A Hybrid Double-Layer Master-Slave Model
For Multicore-Node Clusters

Gang Liu, Hartmut Schmider and Kenneth E. Edgecombe
High Performance Computing Virtual Laboratory (HPCVL),
Queen’s University, 115-993 Princess St., Kingston, Ontario, K7L 1H3, Canada
E-mail: gang.liu@queensu.ca

Abstract. The Double-Layer Master-Slave Model (DMSM) is a suitable hybrid model for executing a workload that consists of multiple independent tasks of varying length on a cluster consisting of multicore nodes. In this model, groups of individual tasks are first deployed to the cluster nodes through an MPI based Master-Slave model. Then, each group is processed by multiple threads on the node through an OpenMP based All-Slave approach. The lack of thread safety of most MPI libraries has to be addressed by a judicious use of OpenMP critical regions and locks.

The HPCVL DMSM Library implements this model in Fortran and C. It requires a minimum of user input to set up the framework for the model and to define the individual tasks. Optionally, it supports the dynamic distribution of task-related data and the collection of results at runtime. This library is freely available as source code. Here, we outline the working principles of the library and on a few examples demonstrate its capability to efficiently distribute a workload on a distributed-memory cluster with shared-memory nodes.

1. Introduction
Modern clusters usually consist of nodes with multiple CPUs or cores. In addition, recent core architectures support multiple threads being concurrently executed on a single core (hyper-threading). The standard approach to exploit parallelism on a cluster is by use of the Message-Passing Interface MPI. This requires the communication between independent “heavy-weight” processes that are deployed across the nodes in a distributed-memory fashion. While it is possible to run multiple independent MPI processes on a single multicore cluster node, it may not be the most efficient use of resources. This is particularly true for multithreaded cores which exhibit a high degree of resource sharing between threads, and are therefore prone to run into bottlenecks, particularly connected to communication resources.

A more appropriate approach to use multicore and multi-threaded nodes is through the use of thread libraries or compiler directives such as OpenMP. The resulting multithreaded programs do not require communication resources and use shared memory for information exchange between “light-weight” processes instead. As a result, resource-bottlenecks are much less likely. Unfortunately, due to their dependence on the shared-memory architecture of the computer, such programs cannot be deployed across a cluster.

It therefore appears that the ideal approach for programming for a multicore, multi-threaded cluster, is a hybrid one: combining MPI to deploy processes across the cluster to the individual
nodes, with OpenMP parallelization of each process to make efficient use of the multi-thread structure of the hardware.

The main complicating issue with this approach is the lack of thread safety of most MPI libraries. In practice this means that communication resources cannot be accessed by more than one thread per process at a time, because internal communication buffers and other variables are not protected against race conditions. As a result the programmer must ensure that MPI is consistently seeing only one thread per rank and that all information that is communicated is made available to all threads at a time when they need it.

In this paper, we introduce a library implementing a simple model that is applicable in cases when independent tasks have to be performed and no intermittent communication is required, the “Double-Layer Master Slave Model” (DMSM). This model combines a standard Master-Slave approach for the workload distribution among the nodes of a cluster, with a local “All-Slaves” scheme on each of the nodes to work through a group of tasks.

Similar models have been discussed previously under various names. A multi-level Master-Slave model was presented by Chen, Lim, and Warsi to overcome communication overheads on the master [1]. Leopold and Süß used a multi-layered hybrid approach to schedule a Global-Water Simulation Program on a heterogeneous cluster [2, 3]. The scalability of “Bag-of-Tasks” applications was the subject of a systematic study by da Silva and Senger [4]. Another hybrid MPI/OpenMP approach was taken Wu, Lai, and Chiu [5] to address heterogeneities on multicore clusters.

The emphasis of this paper is on the implementation of the DMSM model in the form of a freely available library.

2. The Double-Layer Master-Slave Model

The MPI implementation of the classic Master-Slave Model can readily be expanded to a hybrid model if tasks are combined into “workload packages” and distributed across the nodes. Each of the packages is then processed on a node via an OpenMP All-Slaves Model.

(i) On a cluster, one node serves as Master Node, and thread 0 is dedicated as Master. The other nodes are Slave Nodes. Communication is done via MPI calls among nodes.

(ii) In the beginning, each Slave signals to the Master that it is idle, and the Master sends out workload packages that are handled internally by the Slave nodes.

(iii) Any Slave Node works through a package without any further communication among nodes. Once the workload is executed, another package is requested. This happens until a stop signal is received.

(iv) The Master continues to serve requests until all work is distributed. Then a stop signal is passed to the Slaves. The Master insures that no package is distributed more than once or skipped.

In the above Master-Slave scheme the workload execution on the nodes (Step iii) is in itself done through OpenMP:

(i) Once a new workload package is received on a Slave node, a task counter is reset, and multiple OpenMP threads begin to work on the package.

(ii) Each thread assigns itself to a given task in the package and moves the task counter forward to avoid double execution or skipping.

(iii) The task counter is protected by placing updates into a critical region or by use of locks.

(iv) Once all tasks in a package are executed or at least assigned, a single thread communicates with the Master for a new package.
Since on a given node, memory is shared and any thread can access the task counter, there is no need for a master thread. The resulting model can adequately be described as an All-Slave Model. The details give rise to different variations, but it is essential that only one thread at a time communicates with the master node, as MPI libraries are usually not thread-safe.

3. The HPCVL DMSM Library

This library was developed by Gang Liu since 2009 to enable the deployment of a large number of independent computing tasks with unpredictable execution times across a cluster with multithreaded multicore nodes. The library is implemented in Fortran 90 and C. The current version is 3.1. The library is available as source code, free of charge [6].

3.1. Main interface and model variations

The core part of the library is an interface that allows the user to specify parameters for the DMSM model:

```fortran
SUBROUTINE DMSM_ALL( & ! full DMSM run, including initialization and finalization THREADS_PER_PROCESS, & ! number of OpenMP threads per MPI process (node) JOB_DISTRIBUTION_PLAN, & ! specifies variation of the model (see Table 1) TOTAL_JOBS, & ! total number of tasks to be performed NUM_OF_JOBS_PER_GROUP, & ! number of tasks per package DO_THE_JOB, & ! routine performing one job (required) JOB_GROUP_PREPARATION, & ! routine communicating data for package (optional) RESULT_COLLECTION, & ! routine collecting results from nodes (optional) COLLECTION_ENABLED) & ! controls results collection (optional)
```

The only mandatory user routine is `DO_THE_JOB` with one argument, the task number. This implements the execution of a single task. The other routines are optional and will be discussed in section 3.2.

The parameter `JOB_DISTRIBUTION_PLAN` determines what variation of the DMSM is executed: The first column indicates the argument that is passed to `DMSM_ALL`. The second column describes the workload allocation for threads on the master node other than the master thread. The third column describes which thread on the slave nodes is communicating with the master node. Model 11 is the least flexible: only one thread on the master node is active, and OpenMP parallelism on the slave nodes is completely separate from MPI communication, meaning that whenever a thread runs out of work, a new package will only be sent to the node

| Arg | Other threads on Master | Slave Communication |
|-----|-------------------------|---------------------|
| 11  | none                    | 0 from serial region |
| 12  | none                    | 0 from parallel region |
| 13  | none                    | any from parallel region |
| 21  | pre-allocated           | 0 from serial region |
| 22  | pre-allocated           | 0 from parallel region |
| 23  | pre-allocated           | any from parallel region |
| 31  | dynamic                 | 0 from serial region |
| 32  | dynamic                 | 0 from parallel region |
| 33  | dynamic                 | any from parallel region |

Table 1. Variations of the Model
when all other threads are finished. Model 33 is the most flexible: all threads on the master node work through packages dynamically, and on the slave node, new packages are requested and sent as soon as one thread runs out of work.

3.2. Group preparation, result collection and locks
In many cases, initial data have to be distributed to the nodes before work on each task package can commence. For this the user may specify another routine JOB_GROUP_PREPARATION which is called whenever a new package is requested. To address a race condition that exists when new data are distributed while the previous set is still in use, a pair of routines DMSM_WAIT_FOR_INITIAL_LOCKS and DMSM_UNSET_AN_INITIAL_LOCK are supplied by the library. The locks that are used by these routines are not visible to the user.

In other cases, the results from the individual jobs can be collected to the Master node, either dynamically while executing the DMSM or at the end when all jobs are completed. For this purpose, the user can specify a third routine RESULT_COLLECTION. In addition, an integer parameter COLLECTION_ENABLED specifies whether dynamic collection is desired or not. If the collection happens every time a new task package is requested, another race condition may exist between the communication/reset of the temporary result data structure, and the accumulation of results that are still being generated. To address this, another pair of routines DMSM_SET_NODE_RESULT_LOCK and DMSM_UNSET_NODE_RESULT_LOCK are supplied. As before, the employed locks are not visible to the user.

These routines have to be used inside the user-supplied functions to protect data that are shared among local threads. The details of their usage are discussed in the User’s Manual [7].

3.3. Other usages of the DMSM library
The DMSM library can also be used to run other model. For instance, a double-layer MPI model where the “outer layer” is a Master-Slave model, and the “inner layer” is a more communication intensive MPI parallelisation. Support for this in the library is implemented in the form of additional interfaces and MPI communicators. Likewise the parallelism on the nodes does not have to be in the form of an All-Slaves Model. For example, internal OpenMP parallelization may be implemented in the form of parallel loops.

4. Examples
In the following examples, we used the most flexible “Variation 33” throughout. This means that on the Master node, all threads other than the Master perform dynamically scheduled computations, while on the Slave node the first thread that runs out of work requests a new job package immediately. Job requests are protected through critical regions within the library.

4.1. Workload Balancing
To assess how efficiently the DMSM library balances a workload, we used a series of 851 partial optimizations of the $H_2O_2$ molecule with fixed angles and varying bond lengths. These were done through calls to the computational-chemistry software Gaussian’09 [8] from C-code using the DMSM library. As can be seen in Fig.1, the execution times for these optimizations differs greatly, between 50 and 200 seconds. We deployed the optimizations over 16 Sun T5140 nodes [9] with four serial runs (“threads”) per node using the DMSM. The available multithreading within the Gaussian code was ignored. Fig.2 shows that the execution times per thread for the total run vary between 1000 and 1200 seconds, reducing the workload imbalance from 400% to 20%. This makes the DMSM a suitable means for throughput runs on clusters with high thread-count per node and low per-thread performance.
4.2. Single-node performance on a T4-2
To assess the performance of the DMSM on a single node, we used an Oracle T4-2 multithreaded server [9] that supports up to 128 threads. We ran the brute-force evaluation (by alternating series [10]) of 10,000 points of the Riemann Zeta function $\zeta(z)$ in the interval $2 \geq \text{Re}(z) \geq 1.01$ and $1 \geq \text{Im}(z) \geq 0.01$ in quadruple precision. This is a very slowly convergent series, and the required number of terms to reach a given accuracy depends strongly on the argument. This makes it suitable as a testcase. We call the number of MPI processes $N$, the number of OpenMP threads per MPI process $n$, the overall wallclock time $T_w$, and the accumulated job CPU time $T_j$. The total number of threads is $N_t = Nn$, and the number of slave processes (workers) is $N_s = N_t - 1$ for $N > 1$ and $N_s = N_t$ for $N = 1$.

- The theoretical workload efficiency (percentage of time spent doing work) increases with the number of threads: $N_s/N_t$.
- The real workload efficiency ($T_j/T_w$, Fig.3) is very close to the theoretical one except when many MPI processes are used.
- The scaling efficiency ($T_w[1,1]/(N_sT_w[N,n])$, Fig.4) is close to 100% for up to 16 threads (which is the number of cores), then declines but stays above 50%.

The 64-thread scaling performance of the T4-2 server is in the 60% range despite the 4-fold multithreading and the sharing of one floating-point processing unit among 8 threads. For the DMSM model this means that using 64 threads total on one node still improves performance, with a preference for small numbers of MPI processes. The similarity of the real workload efficiency to the theoretical one indicates that the DMSM is robust as long as not too many MPI processes are used.

4.3. Multi-node performance on a small T5140 Cluster
We used the T5140 cluster (see Gaussian experiment section 4.1) to test the DMSM library with evaluations of the Zeta function across multiple nodes. Running 64 evaluations per package on 64 threads (i.e. 4 per core), we scale up the number of nodes from 1 to 16, for a maximum number of 1024 threads.

The results indicate that a substantial loss of both workload efficiency ($T_j/T_w$, Fig.5) and scaling efficiency ($T_w[1]/(NT_w[N])$, Fig.6) occurs as the number of nodes is increased. This happens whether results are communicated back to the node or not, suggesting that it is the latency of the communication that hampers efficiency, not the size of the messages being passed.
The fact that the decline in workload efficiency is very similar to the scaling one makes it likely that the former causes the latter. As the nodes are forced to communicate new job packages frequently, they spend less time working through jobs. Note that this issue is much less prominent in the single-node case, where communication is done through shared memory with very small latencies. Still, this example cannot be executed through a standard Master-Slave Model because the communication requirements of 1024 MPI processes would be prohibitive, nor through a simple All-Slaves model if no large enough shared-memory machine is available.

**Figure 3.** Workload efficiency for 10,000 functional evaluations of $\zeta$, as a function of process number (MPI, abscissa) and threads per MPI process (OpenMP, legend).

**Figure 4.** Scaling efficiency for 10,000 functional evaluations of $\zeta$, as a function of process number (MPI, abscissa) and threads per MPI process (OpenMP, legend).

**Figure 5.** Workload efficiency for 10,000 functional evaluations of $\zeta$, as a function of the number of nodes (MPI processes). The number of threads on the nodes is kept fixed at 16.

**Figure 6.** Scaling efficiency for 10,000 functional evaluations of $\zeta$, as a function of the number of nodes (MPI processes). The number of threads on the nodes is kept fixed at 16. The single-node run is the reference.
5. Conclusions
The Double-Layer Master Slave Model (DMSM) is well suited to deploy a workload consisting of independent heterogeneous tasks on a cluster consisting of multicore nodes. The usage of a hybrid approach with an OpenMP model on the nodes and an MPI framework across the cluster addresses two issues that may occur in a simple Master-Slave (MS) model and a All-Slaves (AS) approach, respectively. Namely, in an MS model, multiple MPI processes on a multicore node tend to overwhelm the local shared resources, while an AS model can only be deployed on a single shared-memory node.

The HPCVL developed DMSM library implements this model in Fortran 90 and C, and is freely available as source code [6]. It provides a framework to run this model with a minimal required amount of user input but also allows some flexibility concerning the dynamic distribution of task-related data and the collection of results. Most importantly, it addresses the difficult issue of a lack of thread-safety of most MPI implementations by providing the proper lock functions and protecting tasks that are subject to race conditions.

References
[1] Chen, H.C. and Warsi, N.A., An efficient multilevel master-slave model for distributed parallel computation. Boundary Element Technology XIII - Incorporating Computational Methods and Testing for Engineering Integrity, Boundary Elements v. 2, p. 439-448, WIT Press, 1999.
[2] Leopold, C. and Süss, M., Observations on MPI-2 Support for Hybrid Master/Slave Applications in Dynamic and Heterogeneous Environments. Proc. EuroPVM/MPI, LNCS, v.4192, p. 285-292, Springer, 2006.
[3] Leopold, C., Süss, M. and Breitbart, J., Programming for Malleability with Hybrid MPI-2 and OpenMP: Experiences with a Simulation Program for Global Water Prognosis. European Conference on Modelling and Simulation, Bonn, p. 665-670, ECMS, 2006.
[4] da Silva, F.A.B. and Senger, H., Improving scalability of Bag-of-Tasks applications running on master-slave platforms, Parallel Computing v.35, p. 57-71, Elsevier, 2009.
[5] Wu, C.C., Lai, L.F. and Chiu, P.H., Parallel Loop Self-Scheduling for Heterogeneous Cluster Systems with Multi-Core Computers. IEEE Asia-Pacific Services Computing Conference, p. 251-256, IEEE Computer Society, 2008.
[6] The DMSM library can be downloaded at http://www.hpcvl.org/sites/default/files/dmsm/DMSMcode.tar. If the library is used in published work, the present paper should be cited as reference.
[7] Liu, G. and Schmider, H. User’s Manual for the HPCVL DMSM Library, http://www.hpcvl.org/sites/default/files/dmsm/DMSMmanual.pdf, HPCVL, 2011.
[8] Frisch, M.J. et al., Gaussian 09, Revision C.01, Gaussian, Inc., Wallingford CT, 2010.
[9] The hardware used in the examples was a single Oracle T4-2 server with 256GB of memory http://www.oracle.com/us/products/servers-storage/servers/sparc-enterprise/t-series/sparc-t4-2 and a 16-node cluster of Sun Sparc Enterprise T5140 servers with 32GB of memory per node http://www.oracle.com/us/products/servers-storage/036000.pdf.
[10] Gourdon, X. and Sebah, P., Numerical evaluation of the Riemann Zeta-function. http://numbers.computation.free.fr/Constants/Miscellaneous/zetaevaluations.pdf, 2003.