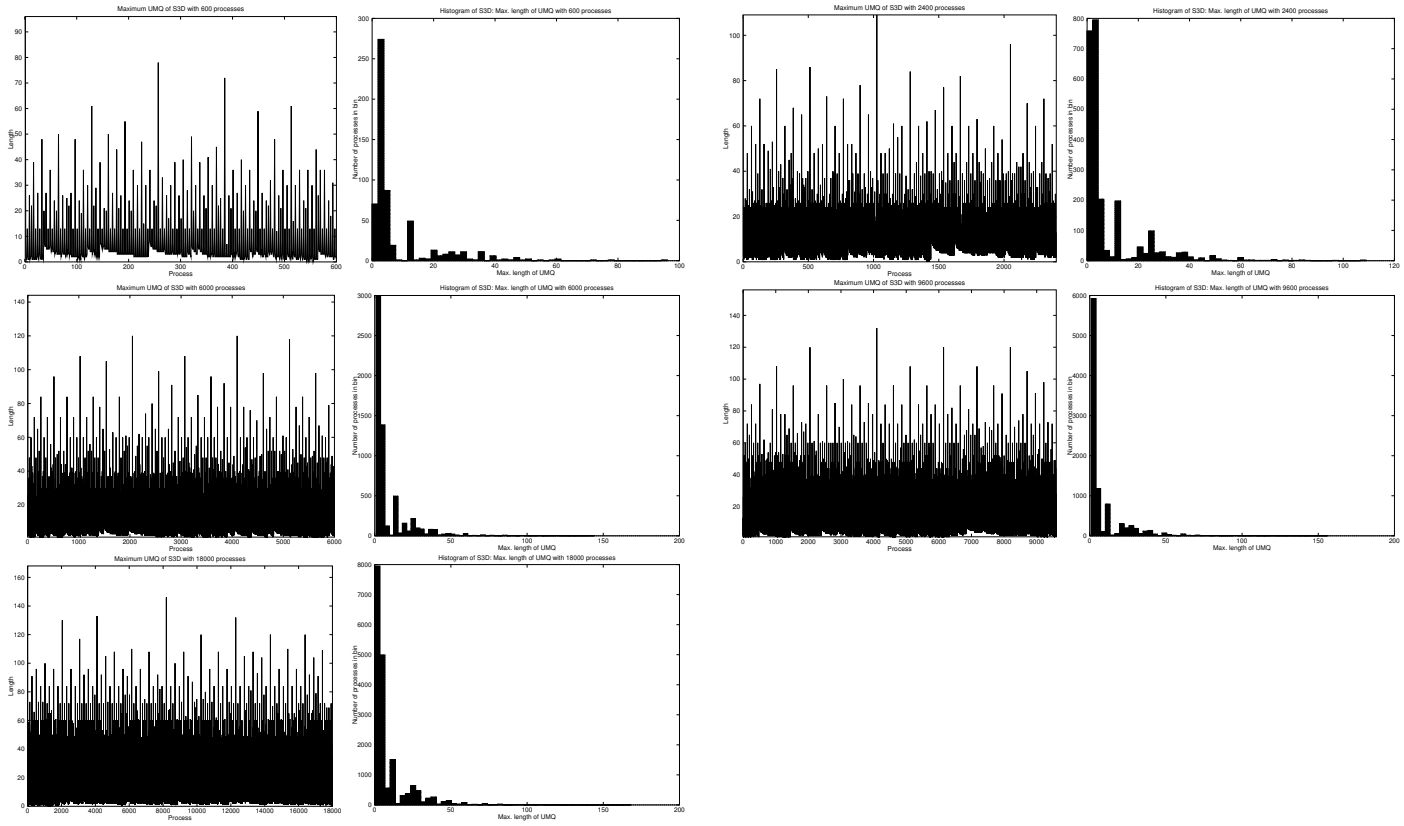

Figure 7: S3D on JaguarPF

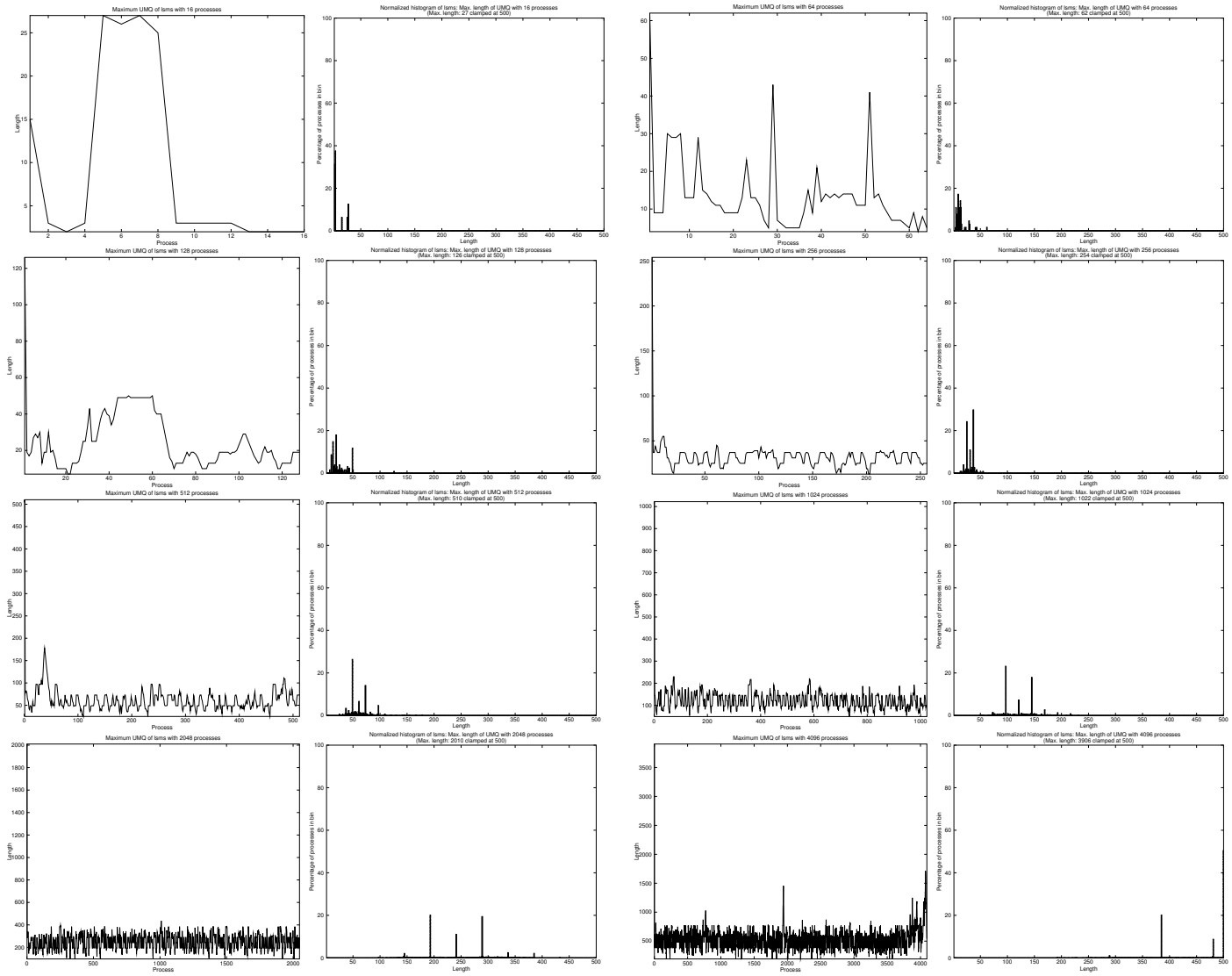

Figure 8: LSMS on JaguarPF

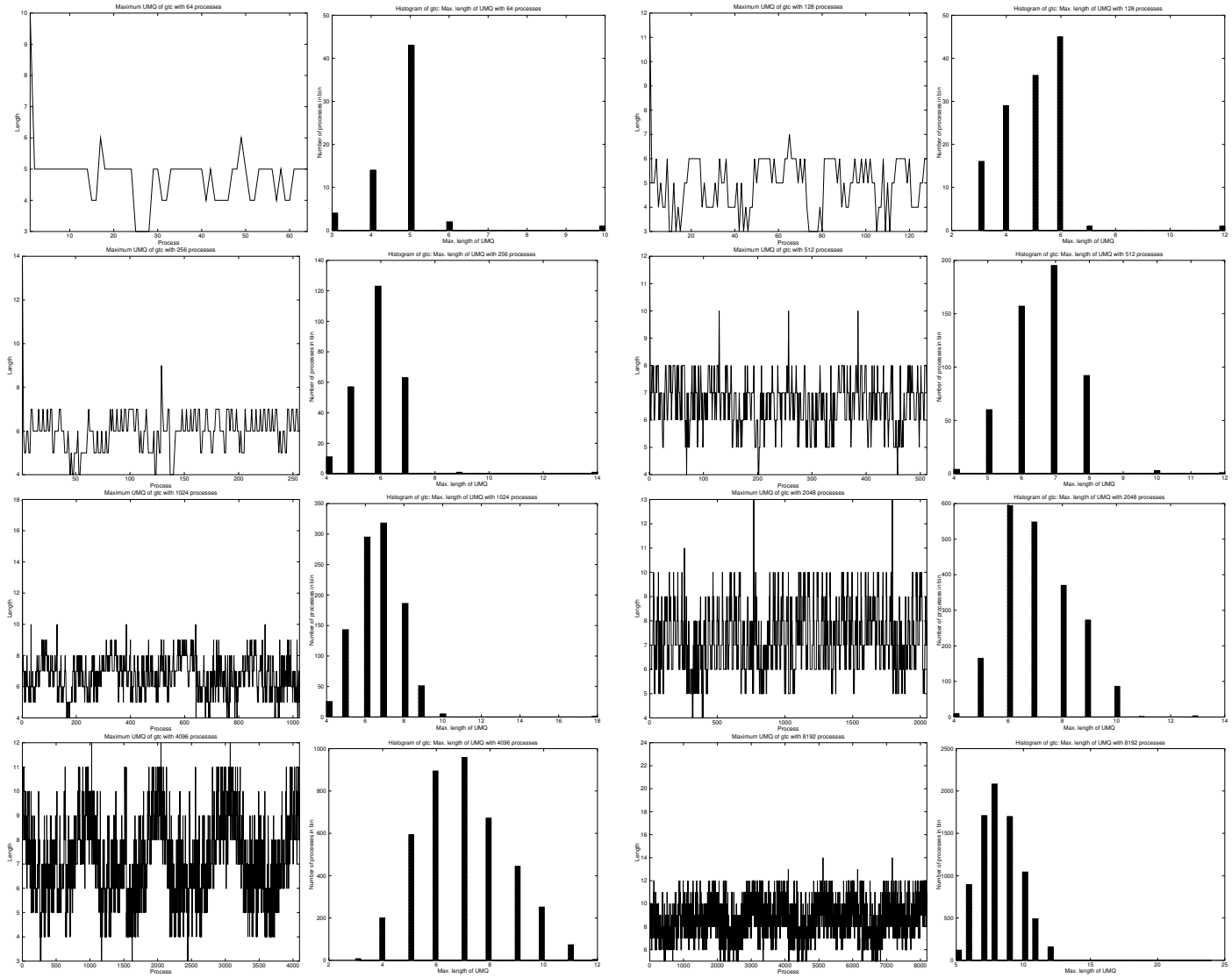

Figure 9: GTC on JaguarPF

6.2 Total search time of UMQ

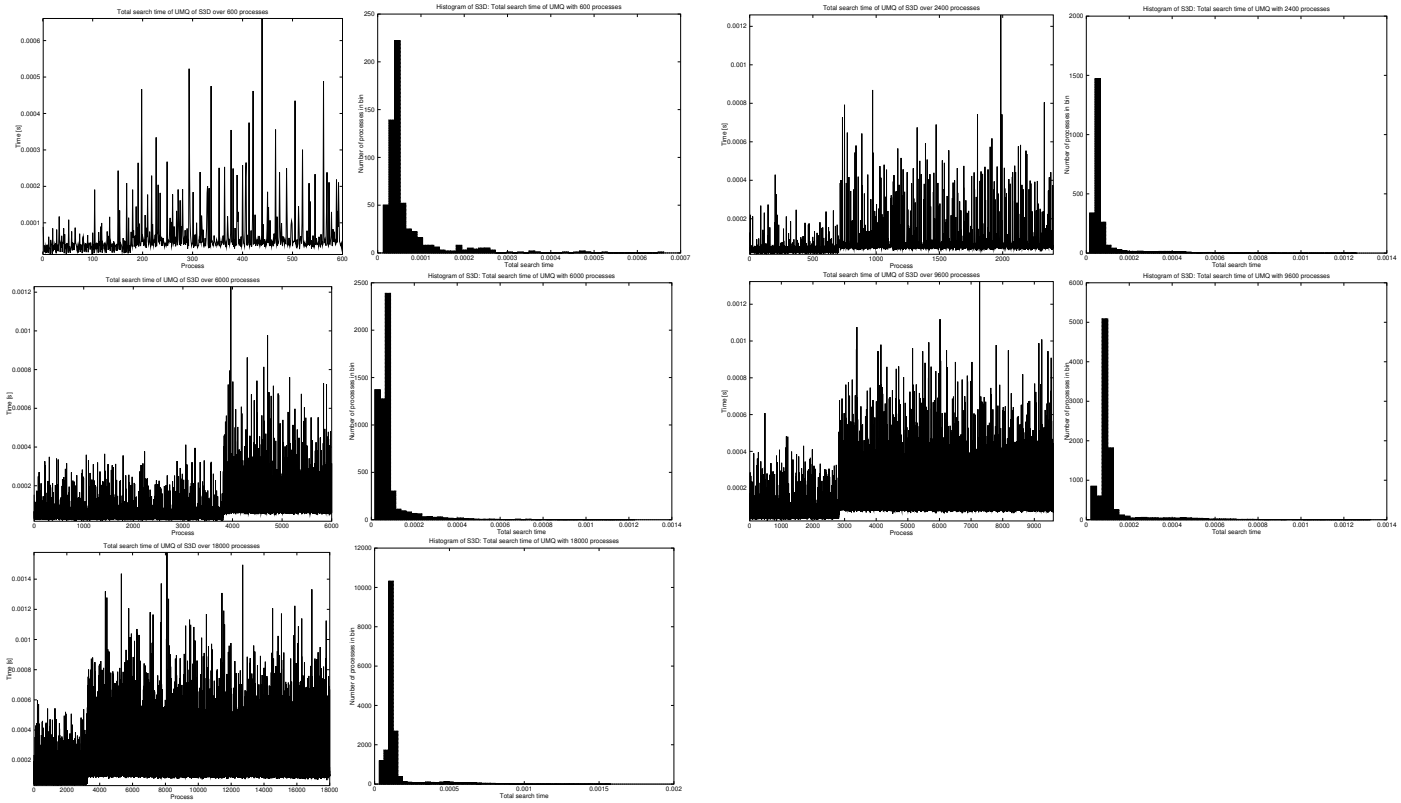

Figure 10: S3D on JaguarPF

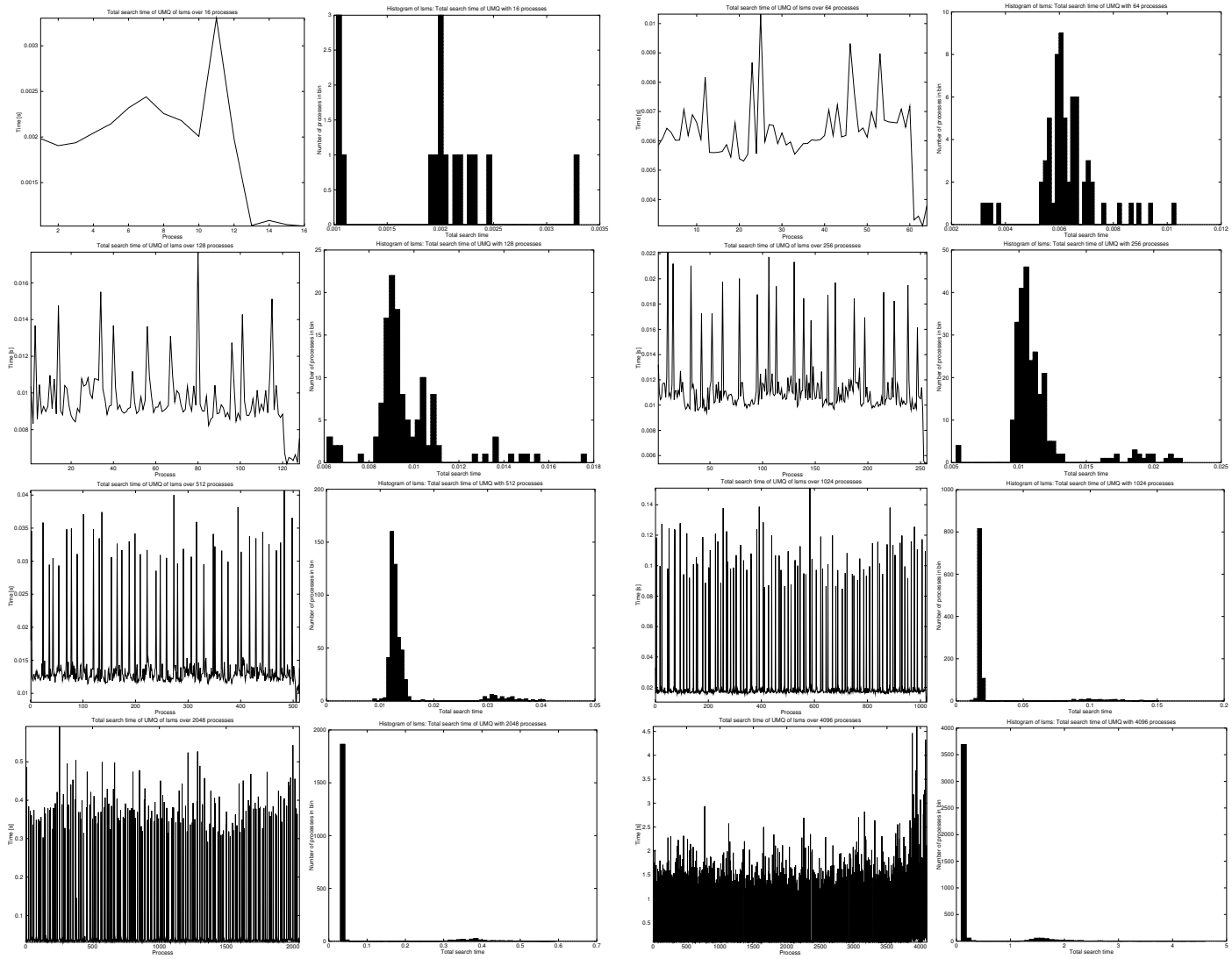

Figure 11: LSMS on JaguarPF

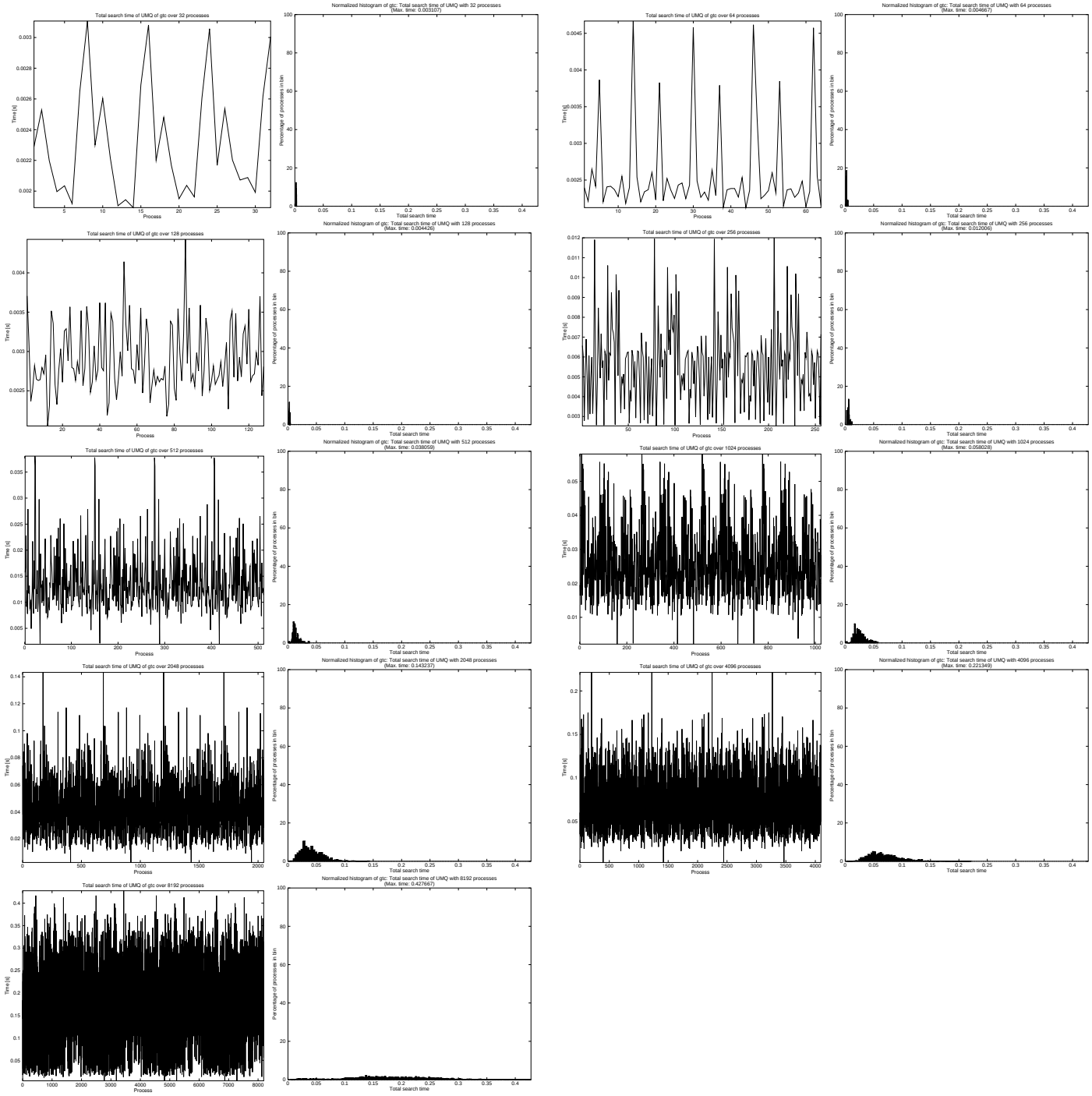

Figure 12: GTC on JaguarPF

6.3 Distance in Ranks

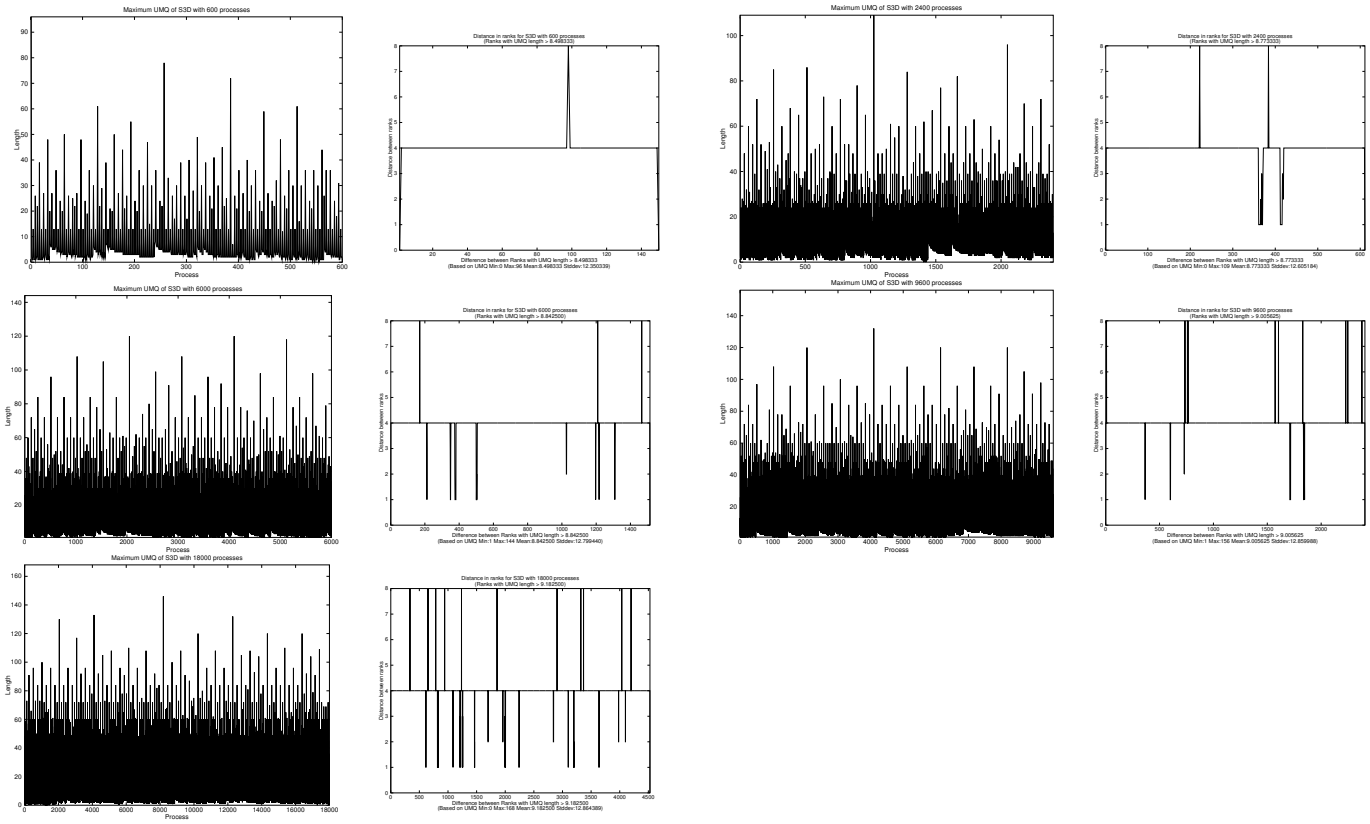

Figure 13: S3D on JaguarPF