Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Optimizing I/O performance and application of results on Linear Scaling Methods for Quantum Hall Transport Simulations

Al. Charalampidou[a,b]*, P. Korosoglou[a,b], F. Ortmann[c], S. Roche[c]

[a]*Greek Research and Technology Network, Athens, Greece*
[b]*Scientific Computing Center, Aristotle University of Thessaloniki, Thessaloniki 54124, Greece*
[c] *Catalan Institute of Nanotechnology, Spain*

Abstract

This study has focused on an application for Quantum Hall Transport simulations and more specifically on how to overcome an initially identified potential performance bottleneck related to the I/O of wave functions. These operations are required in order to enable and facilitate continuation runs of the code. After following several implementations for performing these I/O operations in parallel (using the MPI I/O library) we showcase that a performance gain in the range 1.5 - 2 can be achieved when switching from the initial POSIX only approach to the parallel MPI I/O approach on both CURIE and HERMIT PRACE Tier-0 systems. Moreover, we showcase that because I/O throughput scales with an increasing number of cores overall the performance of the code is efficient up to at least 8192 processes.

1. INTRODUCTION

The Quantum Hall Effect (QHE) is one of the most intriguing phenomena in physics. It was discovered 30 years ago by solid state physics and outreaches nowadays to related disciplines as well. Strongly rooted in condensed matter physics since its discovery, the QHE has been a major character in experimental physics ever since, recently proved by the role played by the integer QHE (IQHE) in the discovery of graphene.

Our focus has been put on the simulation of the IQHE under realistic magnetic fields with linear sample size scaling (order-N). For the simulation of real systems, many time steps are necessary and continuation runs are required in order to work around typical wall time limits of 24 hours commonly encountered on Tier-0 HPC systems.

Our main goal has been to tackle with a potential performance bottleneck related to the I/O of wave functions which is required in order to enable and facilitate continuation runs. The initially implemented restart mechanism required that a number of restart files were written prior to the end of an initial run (one file per MPI process with a fixed size of 127 MB per process). These files are thereafter read through the continuation (restart) job and the computation continues. Although the size of output/input file per process is relatively small, the number of files (typically of the order of 10^3) calls for special treatment to avoid performance decrease.

The code in itself is not I/O intensive. Thus, no algorithmic performance bottlenecks in terms of parallelism up to the targeted number of cores have been foreseen.

2. METHODS

The main objective of this project has been to test and replace the standard POSIX I/O that was used in the code for reading and writing the wave functions in order to allow for continuation runs. The file size per core is fixed at 127 MB (binary file) regardless of the number of cores used (partition size).

According to the conclusions of a previous study made on CURIE Tier-0 system [1], MPI I/O is likely to become comparable to an only POSIX approach when a high number of cores is used and for certain configurations. Therein it is also shown that MPI I/O throughput tends to increase with the number of cores. Moreover, the use of other parallel high level I/O libraries such as HDF5 and PNetCDF is not recommended, as further performance gain has not been achieved.

Thus the improvement for the given case study was initially envisioned through switching paradigms towards parallel I/O by using an MPI I/O strategy. Towards this end we implemented different flavors of MPI I/O. Initially we focused on writing all wave functions into one single shared file using MPI I/O. This was implemented in two ways firstly by independent I/O (in which case I/O is performed independently by each MPI process) and secondly via collective I/O (all processes perform I/O synchronously). Both implementations were tested on CURIE for two partition sizes of size 1024 and 8192 cores each and on HERMIT for a partition size of 1024 cores. This primary set of results is provided in the following section.

We have tried to achieve further optimization of parallel I/O performance by tampering Lustre file parameters (striping size and striping unit) as well as ROMIO hints. Our results from these studies are presented in the following section.

Our next and final approach has been the implementation of parallel MPI I/O to more than one shared files and finally to one separate file per process (using MPI instead of POSIX which has been the initial implementation). Once again these have been tested on CURIE on partition sizes of 1024 and 8192 cores and on HERMIT on 1024 cores.

Within the application processes perform write and read operations on double complex numbers. The basic unit of data access that is used for MPI calls is the primitive data type MPI_DOUBLE_COMPLEX. Access in memory and files is contiguous.

3. RESULTS

Single shared file approach

The first set of benchmark tests on CURIE focused on the impact of independent versus collective I/O parallel modes. The results from those tests led us to the conclusion that collective parallel mode was 1.3 faster than the independent mode. However, collective MPI I/O using a shared file has been 22-23 times slower than POSIX I/O as can been seen also in Figure 1 where throughput results from all three cases are included).

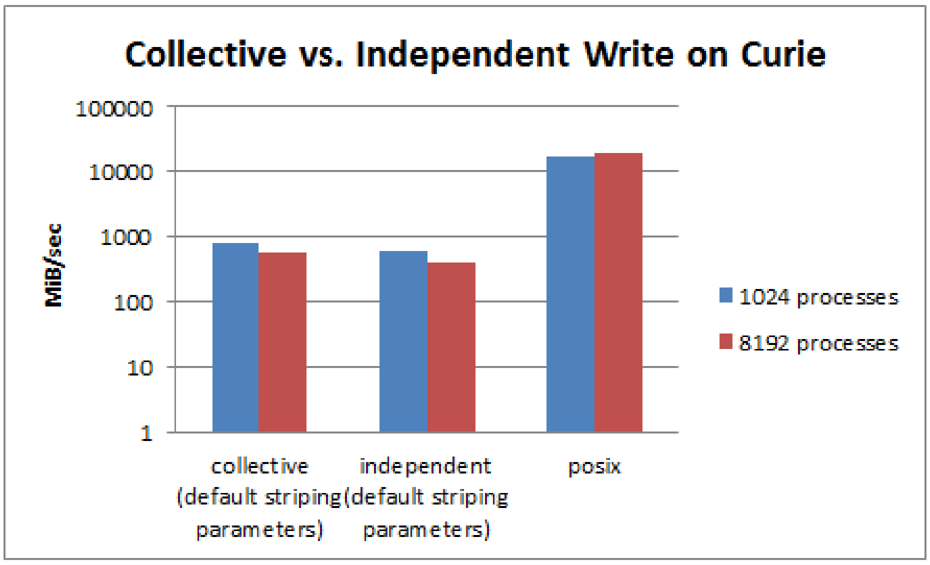

Figure 1 - Comparison of Collective and Independent I/O performance with 1024 and 8192 MPI processes on CURIE (best results shown). Collective I/O exhibits better performance by a factor of approximately 1.3 for both write and read operations. Compared to the initial POSIX implementation both approaches implemented are much slower

More benchmark tests were performed on Curie and Hermit in order to evaluate these results and optimize performance. The basic parameters that were further tuned for optimizing performance have been the Lustre filesystem striping parameters and ROMIO hints.

Lustre striping parameters

One of the initial investigations has been to better fine tune the underlying Lustre file system striping parameters, by changing the default value of striping factor on Curie and Hermit. The default striping factor (4) exhibits fairly low performance when compared to higher values as can be seen in Figure 2.

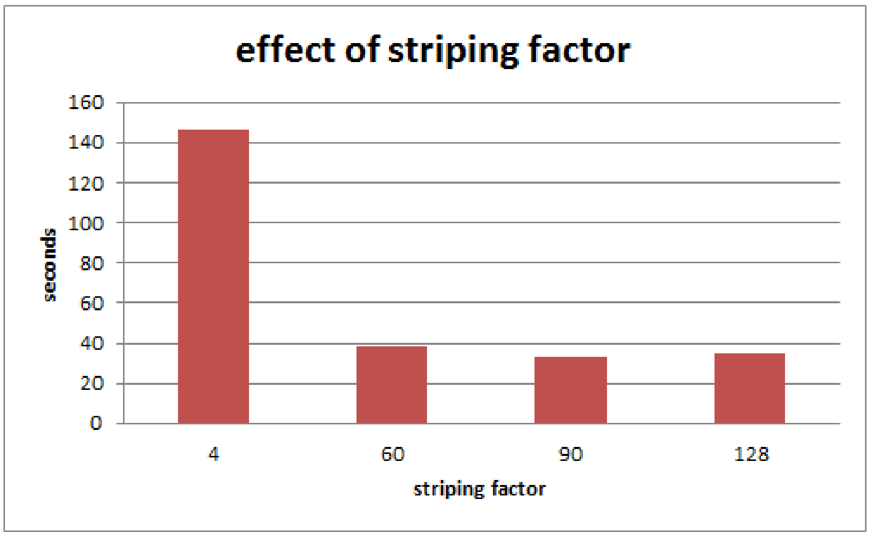

Figure 2 - Effect of striping factor on CURIE Lustre file system (time of write operation to a single shared file is compared) - Less is better

The effect of tuning the striping unit on CURIE has also been studied. As can be seen in Figure 3 having a striping unit of 32 MB produces the best write throughput results. Note that this is comparable to the suggested stripe size of 64 MB.

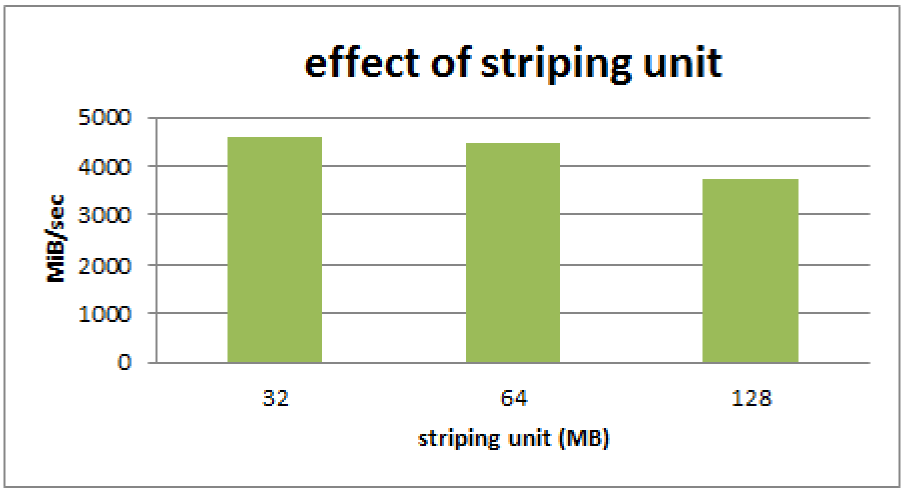

Figure 3 - Effect of striping unit on CURIE Lustre file system. Write throughput to a single shared file is measured (higher is better). Optimal results are received for a striping unit of 32 MB

ROMIO hints

ROMIO hints that have been examined refer to collective buffering and data sieving. As can be seen in Figure 4 the usage of file hints may improve I/O performance, but nonetheless does not outperform the POSIX I/O throughput. Regarding data sieving technique, a small set of tests have been performed, disabling and enabling data sieving with different values for collective buffering. However, significant differences with respect to I/O performance when compared to the default hint values were not observed in any of the cases. Thus, for the case under study further improvement when tuning ROMIO hints has not been measured. The effect of other ROMIO hints in I/O performance has not been examined within this study.

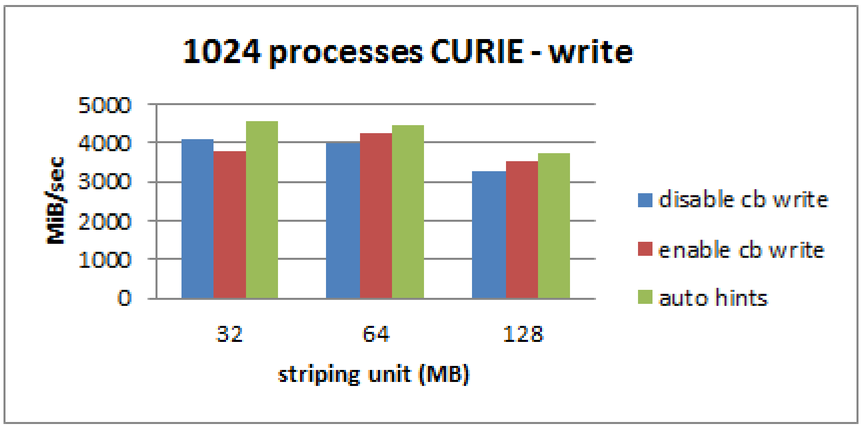

Figure 4 - Test runs with cb_write ROMIO hint enabled and disabled and with striping factor 128.The effect of file hints on I/O performance is not significant (higher is better), thus auto hints have been preferred. Similar results have been drawn via a previous case study performed on CURIE [1].

Several shared files approach

Usage of 2, 4 and 8 shared files between the processes (instead of just one). In this case, the I/O throughput was increased with the number of files. However, even so the performance is lower that in the case of plain POSIX I/O.

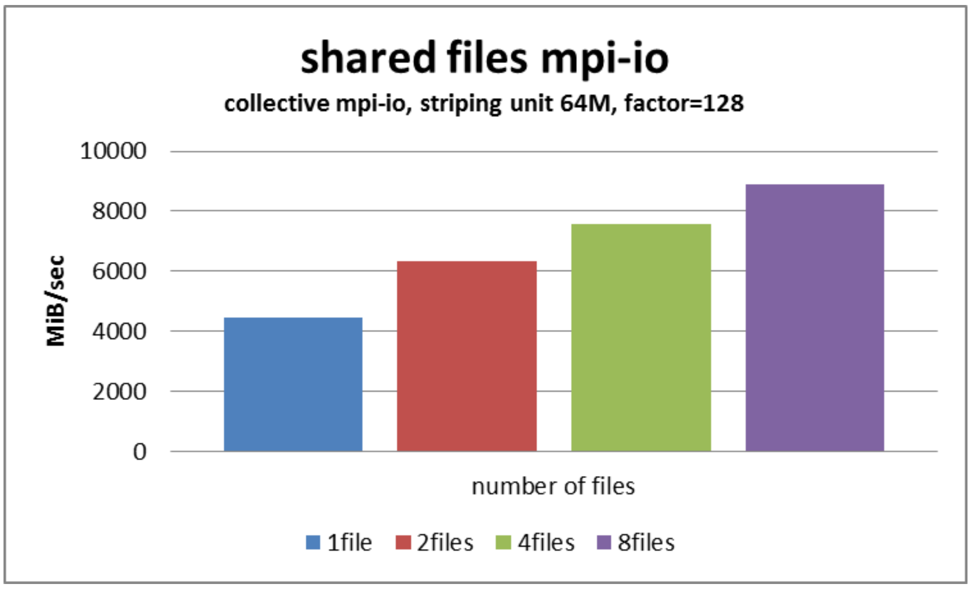

Figure 5 - Throughput (best) write results when using 1, 2, 4 and 8 number of shared files among 1024 MPI processes on CURIE. Our results showcase that when more than one shared files are used on Curie better performance is achieved. This throughput is however lower than the one achieved with one distinct file per process and plain POSIX I/O

MPI I/O to separate files

Our final approach, given the results obtained thus far, has been to use MPI I/O routines writing to and reading from separate files (one file per process). We concluded that this approach can increase overall throughput compared to POSIX I/O under certain configurations both on CURIE and HERMIT Tier-0s. In Figure 6 we compare the overall write throughput achieved on CURIE when using a number of shared files among processes, and separate files (one per MPI process) using MPI and POSIX I/O routines. As can be seen a performance gain of approximately 1.2-1.5 is achieved for both partition sizes (of 1024 and 8192 cores) when using MPI I/O to write to separate files.

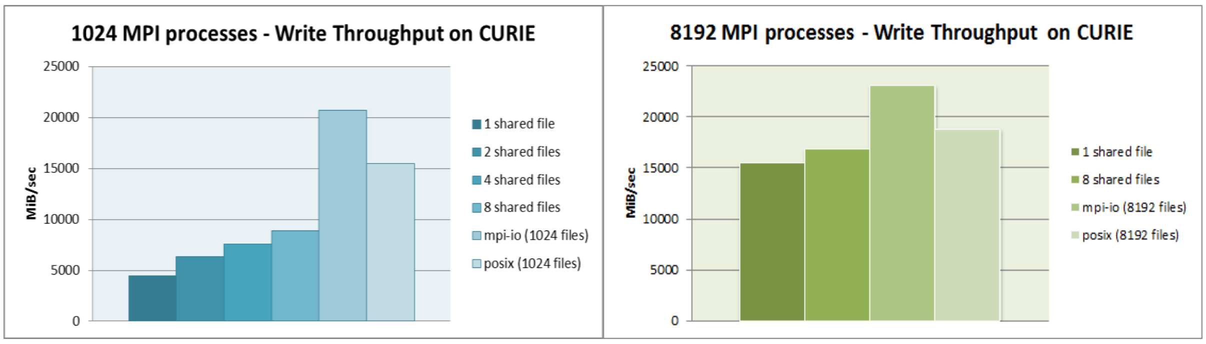

Figure 6 - Comparison of best write throughput results on CURIE for two partition sizes (left is on 1024 cores and right is on 8192 cores). In both cases better throughput results have been achieved when using MPI I/O to separate files (one file per MPI process).

Figure 7 showcases read throughput performances on CURIE for both partition sizes tested. For the large partition size (8192 cores) better read performance is achieved (by a factor of approximately 1.1) when using MPI I/O to separate files. For the small partition size, however, the best performance has been achieved when using the initial implementation of POSIX I/O.

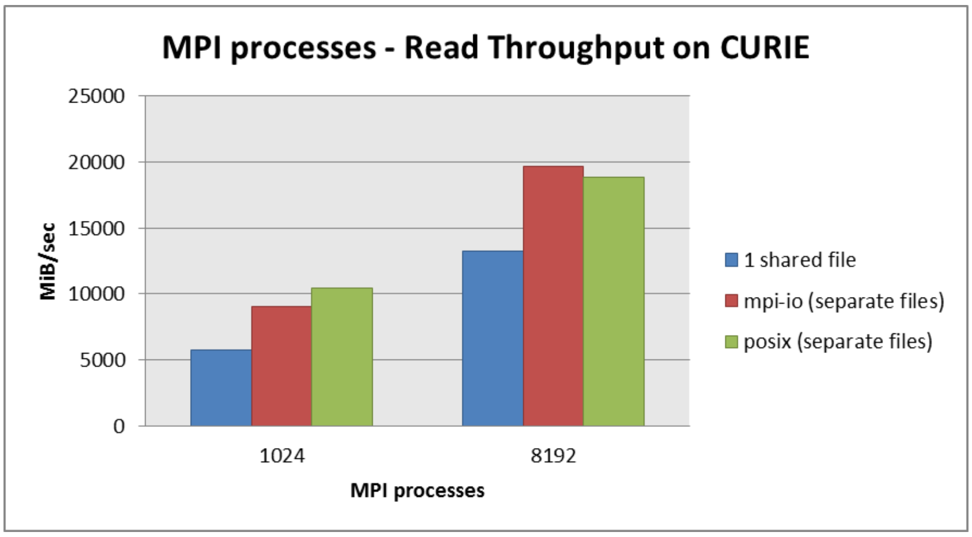

Figure 7 - Comparison of best read performance results measured on CURIE for partition sizes of 1024 and 8192 cores. For the small partition size better read performance was achieved with the initial POSIX implementation. On the large partition an improvement by a factor of approximately 1.1 was achieved via MPI I/O implementation

In Figure 8 the best and worst output throughput of write operations using MPI I/O and POSIX I/O to separate files is showcased. An approximate gain in the range of 1.5-2 when using MPI I/O to separate files instead of POSIX I/O is evident.

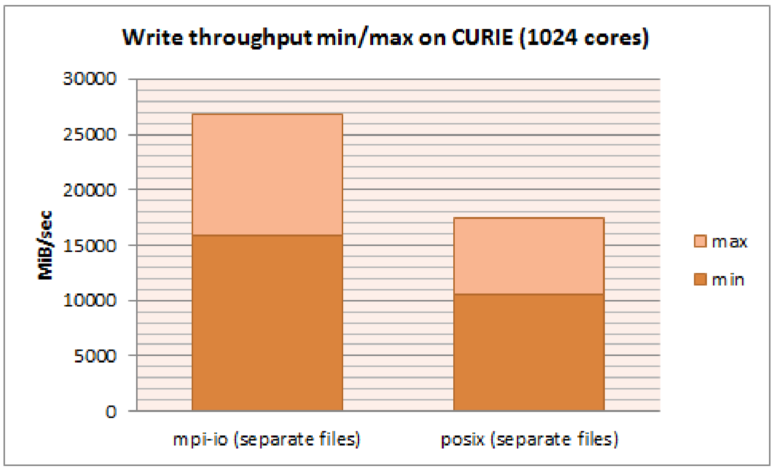

Figure 8 - The minimum and maximum values from all write performance tests on Curie are shown. A performance gain in the range 1.5-2 can be achieved in most cases

In Figure 9 the average write and read throughput results achieved on HERMIT and on 1024 cores are given. For write operations a gain in the order of 1.2-1.5 has been achieved via the implementation of MPI I/O in comparison to POSIX I/O to separate files. For read operations our best results do not exhibit any significant difference.

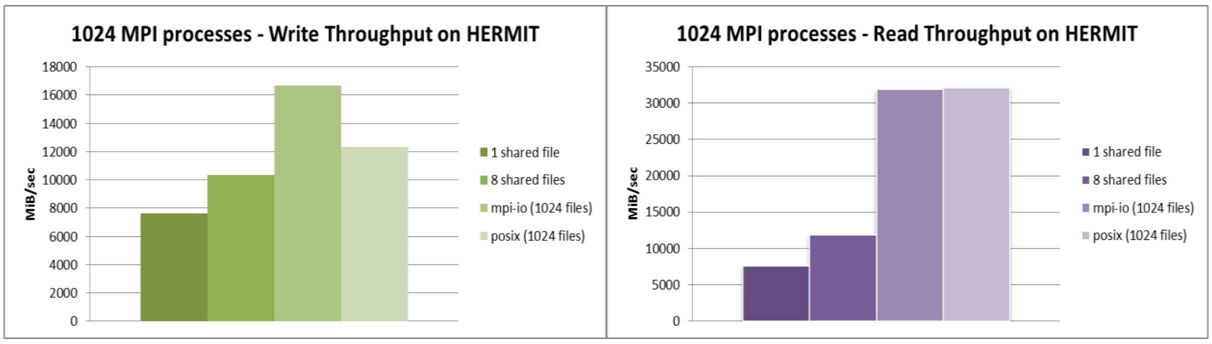

Figure 9 - Average throughput write and read results measured on HERMIT using 1024 cores

4. SUMMARY

In conclusion, our results show that

- MPI I/O using a shared file has been approximately 22 times slower than the initially implemented POSIX I/O
- Collective MPI I/O to a single file has been approximately 1.3 times faster than independent MPI I/O
- MPI I/O using a separate file for each process has been the fastest I/O strategy overall
- Usage of ROMIO hints has not resulted in better timing results

It is worth noting that our results are in general consistent across the two Tier-0 machines that have been used for our investigations (CURIE and HERMIT). In both specific cases the writing performance is improved via the implementation of MPI I/O to separate files per process (in comparison to plain POSIX I/O) by a factor in the range 1.5-2. As far as the reading performance is concerned both approaches on both systems exhibit similar results. It is also worth to mention that the peak performances for read/write operations measured on both systems differ somewhat (i.e. approximately 21000 MiB/sec on CURIE and 17000 MiB/sec on HERMIT for write operations and 19600 MiB/sec on CURIE and 31900 MiB/sec on HERMIT for read operations) something that showcases indirectly the different balances of read/write cache performances on the two systems.

Regarding the initial goals of the project we have showcased that I/O for writing and reading restart files is not a bottleneck with respect to the code itself. I/O throughput scales with an increasing number of cores and overall the performance of the code is efficient up to 8192 processes.

References

[1] P. Wautelet and P. Kestener., “Parallel IO performance and scalability study on the PRACE CURIE supercomputer”, White paper, Prace 2011. http://www.prace-ri.eu/IMG/pdf/Parallel_IO_performance_and_scalability_study_on_the_PRACE_CURIE_supercomputer-2.pdf
[2] Rajeev Thakur, Robert Ross, Ewing Lusk, William Gropp, Robert Latham, “Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation”. http://www.mcs.anl.gov/research/projects/romio/doc/users-guide.pdf
[3] Sun Oracle, Lustre File System, Operations Manual, Version 1.8, http://wiki.lustre.org/manual/LustreManual18_HTML/StripingAndIOOptions.html
[4] Cray, Optimizing MPI-IO for Applications on Cray XT Systems, http://docs.cray.com/books/S-0013-10//S-0013-10.pdf
[5] Cray XE6 Performance Workshop 11-12 July, Optimising I/O on the Cray XE6, http://www.training.prace-ri.eu/uploads/tx_pracetmo/io.pdf

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.