ERDC/CHL CHETN-I-95
November 2018

US Army Corps of Engineers®

Profiling and Optimization of FUNWAVE-TVD on High Performance Computing (HPC) Machines

by Michael Y.-H. Lam, Matt Malej, Fengyan Shi, and Koushik Ghosh

PURPOSE: This Coastal and Hydraulics Engineering technical note (CHETN) discusses the subject of profiling and optimizing numerical codes that are in operational/production use. Profiling is a technique for evaluating the performance of numerical models, and optimization is a method of modifying numerical models to improve performance. In addition, bringing awareness to the subject of profiling and optimization can greatly enhance real-time applicability of many engineering numerical tools/models. Significant run-time gains derived from optimization are discussed, as well as how profilers and programmers can identify key issues with numerical codes. These techniques could be used to manage developers' time when improving code performance effectively. Profiling and optimization techniques are presented using FUNWAVE-TVD version 3.0 (Shi et al. 2016; Malej et al. 2015) as an example. Finally, performance gains in FUNWAVE-TVD, achieved by including parallel distributed multi-processor file input/output (I/O), are presented.

INTRODUCTION: Why profile and optimize? Optimization and profiling may significantly reduce the total runtime of simulations. In the modern computing era, a programmer's time becomes equally important as the actual runtime of the code/model. This becomes especially apparent for frequently changing codes like FUNWAVE-TVD and many other operational physics-based codes. The Performance Optimization and Productivity (POP) group, a European consortium and a Center of Excellence in Computing Application with support from the European Union directed to provide guidance on profiling and optimization of scientific codes, has profiled several high-profile production codes, identifying fundamental deficiencies (Gibson 2017). In turn, addressing these deficiencies has helped developers improve the runtime of their codes between two and six fold (Table 1).

Table 1. Improvements in production codes based on the POP group's recommendations (Gibson 2017).

Name	Developers	Application	Improvement
kWave	Brno University of Tech.	Acoustic waves in tissue-realistic media	2 ×
sphFluids	Stuttgart Media University	Smooth particle hydrodynamics	5 × – 6 ×
GraGLeS2D	RWTH Aachen	Grain growth for polycrystalline materials	2.5 ×

Optimization is time-consuming, and developers' time is valuable; therefore, it is crucial to use their time efficiently. Not all subroutines require equal amounts of computational time; thus it is useful to use a profiler to identify which subroutines of the code are most time consuming. In Figure 1, the output from Open|Speed Shop profiler is shown, listing the most time-consuming subroutines for a given example code. Using this information, it can be seen that if a 20%

optimization is performed on the first subroutine that takes up 50% of the runtime, this yields a 10% overall improvement of the total runtime. Whereas, if a 90% optimization is achieved on the third subroutine accounting for 2% of the simulation, this yields less than a 2% overall improvement in runtime. Following the basic principle of Amdahl's Law (Ware 1972), even though the optimization of the third subroutine was eight times more effective, the optimization of the first routine was five times more efficient in improving the overall performance. Thus, the most time-consuming functions should be optimized first to achieve the greatest performance gains. The choice of routines to optimize must also be balanced with the amount of effort required to achieve those reductions in a given area of the code.

Message Passing Interface (MPI) is a library and a multi-processor programming paradigm, which governs distributed parallel computing by handling the communication among different processors. Each processor/core/thread in principle executes the same code and is assigned a rank number, which is typically used to subdivide the numerical domain amongst the processors/cores and facilitates the exchange of data between the processors. Subsequently, different rank numbers (cores) can branch out and do unique sets of operations based on their specific rank identification number (rank ID). Some of these deviations from the common code can be automated by the MPI collective routines, but many of these are controlled by the programmer/developer. Operating on a multi-dimensional array is computationally expensive and can lead to large memory usages. Thus, to obtain results in a reasonable time frame and distribute memory requirements, it is essential to implement parallel codes.

Scientific codes are complex, especially when utilizing the MPI for parallel processing. Therefore, it is important to understand the complex interplay between different processors (cores/ranks). While a simple debugger may be used to track the flow of an MPI code with a few ranks, the task becomes almost impossible when using 10s, 100s, or 1000s of ranks. Shown on the right in Figure 1 is an example of the output from the Vampir/Vampirtrace profiler, plotting useful work done by ranks (y-axis) as a function of time (x-axis). The profiler organizes the code flow into an easy-to-understand diagram, tracking the useful work done (green bars), down time (red bars), and communication between ranks (black lines). From this plot alone, one can infer that there is an uneven distribution of work among the cores with variable amounts of time spent in a *down-time* state during the time cores/processes are exchanging information. This issue of latency versus bandwidth will be discussed further in the subsequent sections.

PROFILING FUNWAVE-TVD: FUNWAVE-TVD is a nearshore numerical wave model that solves fully nonlinear Boussinesq-type wave equations. It is a phase-resolving wave model, which uses a hybrid finite-volume and finite-difference method. To model the flows in the surf zone, the hybrid model utilizes a total variation diminishing, finite-volume method for nonlinear shallow water equations and the finite difference scheme for dispersive terms. Moreover, the Riemann solver-based shock-capturing scheme takes care of wave breaking and shoreline movement.

The code parallelization is done with a use of an equipartitioned grid domain decomposition technique, where the MPI with non-blocking communication facilitates data communication between processors. Additional details of the FUNWAVE model and the employed numerical methods are not within the scope of this technical note. Hence, the interested reader is referred to Kirby et al. (1998), Malej et al. (2015), and Shi et al. (2016) for further details.

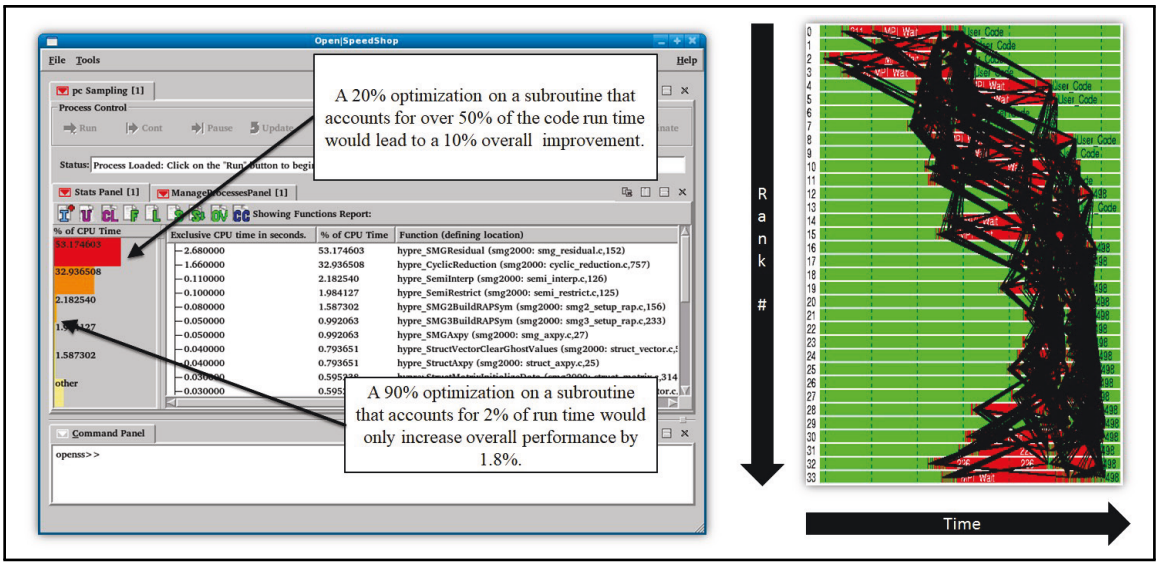

Figure 1. Example of Open|Speed Shop profiler (left) and Vampir/Vampirtrace profiler (right) (from Schulz et al. [2016].

A massively parallel SGI® ICE™ X high performance computer (HPC) known as Topaz is located at the U.S. Army Engineer Research and Development Center (ERDC). Most of the simulations and profiling work presented here has been carried out on Topaz. It contains 3,456 nodes, each with a 36-thread central processing unit (CPU), 124,416 threads in total, and 128 gigabytes of memory (436 terabytes in total). Nodes are connected to each other using four, fourteen data rate, InfiniBand connections, whose theoretical bandwidth is 54 times faster than a standard consumer ethernet connection. For benchmarks performed on 36 or fewer threads, all processes were kept on a single node; thus the interconnects did not play a significant role in benchmarking below 36 cores. Note, however, that many users of production codes may not have access to a similar size of computing resources. Therefore, it is usually a best practice for widely used community-based codes to not be overly optimized for a particular HPC system, but rather the optimization should be as generic as possible. Additional system-specific details on Topaz and other Department of Defense HPC systems housed at ERDC can be found at https://www.erdc.hpc.mil/hardware/index.html.

There are many different key aspects of code performance that can be profiled, such as the level of vectorization, time spent in different places/loops/functions (PCSAMP), checking who calls whom (USERTIME), hardware performance counters (HWCSAMP), MPI trace, I/O tracing, or even memory usage. All of these need a slightly different setup for the portable bash system (PBS) job scheduler/submit script required by the HPC system, and the main differences will be outlined in each subsequent section. Note that the examples listed here are specific to the ERDC Topaz HPC system and its configuration. Briefly, however, the key difference is based upon which environmental variables and modules are being exported/loaded and which executable profiler tool (VTune, Open|Speed Shop, etc.) is being used. In addition, the original source code has to be compiled with extra compiler and linker flags, such as the position independent code (-fPIC), second level of optimization (-O2), dynamic library linking (-Bdynamic), and be in a debug mode (-g) for line numbers. Note, however, that the (-g) option might slow down code execution. Compiling without (-fPIC) and (-Bdynamic) will create a static executable, which

some profiling tools cannot use. It is possible to run Open|Speed Shop without recompiling as long as the object files (.o) are available. Additional details on setting up performance analysis with Open|Speed Shop can be found in Schulz et al. (2016) or by visiting the Open|SpeedShop website and viewing its tutorials and presentations.

Intel® VTune Performance Snapshot. Intel VTune Performance Snapshot profiling provides a brief and straightforward breakdown of the main inefficiencies of a code. An example of how to use Intel VTune Performance Snapshot with FUNWAVE-TVD on Topaz with 36 ranks is shown in Figure 2. When profiling codes, it is important to use real-world examples, as less complex problems may be too simplistic to exhibit code inefficiencies that would occur in production use. Hence, profiling results obtained using Intel VTune shown in Figure 3 were collected from simulations of a realistic and operational harbor wave resonance study done at the St. George Harbor in Alaska. Figure 3 shows that of the total runtime, approximately a quarter of the runtime was spent in MPI communications. In addition, from the time attributed to the MPI communications, over half was due to an MPI imbalance (i.e., over half of the MPI communications are stalled because they are waiting on other ranks to complete their work). In essence, for approximately 14% of the total runtime, majority of the ranks are doing nothing.

```
export PERFSNAP=/work1/compiler-beta/vtune_amplifier_2018.0.0.507509/bin64/aps.sh
export MPS_VERBOSE_MODE=1
export MPS_VTUNE_RESULTS_PATH=$conical_island/myDir3
mkdir -p $MPS_VTUNE_RESULTS_PATH
EXEC=$FUNbin/funwave.x
mpirun -n 36 $PERFSNAP ${EXEC} input
```

Figure 2. Example of how to use Intel VTune Performance Snapshot inside the PBS script.

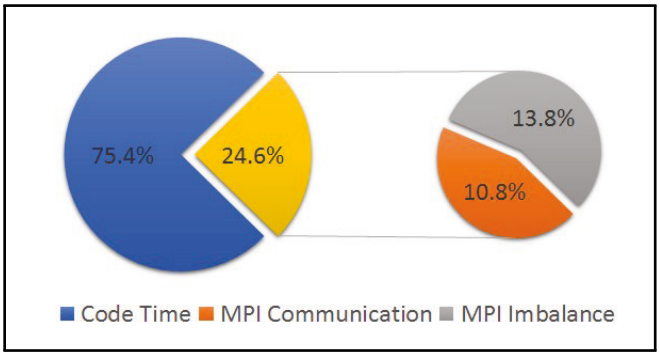

Figure 3. Intel VTune Performance Snapshot results for St. George Harbor, Alaska simulation. The yellow portion is the time attributed to MPI communications.

Intel VTune also shows that approximately 44% of the inefficiencies in the FUNWAVE-TVD are due to back-end stalls, which cause under utilization of the CPU's pipeline slots (i.e., pathways to execute instructions remain empty, causing code inefficiencies). Back-end stalls may result from

- too many divisions, which are more expensive than multiplications
- evaluation of complex mathematical functions (e.g., Log, or Exponentials)
- inefficient nesting of loops (poor memory utilization).

While evaluations of divisions and complex mathematical functions are unavoidable for some codes, excessive evaluations may be avoided in some scenarios by (1) pre-computing function evaluations that do not change throughout the simulation, for example:

$$f[i] = x[i] \times \log(c) \quad \Rightarrow \quad f[i] = x[i] \times d \quad \text{where} \quad d = \log(c) \quad \text{for} \quad i = 1, ..., N ,$$

where c is constant for all i's and d is precomputed or (2) factoring out common terms, for example:

$$\frac{a}{d} + \frac{b}{d} + \frac{c}{d} \Rightarrow \frac{a+b+c}{d} \tag{1}$$

Note that while a compiler should normally perform the optimization shown in (1), in practice, the individual terms may not be as simple as the preceding example and should be addressed by the developer/programmer. Furthermore, the terms may be computed in different subroutines, and the compiler may not recognize the common factor. As a slight aside, it can be a good programming/compiling practice to use aggressive compiler optimization level such as (-O3) to initiate as many compiler-induced improvements as possible.

Poor Memory Access. Poor memory access may be a result of inefficient cache memory usage. Cache memory is an additional layer of memory on top of the random access memory (RAM) located on the CPU and can be up to several orders faster than the RAM. However, in contrast to RAM, its storage capacity is more than several orders smaller. Typically in modern CPUs, there are three levels of memory (Figure 4) where higher levels have less storage but faster access speeds. Efficient use of cache memory is important in fast codes. Currently in FUNWAVE-TVD, approximately one-third of all requests to cache are not met (cache misses). To discuss how cache misses may arise, an explanation of how the data are stored in memory and how looping through data can lead to cache misses follows.

A two-dimensional (2D) grid of data points is often viewed physically as a matrix. In computer memory, however, data is arranged as a contiguous one-dimensional vector. In Fortran, data are stored using column-major ordering (i.e., each column of data is stored in succession, see Figure 5). In other programming languages, data may be stored using row-major ordering (e.g., C and C++).

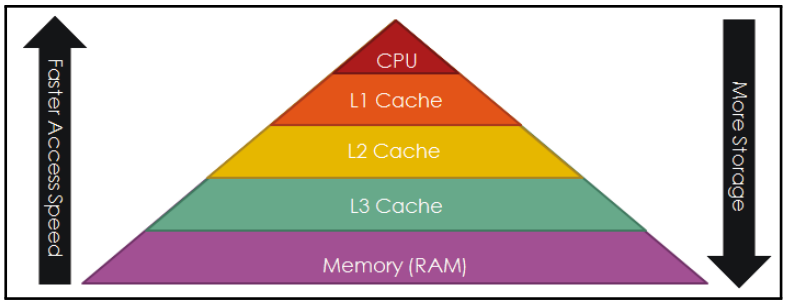

Figure 4. An diagram of the memory hierarchy in a modern computer.

Figure 5. A comparison between how 2D data are viewed as a matrix (left) and how the 2D data are arranged in memory in column-major Fortran (right).

When looping through 2D data, there are two possible ways of nesting the loop: the inner loop (fastest changing index) cycles through the column index (see pseudo-code in Figure 6(a)), or the inner loop cycles through the row index (see pseudo-code in Figure 6(b)). However, the choice of how the loops are nested greatly impacts the cache utilization. For illustrative purposes, it is assumed that there is one level of cache memory that fits one column (or row) of data, and data are column-major ordered in memory (as for Fortran codes like FUNWAVE-TVD). When the CPU requests the first element in the data, the element is retrieved from RAM, and the column of data containing the element is stored in the cache memory (Figure 7(a)). If the inner loop cycles through the column index, the next column element is not in the cache, thus is retrieved from the slower RAM (cache miss) (Figure 7(b)). Alternatively, if the inner loop cycles through the row index, the next row element is in the cache; thus it is retrieved from the faster cache memory (cache hit) (Figure 7(c)). The same access patterns occur for successive row (or column) elements. Therefore, a simple switch in the order in which loops are nested can drastically affect the performance of codes.

```
a) LOOP i = 1,N
       LOOP j = 1,M
           ACCESS DATA(i,j)
       END LOOP
   END LOOP
```

```
b) LOOP j = 1,M
       LOOP i = 1,N
           ACCESS DATA(i,j)
       END LOOP
   END LOOP
```

Figure 6. Examples of simple nested loops for accessing 2D data where a) the inner loop cycles through the column index and b) the inner loop cycles through the row index.

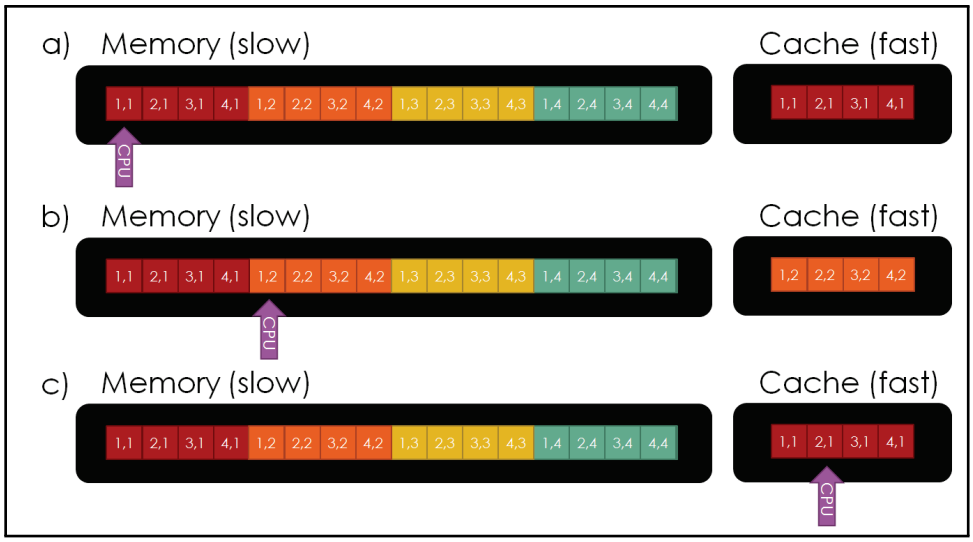

Figure 7. A simple example of how the CPU accesses memory types and how data are loaded into the cache memory. a) CPU requests the first entry in the 2D data, which loads the first column into the cache. b) CPU requests the next column index for a fixed row index (see the pseudo-code in Figure 6(a)), which is not in the cache. c) CPU requests the next row index for a fixed column index (see the pseudo-code in Figure 6(b)), which is in the cache.

The same analysis extends to higher dimensional arrays. In general, in column-major programming languages, to efficiently loop the array, data$[x_1, x_2, \ldots x_n]$, loops must be nested starting with the last index being the outer most loop and iterating the nesting by cycling backward through the indices, ending with the first index being the inner most loop. In row-major languages, the nesting order is reversed.

Vectorization, or array programming, may be used instead of nested loops to express operations on multi-dimensional arrays/matrices. In array programming capable languages, vectorization eliminates the need for loops and allows the compiler to improve code performance by implementing single instruction, multiple data (SIMD) operations, which efficiently use the cache memory and saturate the CPU pipelines. However, in some cases, vectorization may not be possible (e.g., boundary conditions where instructions may differ from the bulk of the domain). Note that some compilers may implement SIMD operations with explicit loops on their own, but that is not always guaranteed.

Open|Speed Shop. Open|Speed Shop is another profiling tool that may be used to get an in-depth analysis of code inefficiencies. One quantity that may be computed to estimate the efficiency of code is the SIMD vs. double precision operations ratio (SIMD/DP OPS). An example of computing this quantity in FUNWAVE with Open|SpeedShop is shown in Figure 8.

```
osshwcsamp "mpiexec_mpt -np 36 $EXEC input" SIMD_FP_256,PAPI_DP_OPS
openss -cli funwave.x-hwcsamp.openss
```

Figure 8. Example of how to use the Open|Speed Shop SIMD analysis for FUNWAVE.

Table 2 lists the SIMD/DP_OPS ratio for the top-four most expensive subroutines and the average for the entire simulation. The data show that on average the SIMD/DP-OPS is approximately 0.175, which is approximately 70% of the ideal value, 0.25, indicating that FUNWAVE can benefit from more vectorization.

Table 2. Single instruction, multiple data vs. double precision operations for FUNWAVE-TVD.

Function	Time (s)	Total	# SIMD	DP OPS	SIMD/DP-OPS
construct_ho_x	596	15 %	1.17×10^{11}	1.17×10^{11}	0.17
construct_ho_y	498	13 %	9.15×10^{10}	1.17×10^{11}	0.18
cal_dispersion	383	10 %	6.72×10^{10}	1.17×10^{11}	0.18
construction_ho	120	3 %	2.33×10^{10}	1.17×10^{11}	0.17
total	2457		4.74×10^{11}	1.17×10^{11}	0.175

As mentioned before, approximately one-third of requests to the cache are not met (indicated by L2 RATIO and L3 RATIO being approximately 0.3). An example of using Open|Speed Shop to compute cache misses is shown in Figure 9, the results of which are shown in Table 3 for the three most expensive subroutines in FUNWAVE-TVD.

```
osshwcsamp "mpiexec_mpt -np 36 $EXEC input" PAPI_L2_DCA,PAPI_L2_DCM
openss -cli funwave.x-hwcsamp.openss

osshwcsamp "mpiexec_mpt -np 36 $EXEC input" PAPI_L3_ TCA,PAPI_L3_ TCM
openss -cli funwave.x-hwcsamp.openss
```

Figure 9. Example of how to use the Open|Speed Shop cache-miss analysis for L2 (top) and L3 (bottom) caches.

Table 3. L2 and L3 cache misses for FUNWAVE-TVD.

Function	L2 Access	L2 Misses	L2 Ratio	L3 Access	L3 Misses	L3 Ratio
construct_ho_x	3.1 × 1010	9.9 × 1010	0.32	1.0 × 1010	3.9 × 109	0.37
construct_ho_y	2.6 × 1010	8.6 × 1010	0.33	8.8 × 109	3.0 × 109	0.3
cal_dispersion	2.4 × 1010	7.1 × 1010	0.30	7.2 × 109	2.0 × 109	0.28

By using Open|Speed Shop, an inefficiency in how loops are nested (or lack of vectorization) was identified that resulted in a considerable number of cache misses. Furthermore, by determining which subroutines are the most expensive, an efficient plan may be devised to improve the overall performance of FUNWAVE-TVD.

Load Imbalance. Load imbalance can arise in MPI codes when work is improperly distributed amongst ranks. As mentioned before, Intel VTune showed that more than half of the time spent in MPI communication routines/tasks are stalled due to MPI imbalances (e.g., ranks are waiting on data to be processed by other ranks before continuing).

Table 4 lists the top-three MPI subroutines in FUNWAVE-TVD, where most communication time is being spent. Note that all of those three MPI functions constitute waiting for other ranks. The results show that approximately 90% of the MPI time involves waiting. Furthermore, the results show that the maximum time a rank spends in one of these subroutines is much larger than the minimum and average, indicating that there is a load imbalance. An example of using Open|Speed Shop's MPI analysis is shown in Figure 10.

Table 4. Load imbalance.

FUNCTION	TOTAL (MS)	% MPI TIME	# CALLS	MIN (MS)	MAX (MS)	AVG (MS)
MPI_Waitall	476389	34.2	17112096	0.0	30.7	0.0
MPI_Barrier	395200	28.4	198432	0.0	340.8	2.0
MPI_Wait	380528	27.4	1357920	0.0	5.8	0.3

```
ossmpi "mpiexec_mpt -np 36 $EXEC input"
openss -cli funwave.x-mpi.openss
```

```
ossio "mpiexec_mpt -np 36 $EXEC input"
openss -cli funwave.x-io.openss
```

Figure 10. Example of how to use the Open|Speed Shop MPI analysis (left) and I/O analysis (right).

To get a better understanding of where the MPI imbalances arise in FUNWAVE-TVD, a cluster analysis may be performed to identify outlying ranks (e.g., producing a graph similar to Figure 1 using Vampir). However, this technical note leaves this for future work and focuses on the MPI imbalance due to I/O of data.

I/O Analysis. I/O analysis with Open|Speed Shop shows that 95% of the I/O tasks is based on writing to a file, which is expected as reading data from files only occurs when initializing FUNWAVE-TVD. See Figure 10 for an example code for performing the I/O analysis and Table 5 for more detailed results.

Table 5. Load imbalance with I/O.

Function	Time (MS)	% Total	# Calls
write	177664	95	27409
read	7874	4	6874
open64	1950	1	521

The MPI imbalance related to I/O in FUNWAVE-TVD is due to serial writing/output to a file. The data to be written/outputted to a file are first gathered at the root rank (typically rank 0) and are subsequently written to a disk/file. In the meantime, as the root rank writes data to a file, all other ranks must wait until the root rank finishes, thereby causing an MPI imbalance.

Alternatively, parallel I/O may be implemented to use all ranks to write to a file collectively and concurrently. In this mode, each rank is assigned its corresponding section of the file, which normally depends on which subsection of the domain the rank was assigned to do its work. Then, each rank will write its data to the file without interfering with one another. While there is an overhead in coordinating the I/O, the benefit of using all ranks in parallel I/O outweighs the overhead of collecting data to one rank and the load imbalance arising from the other ranks waiting (see Figure 11 for a sample comparison between load balancing in serial I/O and parallel I/O). Note that with the parallel I/O implementation, wait time is virtually eliminated for this example

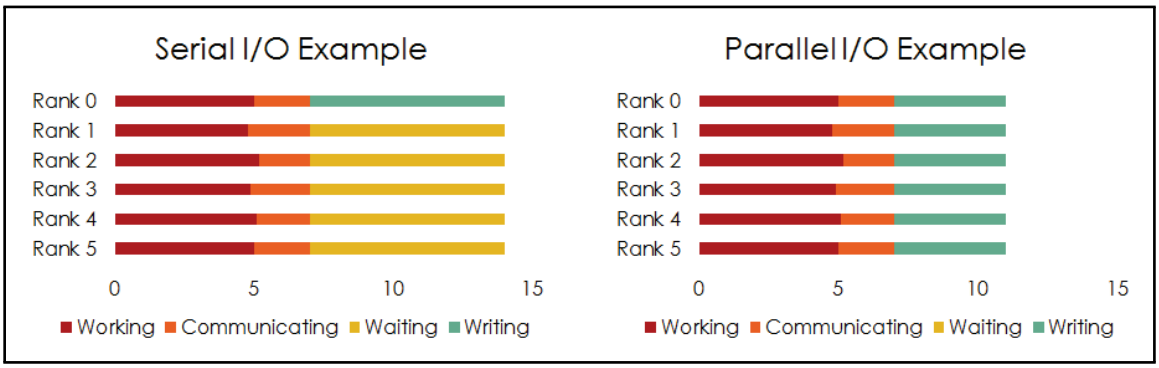

Figure 11. Serial (one core) and parallel (all cores) I/O comparison example.

Reducing the output file size can also improve the performance of codes, as lowering the output file size will obviously lower the required I/O time needed to write the data to a file. Currently, FUNWAVE-TVD outputs data in a user-readable text format (ASCII). However, outputting the data in a native binary format reduces the amount of information to be transferred and the required binary to ASCII conversion.

Alternatively, self-describing data formats such as Network Common Data Form (NetCDF), Hierarchical Data Format (HDF4/HDF5), and eXtensible Model Data Format (XDMF) use binary storage, thereby effectively reducing the file size. Furthermore, these formats include additional (self-describing) information associated with the data (e.g., partition sizes, simulation parameters, grid type) eliminating the need to include a secondary file with this information.

PARALLEL AND BINARY I/O IMPLEMENTATION RESULTS: Results from extending FUNWAVE-TVD with parallel I/O are discussed, in particular, implementing parallel I/O for outputting 2D field data such as the surface elevation, velocity fields, significant wave height, etc. In addition, parallel I/O supports binary, NetCDF, and XDMF formats. As mentioned earlier, not all users have access to sophisticated HPC systems such as Topaz. Therefore, results are also presented from simulations on a Linux laptop. The main improvements with the new I/O are as follows:

- Binary file size is 50% smaller than text output (ASCII). Similar improvements for XDMF and NetCDF
- 10% – 35% speed up in runtime
  - St. George Harbor, AK (762 × 552 points): 10%-15% speedup on Topaz (144–288 ranks) and a Linux laptop (2–4 ranks)
  - Marina di Carrara Harbor, Italy (1512 × 1152 points): 30%–35% speedup on Topaz.

What was immediately observed was that as the domain size increased, the computational gains improved (e.g., Marina di Carrara Harbor). This is the common case of bandwidth versus latency interplay. Larger domains take better advantage of bandwidth whereas smaller domains are quickly subjected to latency. As with any distributed HPC system, there are multiple factors that affect computation and communication among the ranks. For example, if a particular simulation requires very frequent output of field data to carry out a time-series analysis, or perhaps to generate smooth animations, the constant access and frequency of writing to a disk from memory will adversely affect the total computational time. The computational gains listed above are just an illustration of what was seen on the HPC machines and a local Linux 8-core laptop. The types of cable interconnects and their respective bandwidth, along with sophistication in system/node interconnecting network architecture, will inevitably deliver different results across different HPC platforms. The relatively simple change of implementing parallel and binary I/O in FUNWAVE-TVD, however, has already delivered 30% or more of speed up. This computational gain becomes even more apparent when one has to run multiple scenarios across multiple storm/field and wave conditions.

SUMMARY AND FUTURE WORK: Profiling and optimization of large operational and in particular parallel processor codes can seem intimidating. It can, however, provide basic guidance on how to understand the performance of a code and help the programmer plan additional development or optimization work. Many important but very difficult-to-spot issues can surface from performing basic code profiling. As outlined in profiling of FUNWAVE-TVD, the ability to find and correct I/O bottlenecks and cache misses can greatly enhance operational utility of a computational model. Hence, profiling followed by tuning and optimization work should be an integral part of a code/model development effort.

Going forward, FUNWAVE-TVD could benefit from additional profiling and optimization, and the following suggested work should be explored:

- Add support for standardized output formats like NetCDF and XDMF.
- Carry out additional profiling of FUNWAVE with new field output types (e.g., XDMF, NetCDF).
- Investigate how new modules and capabilities in versions 3.1 and 3.2 (e.g., random division of work among ranks, ship wake modules) utilize the distributed HPC resources.
- Develop, implement, and profile how using auxiliary rank(s) to write to a file helps with parallel I/O. POP has shown that, for example, for Electron-Phono Wannier code, University of Oxford, using 1 auxiliary rank instead of all 480 to concurrently write to a file, has led to a 450x improvement in writing data to a file (Gibson 2017).

ADDITIONAL INFORMATION: This CHETN was supported in part by an appointment to the Student Research Participation Program at ERDC, Coastal and Hydraulics Laboratory (CHL), administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the U.S. Department of Energy and EDRC. This CHETN is a product of the Flood and Coastal Systems Research Program being executed by the ERDC/CHL.

Questions about this technical note can be addressed to Dr. Matt Malej (Voice: 601-634-3742; email: *Matt.Malej@usace.army.mil*). For information about the Flood and Coastal Systems Research Program, please contact Dr. Julie A. Rosati (Voice: 202-761-1850; email: *Julie.D.Rosati@usace.army.mil*) or Mary A. Cialone (Voice: 601-634-2139; email: *Mary.A.Cialone@usace.army.mil*). This technical note should be cited as follows:

> Lam, M. Y.-.H., M. Malej, F. Shi, and K. Ghosh. 2018. *Profiling and Optimization of FUNWAVE-TVD on High Performance Computing (HPC) Machines*. ERDC/CHL CHETN-I-95. Vicksburg, MS: U.S. Army Engineer Research and Development Center. *http://dx.doi.org/10.21079/11681/30037*

REFERENCES

Gibson, J. 2017. *How to Improve the Performance of Parallel Codes*. Performance Optimisation and Productivity Webinar.

Kirby, J. T., G. Wei, Q. Chen, A. B. Kennedy, and R. A. Dalrymple. 1998. *FUNWAVE 1.0. Fully Nonlinear Boussinesq Wave Model*. Center for Applied Coastal Research, Department of Civil and Environmental Engineering, University of Delaware.

Malej, M., J. M. Smith, and G. Salgado-Dominguez. 2015. *Introduction to Phase-Resolving Wave Modeling with FUNWAVE*. ERDC/CHL CHETN-I-87. Vicksburg, MS: U.S. Army Research and Development Center.

Schulz, M. J. Green, D. Montoya, D. Maghrak, J. Galarowicz, and T. Thomas. 2016. *How to Analyze the Performance of Parallel Codes 101 A Case Study with Open—SpeedShop*. International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, Utah.

Shi, F., J. T. Kirby, B. Tehranirad, J. Harris, Y.-K. Choi, and M. Malej. 2016. *FUNWAVE-TVD: Fully Nonlinear Boussinesq Wave Model with TVD Solver*. Center for Applied Coastal Research, Ocean Engineering Laboratory, University of Delaware.

Ware, W. H. 1972. "The Ultimate Computer." *IEEE Spectrum* 9(3): 84–91.