Towards Improving the Performance of the ADCIRC Storm Surge Modeling Software

by:

Nicholas Weidner
Department of Computer Science
University of South Carolina
Columbia, SC, 29208, USA

Tim Stitt
Center for Research Computing
University of Notre Dame
Notre Dame, IN, 46556, USA

Original Submission: October 15, 2013
Revised Submission: October 24, 2014

Caravel
University South Carolina
Undergraduate Research Journal

Abstract

Accurately predicting storms and hurricanes is critical to saving lives and reducing economic loss. Therefore, it is necessary to use the most efficient software and hardware technology available in order to improve the performance and fidelity of these predictive mathematical models. For over ten years, the Computational Hydraulics Lab (CHL) at the University of Notre Dame has been involved in developing the high-resolution ADvanced CIRCulation (ADCIRC) storm surge model to predict storm surges in coastal areas. The objective of the work reported here was to port a novel adaption of the parallel ADCIRC code to the state-of-the-art Intel Xeon Phi Co-Processor system (Stampede) at the Texas Advanced Computing Center (TACC) and ideally demonstrate speedup on a set of benchmark calculations. The porting process was accomplished by identifying fine-grained parallelism and vectorizable compute-intensive loops that could be offloaded to the 61-core Xeon Phi co-processors while leveraging their 512-bit wide vector units. Due to transfer latencies, offloading excessive amounts of code can reduce the effectiveness of using the Xeon Phi co-processors so the code was initially profiled in order to identify the code hotspots where the host processors spent the majority of their time. This analysis allowed us to focus the work of OpenMP and Xeon-Phi offloading on the most expensive routines. These routines were modified to take advantage of the full 16 threads available on the host processor, and will ultimately allow us to offload work to the increased number or threads on the partnered Xeon-Phis.

Introduction

Hurricanes are disastrous to coastal regions all across the world. They are especially dangerous if the storm is large, and the amount of preparation for the storm is small. Because of this, it is important to predict the actions of these storms before they happen, and evaluate the damages done by previous storms in order to prepare for ones still to come. The generalize goal of storm-surge modeling simulations is to better prepare individuals, governments and disaster-relieve agencies for storms in order to save lives and reduce property damages [1-5]. For example, hurricane Sandy, which measured 820 miles in diameter, hit New Jersey in October of 2012 and caused an estimated $25 billion in lost business activity, and left 8.1 million homes without power [6]. It is important for residents, agencies and municipalities to ready themselves for the storm to minimize the damages and negative impacts on coastal regions. Performing the appropriate calculations involves a large amount of computational power and time. For models to be effective and return timely, accurate results, they need to take advantage of the parallel computing power of super computers.

The high-resolution, ADvanced CIRCulation (ADCIRC) storm-surge model collaboratively developed at the Computational Hydraulics Lab (CHL) at the University of Notre Dame is an excellent example of highly parallel-processing code that is run on high-performance computers [4-5]. The ADCIRC model has been and continues to be the standard coastal model utilized by US Army Corps of Engineers (USACE) and Federal Emergency Management Agency (FEMA) [7]. It has been used, for example, to help reconstruct the levees in New Orleans after Katrina to prevent a future disaster of that magnitude [7,8]. Recent improvements to the ADCIRC code has changed the underlying numerical approach for calculating the results [4] This new numerical approach should provide more accurate results and stable performance.

However, this increased fidelity could come at the expense of computational speed if the new code does not take advantage of the parallelism inherent in this new numerical approach.

The objective of the work reported here was to port the new parallel ADCIRC storm-surge code (referred to here as DGSWEM) to the state-of-the-art Intel Xeon Phi Co-Processor system (Stampede) at the Texas Advanced Computing Center (TACC) with the goal of showing increased performance on a set of benchmark calculations. The code was initially profiled in order to identify code hotspots, which allowed us to focus co-processor offloading to regions that could exploit the parallel characteristics of the Xeon-Phi co-processor and hence provide the most benefit.

Background

Parallel processing has been critical to the evolution of high-performance computing systems for the last four decades. Originally, parallel processing involved multiple processors sharing a single pool of memory [9]. Issues with coordinating and scaling multiple processors within a single memory address space led to the advent of distributed memory systems, where each processor has its own local memory. This greatly improved performance, but it led to communication challenges among the individual processor-memory pairs. More recently, hybrid architectures have been developed where each node is a multi-core shared memory computer connected with a high-speed network for message-passing. A schematic of this configuration is shown in Figure 1a, where each node is a multi-core shared-memory computer with communication between the nodes. One drawback of this architecture is that each core sees only the memory on its own node with remote memory data access requiring specific messaging between nodes, which introduces significant communication latency and bandwidth overhead to the calculations. One solution to this drawback is to leverage local accelerator and co-processor

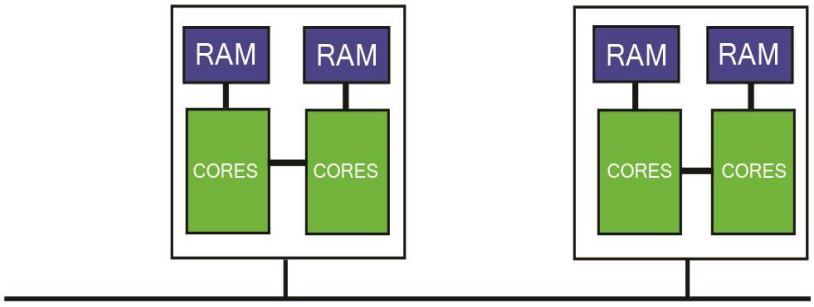

Figure 1a: A schematic showing two nodes in a hybrid computer architecture. Each node is a multi-core shared-memory computer with communication between the nodes.

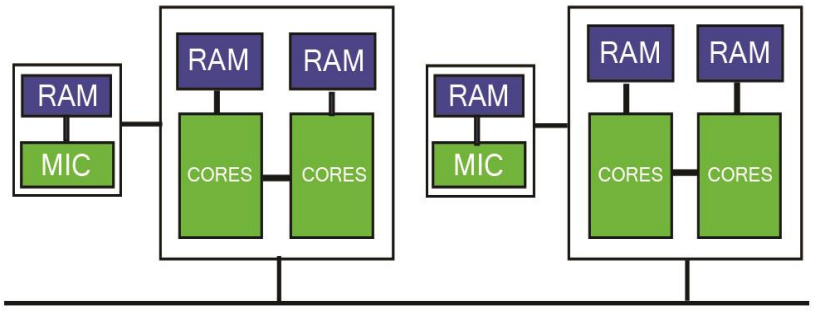

Figure 1b: A schematic of the architecture used in Stampede's Xeon Phi MICs (Many Integrated Core) co-processor.

hardware to offload highly parallel workloads and avoid expensive network communication. Intel has recently released its MIC (Many Integrated Core) co-processor architecture (see Figure 1b), which can greatly improve computing performance [10].

Although there are many advances to the various forms of parallel processing, it is still a difficult undertaking, Dividing up processing tasks and communicating between them introduces additional overhead, and if this time outweighs the amount of CPU speedup, then there is no wall-clock speedup [11]. In addition, if the simulation tasks are divided up between processors but one processor cannot start its task until another processor finishes, then no net speed up in

wall-clock speed is achieved. The goal therefore is to identify program tasks that multiple processes can run independently, while minimizing the time processes spend communicating. When done correctly, the wall-clock time needed for the resulting code to run on a parallel-processing machine can be reduced by orders of magnitude. That is, a well written parallel program should result in strong scaling, meaning there is speedup achieved on a multi-processor system without increasing the size of the problem at hand. Accelerators and co-processors can push parallel processing even farther by providing a many-core architecture that can manage hundreds or thousands of threads of execution simultaneously.

The new numerical approach being implemented by the DGSWEM code provides more accurate results and stable performance. However, this increased fidelity could come at the expense of computational speed if the new code does not take advantage of the parallelism inherent in this new numerical approach. The code divides up the region being simulated into a grid of user defined cells. It uses a message-passing interface (MPI) to communicate between these cells where each cell is given a processor to do calculations for it. The simulation equations are done independently for each cell on each processor and the MPI communicates any necessary data to its neighboring cells when needed. This is why large scale parallel computing is so important in this kind of simulation.

Large storm surge simulations are routinely run on the host processors of the large Stampede system at TACC. Stampede is a 6,400 compute node system, where each node has two Intel Sandy Bridge processors paired with two Intel Xeon-Phi co-processors. Each node is configured with 32GB of memory with an additional 8GB of memory on each Xeon-Phi [10]. The machine operates on a Linux kernel and all of our development was performed using Intel Compiler v14.

Results and Discussion

Since the size of the source code is so large, it was necessary to narrow down which routines would benefit most from speed up, and which segments had small enough execution times to remain running as they had been. Profiling allowed us to evaluate how much execution time was spent in each individual routine. Profiling was done using the University of Oregon's Tuning and Analysis Utilities (TAU) [12], and served two primary purposes. First we calculated the time spent in each individual source routine, for a set of benchmark runs, in order to determine which routines contributed most to the overall runtime. Secondly we profiled test runs at different processor counts in order to ensure that these hotspots remained significant as we increased the processor count. If this was not the case, then we would want to reevaluate where to focus our improvements since most realistic simulations are performed at high processor counts (typically 2000 – 16,000 cores). All of these runs were performed solely on the host processor and only wall-clock times were calculated.

Figure 2 shows a 128 core run of the DGSWEM code and displays individual wall-clock times for each routine on each core. What we see are a few routines in the program take a

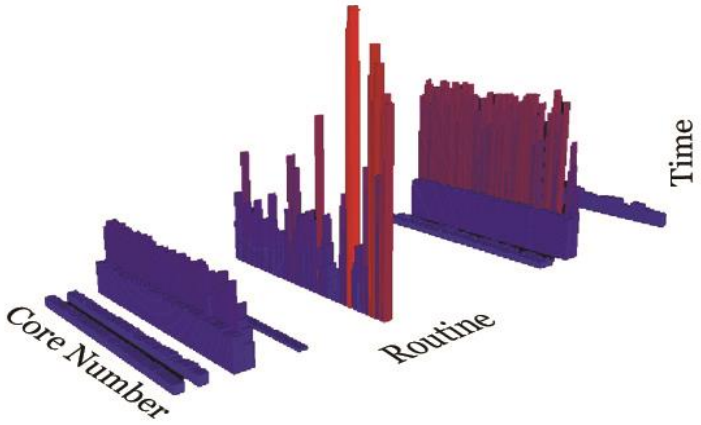

Figure 2: Example of output from a 128 core run of the DGSWEM code. Individual wall-clock times for each routine on each core is display by the relative heights of the bars.

significantly longer time to run than do others. Identifying the routines with the longest run times can be seen more easily by replotting Figure 2 for just three sets of cores (128, 256, and 512 cores), which is shown in Figure 3. The five longest running routines during the simulation

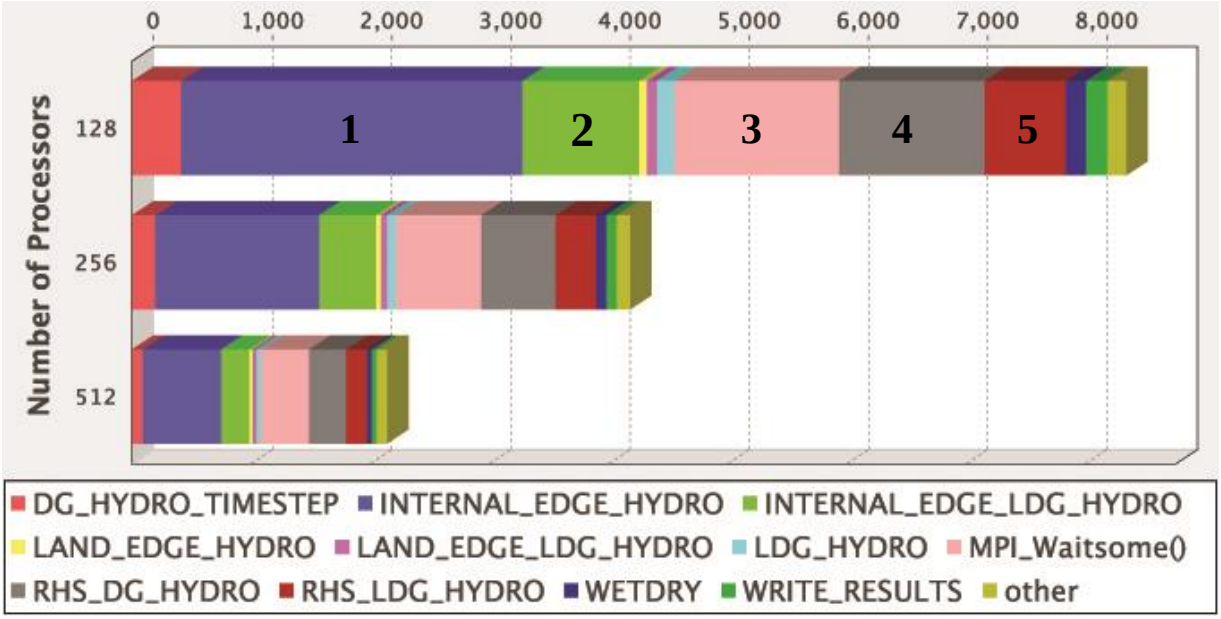

Figure 3: The horizontal bars represent wall-clock times for the various routines in seconds for three separate DGSWEM runs on 128, 256, and 512 cores.

are INTERNAL_EDGE_HYDRO (1), INTERNAL_EDGE_LDG_HYDRO (2), MPI_Waitsome (3), RHS_DG_HYDRO (4), and RHS_LDG_HYDRO (5). MPI_Waitsome (routine 3) is actually a routine designated for handling the MPI between the processors. Although improvements can be made in this area as well, this was outside the scope of this research. Therefore, we focused our offloading efforts on the other 4 routines. We also see from Figure 3 that the relative speed of each routine was constant as the core count increased. For example, the 4 routines (1, 2, 4 & 5) account for approximately 70% of the overall run time at 128, 256 and 512 cores. Therefore, speeding up these routines should have a significant impact on the speed up of the overall run. This figure also demonstrates the scalability of the code. We can see that doubling the core count

appears to cut the time spent on the entire run almost in half. This showcases just how well the code is already optimized for regular host-processor runs and how important it is to sustain this scalability as we move to DGSWEM.

Figure 4 further demonstrates the scalability of the existing code, and how far it needs to be sustained. We can see that the effectiveness of increasing the core count does not fall off until around ten thousand cores (for the two larger EC2001 grids, x 4 and x16).

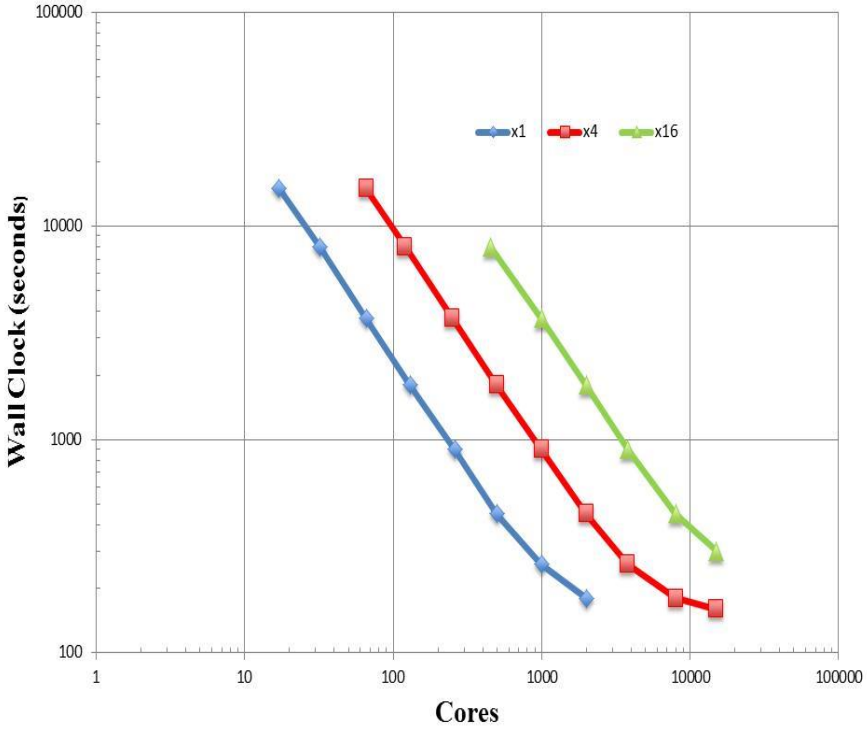

Figure 4: Each line represents a different sized grid run (x1, x4 and x16) on a number of different core counts in order to evaluate wall-clock time.

With our routines chosen, we began implementing both OpenMP and offloading the designated sections. OpenMP was implemented by designating which data values in each routine were either shared or private to the individual thread. This meant classifying each individual variable used in each routine. We mentioned earlier that dividing up tasks worked best on code that was vectorized. The problem is that the Internal_Edge_Hydro (routine 1), our largest

routine, was not completely vectorized. Because this routine contains shared data values that are read/written from multiple threads, statements were added to the code to prevent different threads from having the same value.

Figure 5 shows the run time results of the code with OpenMP implemented over a number of threads. At 128 processors the original non OpenMP DGSWEM runs faster than an OpenMP DGSWEM with only a single thread implemented. This is due to the overhead of the atomic statements. However, this overhead is overcome once we have at least 2 threads implemented, and by 16 threads DGSWEM runs 4 times faster than DGSWEM without OpenMP (i.e., 1899 seconds compared to 7892 seconds). This is still slower than the old ADCIRC with the

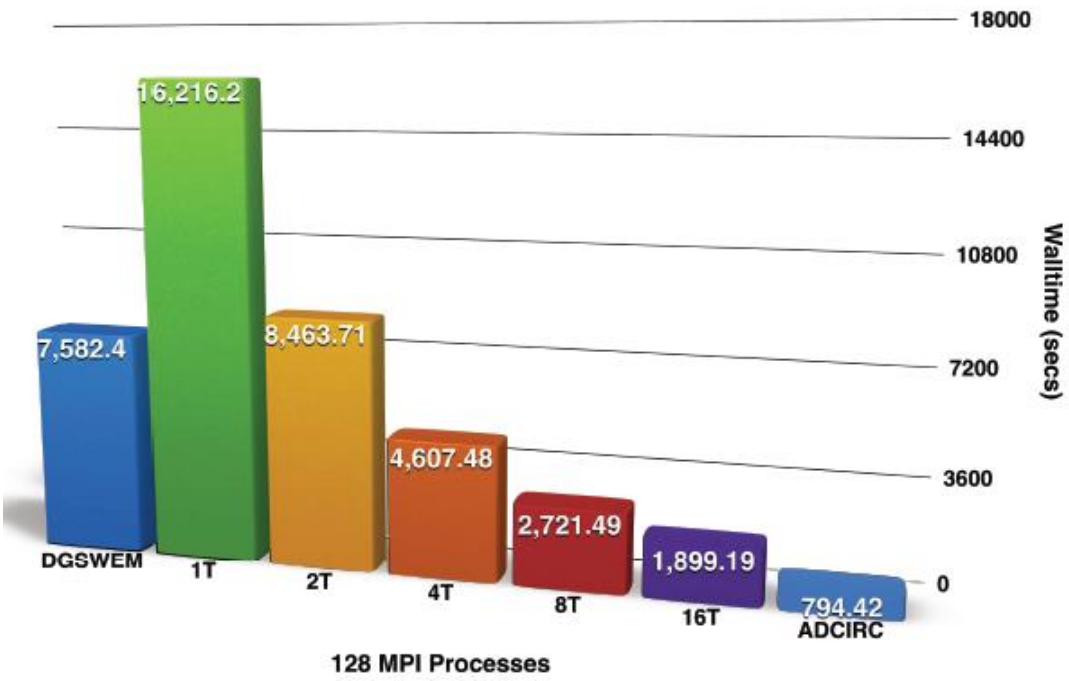

Figure 5: Wall-clock times for the old ADCIRC code compared to the one with the new numerical approach before implementing OpenMP (i.e., DGSWEM) and after with different number of threads. The doubling in time for the OpenMP with one thread (1T) reflects the computational overhead in implementing OpenMP. With two threads (2T), the overhead is essentially eliminated and with sixteen the code is running 4 times faster than the unparalized DGSWEM.

less accurate numerical approach, but we can push the limitation of 16 host processor threads by taking advantage of the additional ones on the partnered Xeon-Phis. Therefor the next step is implementing Xeon-Phi offloading to take advantage of the additional threads.

Conclusions

The ADCIRC storm-surge computer code collaboratively developed at the Computational Hydraulics Lab (CHL) at the University of Notre Dame was initially ported to the state-of-the-art Intel Xeon-Phi Co-Processor system (Stampede) at the Texas Advanced Computing Center (TACC). The porting was attempted by identifying and vectorizing compute-intensive loops to be offloaded to the 61-core Xeon-Phi co-processors. Offloading excessive amounts of code can reduce the effectiveness of using the Xeon Phi co-processors so the code was initially profiled in order to identify the code sections where the host processors spent the majority of their time. This analysis allowed us to focus the work of OpenMP and Xeon-Phi offloading on the most expensive routines. We identified four calculation-heavy routines and implemented OpenMP directives into the code. In order to overcome vectorization problems, additional overhead was introduced with atomic statements. This overhead was overcome though once OpenMP implementation was complete and at least 2 threads were used. Using 16 threads, the new code ran approximately 4 times faster on the host processors than it did without OpenMP. This did not match the performance of the original ADCIRC code in terms of speed, but through the use of Xeon-Phi offloading the new more accurate DGSWEM code will take advantage of the additional threads available on the Xeon-Phi to reach much faster run times with a more accurate and robust numerical approach.

Acknowledgments

We would like to thank the Center for Research Computing at the University of Notre Dame for hosting the Research Experience for Undergraduates program, as well as the National Science Foundation for funding the program. We would also like the thank Notre Dame's Computational Hydraulic Lab for allowing us to work with their codes and for providing assistance with the project when it was needed. Finally, we would like to thank the Undergraduate Researach Office at the University of South Carolina for funding through the Magellan Apprentice program.

References

1. Westerink, J. J., R. A. Luettich, A. M. Baptista, N. W. Scheffner, P. Farrar, "Tide and Storm-Surge Predictions Using Finite-Element Model," *J. Hydraulic Engineering*, **118(10)**, 1373-1390 (1992).
2. Bode, L., and T. A. Hardy, "Progress and Recent Development in Storm Surge Modeling," *J. Hydraulic Engineering*, **123(4)**, 315-331 (1994).
3. Hubbert, G. D., and K. L. McInnes, "A Storm Surge Inundation Model for Coastal Planning and Impact Studies," *J. Coastal Research,* **15(1)**, 168-185 (1999).
4. Dawson, C., E. J. Kubatko, J. J. Westerink, C. Tranhan, C. Mirabito, C. Michoski, N. Panda, "Discontinuous Galerkin Methods for Modeling Hurricane Storm Surge," *Advances in Water Resources,* DOI 10.1016/j.advwatres.2010.11.004, **34**, 1165-1176 (2011).
5. Tanaka, S., S. Bunya, J. J. Westerink, C. Dawson, R. A. Luettich, "Scalability of an Unstructured Grid Continous Galerkin Based Hurricane Storm Surge Model," *J.*

Scientific Computing, **46**, 329-358 (2011).

6. Webley, K., "Hurricane Sandy By the Number: A Superstorm's Statistics, One Month Later," (November 26, 2012), Retrieved from: http://nation.time.com/2012/11/26/hurricane-sandy-one-month-later/.
7. "Flood Insurance Study: Southeastern Parishes, Louisiana, Intermediate Submission 2: Offshore Water Levels and Waves," FEMA, US Army Corps of Engineers, New Orleans District, July 24, 2008.
8. Link, L. E., J. J. Jaeger, J. Stevenson, W. Stroupe, R. L. Mosher, D. Martin, J. K. Garster, D. B. Zilkoski, B. A. Ebersole, J. J. Westerink, D. T. Resio, R.G. Dean, M. K. Sharp, R.S. Steedman, J. M. Duncan, B. L. Moentenich, B. Howard, J. Harris, S. Fitzgerald, D. Moser, P. Canning, J. Foster, B. Muller, "Performance Evaluation of the New Orleans and Southeast Louisiana Hurricane Protection System," Volume I – Executive Summary and Overview, Draft Final Report of the Interagency Performance Task Force, U.S. Army Corps of Engineers, Washington, D.C., June 2008.
9. Patterson, D. A., and J. L. Hennessy, "Computer Organization and Design," Fourth Edition, Elsevier, 2012.
10. Texas Advanced Computer Center, Retrieved from: http://www.tacc.utexas.edu/stampede/
11. James, D., "Introduction to Parallel Computing," Texas Advanced Computer Center, (September, 2013) Retrieved from: http://www.tacc.utexas.edu/user-services/training/course-materials
12. Shende S., and A. D. Malony, "The TAU Parallel Performance System," *Int. J. High Perform. C.*, **20(2)**, 287-331 (2006).