HOMPI: A Hybrid Programming Framework for Expressing and Deploying Task-Based Parallelism*

Vassilios V. Dimakopoulos and Panagiotis E. Hadjidoukas

Department of Computer Science
University of Ioannina, Ioannina, Greece, GR-45110
{dimako,phadjido}@cs.uoi.gr

Abstract. This paper presents HOMPI, a framework for programming and executing task-based parallel applications on clusters of multiprocessors and multi-cores, while providing interoperability with existing programming systems such as MPI and OpenMP. HOMPI facilitates expressing irregular and adaptive master-worker and divide-and-conquer applications avoiding explicit MPI calls. It also allows hybrid shared-memory / message-passing programming, exploiting fully the availability of multiprocessor and multi-core nodes, as it integrates by design with OpenMP; the runtime infrastructure presents a unified substrate that handles local threads and remote tasks seamlessly, allowing both programming flexibility and increased performance opportunities.

Keywords: cluster programming, task-based parallelism, load balancing, MPI

1 Introduction

The pool-of-tasks (or master-worker) paradigm is one of the most widely used paradigms for programming a multitude of applications on a variety of parallel computing platforms. According to this model, the master assigns tasks to a set of workers, providing them with any required input data, and waits for the results. The number of tasks usually exceeds the number of workers and the master may generate new tasks dynamically, depending on the received results. In the simple case, a few primary message passing (MPI) calls are enough to implement the model on a distributed-memory platform with a self scheduling mechanism where inactive workers dynamically probe the master for work. On the other hand, limitations and difficulties arise if advanced functionality is needed. First, because of the bottleneck at the master, the model may suffer from low scalability. Hierarchical task parallelism and techniques like distributed task queues require additional and non-trivial programming effort. Finally, a pure MPI-based implementation cannot easily adapt to take advantage of a multi-core node's physically shared memory.

* This work is supported in part by the Artemisia SMECY project (grant 100230).

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 14–26, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Although exploring new languages and programming models is currently a major issue in the parallel processing research community (and possibly the ultimate solution to leveraging current and emerging parallel hardware), other pragmatic approaches seem more promising for wide adoption in the short- to medium-term. Programming constructs that extend without changing a popular language have proven quite successful, OpenMP [2] being the most prominent example. Along the same lines, interoperability with popular programming models is another important requirement, easing the utilization of existing codebases.

In this work we present HOMPI, an infrastructure for programming and executing task-based applications on clusters of multi-cores. It consists of a source-to-source compiler that provides for simple directive-based definition and execution of tasks and a runtime library that orchestrates the execution over a variety of platforms, including pure shared-memory systems and clusters of such nodes. HOMPI targets message passing, shared address space and hybrid programs. In the standard master-worker case, the programmer does not have to use low-level message passing primitives at all, hiding away the communication details while providing load balancing transparently. For more advanced functionality, HOMPI integrates the tasking model into traditional MPI programs, allowing one or more MPI processes to independently spawn tasks. Each task may also spawn OpenMP-based parallelism, allowing seamless hybrid programming possibilities. The compiler supports OpenMP by design while the runtime system provides unified support for OpenMP threads as well as remotely executed tasks.

A number of programming tools and languages for task parallelism have been proposed recently for contemporary and emerging architectures. On shared-memory platforms, OpenMP has been extended in V3.0 with support for a tasking model [2], similar to Cilk [1]. For the Cell BE, runtime libraries include MPI microtask [3], ALF [4] and StarPU [5]. HOMPI's programming model borrows the `#pragma`-based annotation style of OpenMP and is reminiscent of other proposals such as HMPP [6] and StarSs [7], which combine runtime and compiler support to provide a (limited) task-based environment. These proposals target mainly accelerator-equipped systems and, in contrast to HOMPI, they do not support the divide-and-conquer model, since they do not allow recursive parallelism.

The contribution of this work is twofold. First, we introduce an easy-to-use programming framework which preserves the base language, while providing convenient code annotation for expressing task-based master/slave and divide-and-conquer parallel algorithms. While the annotation style is in the spirit of other proposals, to the best of our knowledge, HOMPI is the first of its kind targeting (and fully exploiting) clusters of SMPs/multi-cores. Second, albeit self-contained, our framework is fully interoperable with standard programming systems like MPI and OpenMP, allowing legacy or already parallelized code to be trivially integrated in an application. In our opinion this is a crucial attribute for the viability of any programming model proposal.

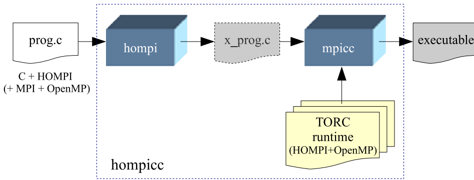

Fig. 1. HOMPI compilation procedure

```
#pragma hompi taskdef in(n) out(res)
void fib (int n, unsigned long *res) {
    unsigned long res1, res2;

    if (n <= 1) {
        *res = n;
    } else {
        #pragma hompi task
         fib(n-1, &res1);
        #pragma hompi task
         fib(n-2, &res2);
        #pragma hompi tasksync
        *res = res1+res2;
    }
}

void main(int argc, char *argv[]) {
    unsigned long res;
    fib(50, &res);
}
```

Fig. 2. Recursive Fibonacci in HOMPI

2 Programming Environment

HOMPI is based on a source-to-source translator that can handle `#pragma`-based directives within the user code, similar to OpenMP. Fig. 1 shows the compilation steps: from the annotated source code, the source-to-source compiler (`hompi`) produces an intermediate, transformed C file (`x_prog.c`) augmented by runtime calls. This file is then compiled by the system's native `mpicc` compiler and linked with HOMPI's runtime libraries to produce the final executable. The whole process is automated by the `hompicc` script.

HOMPI's execution model assumes that an application consists of multiple MPI processes with private memory, running on cluster nodes. Furthermore, multithreading is used to exploit the multi-processor/core configuration of a node; each process consists of one or more kernel threads sharing the process memory.

A task in HOMPI corresponds to the remote execution of a function on a set of data that are passed as arguments to this function, in the spirit of remote procedure calls. Tasks are executed asynchronously and in any order, without any data dependencies or point-to-point communication between them. Tasks have a parent-child relationship and can be arbitrarily nested, allowing multiple levels of parallelism and straightforward coding of divide-and-conquer algorithms.

The HOMPI programming model in essence requires that the programmer only designates which of the program functions can be used as tasks and be executed on (possibly) remote nodes. In many cases, just two directives are enough for the application to take advantage of the infrastructure, resulting in minimal programmer effort. The directive for designating a function as an independent task is `taskdef` and is placed right before the definition of a C function. A `taskdef` directive may contain *intent* clauses, similar to intent attributes of Fortan 90, which specify the intended usage of the function arguments: `in(`*variable-list*`)`, for variables that are to be passed to the function by value, `out(`*variable-list*`)`,

for results returned by the function and `inout(`*variable-list*`)` for variables whose values are passed to the function but will also be used to return a result.

An example is given in Fig. 2, which presents a complete HOMPI application that uses a recursive function (fib) to compute the 50th Fibonacci number. The `taskdef` directive designates the fib function as a task that accepts an argument by value (`n`) and computes a result (`res`). If any of the arguments is an array, the number of elements must be known, and this is either determined by `hompi` from the function prototype (if a size expression exists) or must be specified explicitly in the intent clauses of the `taskdef` directive.

The actual execution of a function as a task occurs with the `task` directive, which must be placed right before the function call. Finally, task joining (blocking until all child tasks finish their work) is possible anywhere in the code through the `tasksync` directive. In Fig. 2, the fib function generates two new tasks that are distributed across the available workers and waits for their results. Notice (i) the complete absence of explicit messaging and (ii) that if the directives are ignored by the compiler, the program's semantics remain the same to pure sequential execution.

2.1 Callbacks, Reductions and Detached Tasks

Normally, a parent task creates an arbitrary number of tasks and uses the `tasksync` directive to suspend itself until all child tasks have finished and their results have been returned. HOMPI supports *callback* functions, which allow for asynchronous execution of post-processing code on the process where the parent task runs on, even if the parent task is suspended. The callback function is defined immediately following the task definition through a `callback` directive; the callback function specifier is generated by the compiler and assumes the exact same arguments as the corresponding task function, providing thus access to the input parameters and the result of the task. An example where the callback just prints the results of each generated task, is depicted below.

```
#pragma hompi taskdef in(a) inout(b[2])
void taskfunc(int a,int  *b) {
    b[1] = b[0] + a + 1;
}
#pragma hompi callback
{
    printf("result = %d\n", b[1]);
}
```

HOMPI also supports *reduction operations* (which can be actually seen as special cases of callbacks) for the common scenario where each child task computes a partial result which is collected by the parent to produce the final result. These operations (summation, product, etc.) replace the `out` intent clause and are specified exactly in OpenMP style, as seen below:

```
#pragma hompi taskdef in(a) reduction(+:b)
void taskfunc(int a, int  *b) {
    *b = a;
}
```

Reduction operations are supported for *both* scalar variables and arrays.

Finally HOMPI supports *detached* tasks, that is tasks that execute without the parent being able to wait on them. In such cases task management is left up to the programmer. Detached tasks are executed as such by including the `detached` clause in the `task` directive. They can be combined with callbacks which actually provide the only way for them to synchronize with their parents; for example, within a callback, a detached task can modify a condition on which the parent task is explicitly waiting. Moreover, new tasks can be created within the callback routine. Detached tasks combined with callbacks offer a powerful mechanism; for example, they can be used for implementing dependencies among arbitrary subsets of tasks.

2.2 Task Distribution and Scheduling

Although not always necessary, in many cases one needs to control how tasks are distributed across workers or cluster nodes (e.g. due to particular load balancing needs). HOMPI offers two ways for achieving this. First, it provides a standard cyclic distribution scheme with tunable parameters. This is especially useful when tasks are created within a iterative control structure (e.g. while, for). The parameters of this scheme include the *scope* (whether tasks are distributed per node or per worker), the *starting* point (node id or worker id) and the *stride* (increment). These parameters are specified using a `taskschedule` directive:

```
#pragma hompi taskschedule scope(workers) start(0) stride(1)
for (t = 0; t < 8; t++) {
    #pragma hompi task
    func();
}
```

If the stride is zero then all tasks are submitted to the target node or worker. The default scheduling policy is represented with the tuple (nodes, -1, 1), i.e. distribution across nodes with stride 1 starting from the current node.

The second mechanism allows the user to explicitly specify the node or worker where a task will be submitted for execution. This is achieved with the `atnode(`x`)` and `atworker(`y`)` clauses in the `task` directive, where x and y are the identities of the intended node and worker respectively, e.g.

```
for (t = 0; t < K; t++) {
    #pragma hompi task atworker(t % hompi_total_workers())
    func();
}
```

It must be noted that the runtime system of HOMPI, which is presented next, includes a work-stealing mechanism whereby tasks may be stolen from a node and executed at another. If this mechanism is activated then all the above refer to the *initial* placement of a task; the actual node/worker that will ultimately execute it may be different. To explicitly control this, tasks can also be classified as *tied* and *untied* (using homonymous clauses in the `task` directive), similarly to OpenMP 3.0; a tied task can never be stolen, and will run on the process it was initially submitted for execution.

3 TORC: The Runtime System

In this section we give a short overview of TORC, the runtime environment of HOMPI. More details can be found in [10]. TORC uses exclusively POSIX and MPI calls for portability and performance, and integrates seamlessly hardware shared-memory and message passing. It provides application adaptability to the same application code, or even binary, on both shared-memory multiprocessors/multicores and clusters of them. TORC views an application as a collection of MPI processes. Each process consists of one or more POSIX kernel threads that execute tasks and a *server* thread that is responsible for the remote queue management and the asynchronous data movement. There exist private and public worker-specific and node-specific ready queues where tasks can be submitted for execution. A two-level threading model is implemented, where each kernel thread is a worker that continuously dispatches and executes ready-to-run tasks.

Tasks are associated with the process (home node) they were created on and can be executed either locally or remotely. In the latter case, explicit but transparent to the user data movement takes place. A worker thread executes a task by calling the task function with the locally stored arguments. When it finishes, it sends a notification message back to the home node, along with any arguments that represent results (`out/inout`). These are received asynchronously by the server thread and copied on their actual memory locations in the address space of the home process. A running task that spawns parallelism can suspend its execution, waiting for the termination of all its child tasks. The execution state of the current task is saved, releasing the underlying kernel thread, which runs the scheduling loop for selecting the next-to-run task. When all child tasks have completed (and all callbacks, if any, have finished), the suspended task becomes ready for execution and eventually resumes. A callback is implemented as a tied task, submitted for local execution when the corresponding user task finishes and notifies its parent task. The submission is performed by either the worker thread that executes the user task (if this is executed locally) or the server thread of the same process.

Data transfer mechanisms. The low level communication subsystem of TORC is based on MPI. However, other data transfer mechanisms have also been considered. In particular, we have successfully integrated two more mechanisms: MPI-2's one-side communications and software distributed shared memory (SDSM). MPI-2's remote memory access (RMA) [8] supports data transfer through one-sided operations. In TORC, the MPI_Get routine is used for fetching input data and MPI_Put for writing the results back to the home node. On the other hand, SDSM implements the notion of global memory on distributed computing environments and provides implicit data movement through the memory consistency protocol [9]. One-sided operations and SDSM provide receiver-initiated data movement for remotely executed tasks, performed by the worker thread just before or during the execution of the task function. This on-demand data movement does not allow data pre-fetching opportunities for server threads but may avoid unnecessary data transfers if task stealing is enabled.

Dynamic load balancing. Spawning a large number of tasks can be an effective approach to distribute the work evenly among the available workers. The user can specify the node or worker where each task will be submitted for execution and then employ the internal stealing mechanism for untied tasks that TORC provides. A idle worker extracts and executes a task from its local ready queue. If this is empty and task stealing is enabled, the worker first searches for work in the rest of the ready queues of the same node and then visits randomly the remote nodes. The worker waits synchronously for a response from the server thread of the target node. The answer is either a message that denotes unavailability of work at the target node, or an untied task descriptor that will be immediately executed upon receipt. Remote task stealing includes the corresponding data movement, unless the task returns to its home node.

4 Mixed-Mode and Hybrid Programming

Although the default execution model of HOMPI is that of master-worker, mixing it with SPMD execution and dynamically switching between them may be beneficial or required. For instance, a task parallel program may take advantage of common scientific SPMD libraries built on top of MPI.

The `atnode(*)` clause is a special case in the `task` creation directive that provides the above functionality; at runtime, the application creates as many tasks as the number of available nodes. These tasks are marked as tied and are distributed across the cluster nodes. The specified task function is executed by a single worker on every node. This approach matches the execution model of MPI and at the same time allows for hybrid MPI + OpenMP programming. When all the tasks have finished, the execution model switches back to master-worker. In Fig. 3, we demonstrate the flexibility of the `atnode(*)` clause; the master broadcasts the global array (`ga`) to the other nodes by issuing a collective call to the native MPI_Bcast function (with all workers participating). In this way, the application does not need to send `ga` with every task, avoiding thus unnecessary data transfers.

The `atnode(*)` clause improves the programmability of our system by facilitating the insertion of legacy MPI codes into the supported task-based execution environment. Following a similar approach, native MPI applications can be seamlessly enriched with the tasking model that HOMPI provides. Specifically, by setting a particular environmental variable (`HOMPI_MODE`), the TORC library is initialized for SPMD execution and thus the primary thread of all MPI processes executes the main routine. Switching the MPI application's execution model to master-worker is possible with a special directive (`spmd_barrier`). In Fig. 4, one of the MPI processes (e.g. the one with rank 0) becomes the master task that spawns work, while the rest of the processes block at the `spmd_barrier` directive which converts them to workers, activating the scheduling loop in TORC. After task completion, the master process reaches `spmd_barrier` and all MPI processes resume their execution, while the execution model switches back to SPMD.

```
int ga[16];

#pragma hompi taskdef in(root)
void spmdfunc(int root) {
  /* legacy MPI code can run here */
  MPI_Bcast(ga, 16, MPI_INT, root, MPI_COMM_WORLD);
}

#pragma hompi taskdef out(b[16])
void func(int *b) {
  for (int i = 0; i < 16; i++) b[i] = ga[i];
}

main() {
  int res[8][16], root = hompi_node();
  for (int i = 0; i < 16; i++) ga[i] = i;

  #pragma hompi atnode(*)
  spmdfunc(root);

  for (int t = 0; t < 8; t++) {
    #pragma hompi task tied
    func(res[t]);
  }
  #pragma hompi tasksync
}
```

Fig. 3. Example of `atnode(*)`

```
#pragma hompi taskdef
void func() { ... }

main(int argc, char *argv[]) {
  /* legacy MPI code */
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...

  if (rank==0) {    /* tasking */
    for (int i = 0; i < N; i++) {
      #pragma hompi task
      func();
    }
    #pragma hompi tasksync
  }
  #pragma hompi spmd_barrier

  /* legacy MPI code continues */
}
```

Fig. 4. Tasking in MPI code

Implementation of hybrid programming. HOMPI allows expressing the intra-node parallelism of a task function with OpenMP directives, in accordance to the hybrid MPI + OpenMP programming model. However, because task functions are executed by TORC's underlying worker threads, the utilized OpenMP compiler must support interoperability between OpenMP and independent POSIX threads. Moreover, caution is needed because the combination of TORC threads and OpenMP threads can easily oversubscribe the system, a situation resulting in performance degradation [11].

To cope with the above problem, we have constructed a *unified library* that handles both levels of parallelism within the same compilation and runtime environment. In particular, we have introduced a threading layer that is implemented on top of TORC into the OMPi OpenMP compiler [12]. Thanks to the layered architecture of OMPi, TORC was attached as an opaque OpenMP thread provider, thus letting OMPi control OpenMP execution through TORC-provided threads while at the same time TORC handles tasks independently.

The HOMPI translator was implemented by extending OMPi's translator, in order to have it parse and transform the new directives. Both HOMPI tasks and OpenMP threads are executed within the same runtime infrastructure. Internally, as worker threads first access the private and then the public ready queues of their node, OpenMP parallelism has a higher priority with respect to internode parallelism expressed with HOMPI tasks.

5 Experimental Evaluation

In this section we present preliminary experimental evaluation of our HOMPI prototype. We report both benchmarking results and results from full applications executed on a Sun Fire x4100 cluster of 16 nodes interconnected with Gigabit Ethernet. Each node has 2 dual core AMD Opteron-275 processors running at 2.2GHz giving a total of 64 cores. The cluster nodes are running Linux 2.6, while HOMPI was built with GNU GCC 4.3 as the system's native C compiler and the MPICH2 implementation of MPI. Thanks to the design of TORC, the same application binary can exploit the 4 processor cores of a single node with several combinations in the number of processes and workers. Therefore, our performance results refer to both distributed and shared-memory organizations.

Data transfer overheads. To evaluate the three different data transfer methods we discussed in Section 3 (MPI, RMA and SDSM) implemented in TORC, we measure the time required for the remote execution of a single task with input argument an array of double-precision floating point numbers that has been initialized by the parent before task creation. The task computes the sum of the array elements. For a fair comparison, we spawn exactly one task and thus preclude any data prefetching through the server thread when MPI calls are used. Besides the data movement for the argument of the task, the measured time includes the overhead for the allocation of the descriptor and its insertion in the queue of the remote process, the execution of the task function and the notification of the parent task at the first process.

Figs. 5 and 6 illustrate the overhead of the three methods with respect to the argument size, for `in` and `inout` argument types respectively. Regarding SDSM, we provide results for two libraries: Mome [13] and Mocha [14]. To enforce the `inout` semantics for the SDSM case, the parent task on process 0 accesses the array after task completion. Due to the relaxed consistency model of Mocha, SDSM barrier calls were introduced in the benchmark code. We observe that the overhead of the MPI and RMA methods, which both involve explicit communication, is almost identical. SDSM exhibits significantly higher overhead, due to the page-based consistency protocol and the multiple invocations of the page-fault handler. The performance difference between Mocha and Mome is because the

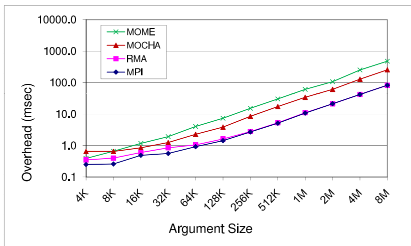

Fig. 5. Task execution overhead for the three data transfer methods (`in`)

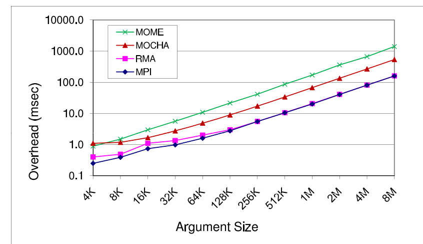

Fig. 6. Task execution overhead for the three data transfer methods (`inout`)

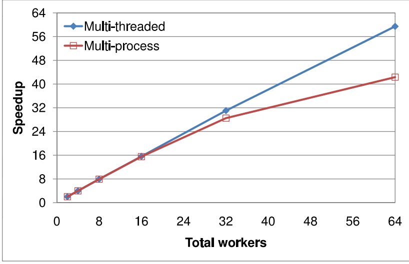

Fig. 7. EP performance

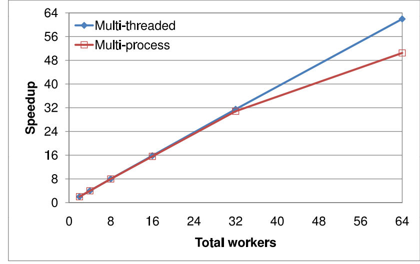

Fig. 8. PMCMC performance

former uses an 8KB (instead of 4KB) page size, resulting in a 50% reduction in the number of page faults.

Applications. For evaluating the performance of HOMPI we used two applications: EP and PMCMC. EP is an Embarrassingly Parallel benchmark that involves minimal inter-processor communication. The number of spawned tasks is equal to the number of workers, while the results of the tasks are accumulated through a reduction (+) operation. PMCMC implements an embarrassingly parallel Markov Chain Monte Carlo algorithm of the hard-disk problem. Each task is assigned a seed and performs a large number of Markov Chain computations. The code of this application was adapted from the ADLB library [15].

Fig. 7 depicts the performance of EP for 2^{28} random numbers and specifically the best observed speedup for a particular number of workers, using the multi-threaded and multiprocess configurations. When a single process per node and multiple workers per process are used, we observe that EP scales almost linearly; the slight performance degradation on 32 and 64 processors is mostly attributed to load imbalance effects due to the small number of generated tasks. The performance of the same application is significantly lower when multiple processes, of a single worker thread each, are deployed at each node of the cluster. The drawbacks of this configuration are the increased number of explicit messages and the oversubscription of processor cores due to the the multiple server threads on each node. Similarly, Fig. 8 presents the performance results for PMCMC when 128 independent tasks are used. We observe that the application exhibits almost perfect scalability for the multi-threaded approach. The performance of the multi-process approach is similar for all but the 64-process case, where its efficiency is significantly reduced to 78.87%.

Load balancing. We demonstrate the load balancing mechanism of HOMPI using the Mandelbrot application included in the LAM/MPI software package, rewritten to follow the tasking model of HOMPI. In our case, the main routine of the application creates a single task for each image block. The task receives as input arguments the coordinates of the block and as output argument an array for the image block. Each task is also associated with a callback routine which copies the processed block to the image region. Tasks are either distributed cyclically across the available workers or inserted in the queue of the master process.

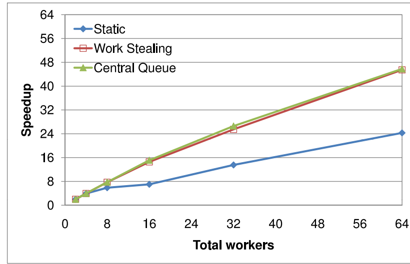

Fig. 9. Performance of task scheduling schemes on Mandelbrot

Table 1. Speedup of Mandelbrot on the (16,1,4) configuration for various task numbers

Nodes	Processes per node	Workers per process	Total workers	Total tasks	Static Scheduling	Work Stealing	Central Queue
16	1	4	64	256	24.30	45.47	45.79
16	1	4	64	512	26.70	50.74	50.32
16	1	4	64	1024	24.78	56.83	55.12

Table 2. Speedup of Mandelbrot for the hybrid programming model (256 tasks)

Nodes	Processes per node	Workers per process	OpenMP threads	Total workers	Static Scheduling	Work Stealing	Central Queue
1	1	1	4	4	3.79	3.79	3.79
2	1	1	4	8	7.33	7.52	7.54
4	1	1	4	16	13.30	14.94	14.98
8	1	1	4	32	22.14	29.02	29.26
16	1	1	4	64	24.30	54.99	56.62

Fig. 9 presents the speedup of the Mandelbrot application on the Sun cluster for an image of 2048x2048 pixels, 50000 maximum iterations for each pixel, and blocks of 128x128 pixels (256 tasks). We spawn a single process per node and provide results for the cyclic task distribution scheme, having the inter-node stealing mechanism disabled (termed 'Static Scheduling') or enabled (termed 'Work Stealing'). In addition, we evaluate the central-queue approach, where workers access the queue of the master process to get a task to execute. We observe that as the number of cores increases, the application manages to scale efficiently only if task stealing has been activated. For instance, the speedup of the application on 64 cores is approximately 24 and 45 for the static and work stealing approach respectively. The attained performance of the cyclic distribution and central queue are almost identical because, for this particular experiment, the latter does not suffer from bottlenecks as the server thread manages to handle the stealing requests efficiently. The scalability of the application declines with the number of processors, mostly because of the overhead for storing the results through the callback routine.

Table 1 studies the behavior of Mandelbrot for the (16 nodes, 1 process per node, 4 workers per node) configuration for smaller block sizes and thus a larger number of spawned tasks. It is apparent that better load balancing is achieved if the work stealing mechanism is enabled and thus the scalability of Mandelbrot

is further improved. As the number of tasks of finer granularity increases, the cyclic distribution scheme with work stealing begins to outperform the central queue approach.

Our last experiment demonstrates the effectiveness of hybrid programming for the Mandelbrot application. Specifically, a single process is spawned on each cluster node and the loop-level intra-task parallelism is expressed with OpenMP. The performance results are depicted in table 2. We observe that the attained performance of the hybrid programming approach is higher than that of the corresponding configurations in the previously presented experiments. This is attributed to better load balancing, as the 256 tasks are distributed to a smaller number of processes, and the full utilization of OpenMP threads as the serial fraction of the task function in this particular application is negligible.

6 Conclusion

This paper presents HOMPI, a directive-based programming and runtime environment for task-parallel applications on clusters of multiprocessor/multi-core nodes. The framework consists of a source-to-source C compiler that understands a small number of `#pragma`-based directives which allow for rather straightforward task creation and scheduling across the cluster. The output of the compiler contains calls to TORC, a sophisticated runtime library that handles all the task execution details, providing transparent load balancing and resulting in significant performance figures. The HOMPI infrastructure integrates features of several parallel programming models, from threads and OpenMP to MPI and remote procedure calls and in addition it is fully interoperable with them. As such, we believe its applicability will be quite general. We are currently extending our infrastructure on heterogeneous platforms and computational grids and introducing fault tolerance mechanisms.

References

1. Blumofe, R.D., Joerg, C.F., et al.: Cilk: An efficient multithreaded runtime system. J. Parallel Distrib. Comput. 37(1), 55–69 (1996)
2. OpenMP Architecture Review Board: OpenMP Specifications, `http://www.openmp.org`
3. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the cell broadband engine processor. IBM Syst. Journal 45(1) (2006)
4. IBM Corporation: Accelerated Library Framework (ALF) for Cell Broadband Engine programmer's guide and API reference. SDK for Multicore Acceleration V3.0
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: STARPU: A unified platform for task scheduling on heterogeneous multicore architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
6. Dolbeau, R., et al.: HMPP: A hybrid multi-core parallel programming environment. In: 1st Wrkshp on General Purpose Processing on GPUs, Boston, MA (2007)

7. Planas, J., Badia, R.M., et al.: Hierarchical task-based programming with StarSs. Int'l. J. of High Perf. Comput. Applic. 23(3), 284–299 (2009)
8. Geist, A., Gropp, W., Lusk, E., et al.: MPI-2: Extending the message-passing interface. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, Springer, Heidelberg (1996)
9. Li, K., Hudak, P.: Memory coherence in shared virtual memory systems. ACM Trans. on Computer Systems 7(4), 321–359 (1989)
10. Hadjidoukas, P.E., Dimakopoulos, V.V.: TORC: a tasking library for multicore clusters. Tech. Report TR-2011-6, CS Dept., University of Ioannina, Greece (2011)
11. Hadjidoukas, P.E., Dimakopoulos, V.V.: Nested parallelism in the OMPi OpenMP C compiler. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, Springer, Heidelberg (2007)
12. Philos, G.C., Dimakopoulos, V.V., Hadjidoukas, P.E.: A runtime architecture for ubiquitous support of OpenMP. In: 7th Int'l. Symposium on Parallel and Distrib. Comput., Krakow, Poland (2008)
13. Jegou, Y.: Implementation of page management in Mome, a user-level DSM. In: 3rd IEEE Int'l. Symposium on Cluster Comput. and the Grid, Tokyo, Japan (2003)
14. Kise, K., et al.: Evaluation of the acknowledgment reduction in a sDSM system. In: 6th Int'l. Conf. on Parallel Processing and Applied Math., Poznan, Poland (2005)
15. ADLB library, http://www.cs.mtsu.edu/~rbutler/adlb/