Int J Parallel Prog (2015) 43:894–917
DOI 10.1007/s10766-015-0350-0

Steal Locally, Share Globally

A Strategy for Multiprogramming in the Manycore Era

Ashkan Tousimojarad · Wim Vanderbauwhede

Received: 7 July 2014 / Accepted: 3 February 2015 / Published online: 28 March 2015

Abstract In a general-purpose computing system, several parallel applications run simultaneously on the same platform. Even if each application is highly tuned for that specific platform, additional performance issues are arising in such a dynamic environment in which multiple applications compete for the resources. Different scheduling and resource management techniques have been proposed either at operating system or user level to improve the performance of concurrent workloads. In this paper, we propose a task-based strategy called "Steal Locally, Share Globally" implemented in the runtime of our parallel programming model GPRM (Glasgow Parallel Reduction Machine). We have chosen a state-of-the-art manycore parallel machine, the Intel Xeon Phi, to compare GPRM with some well-known parallel programming models, OpenMP, Intel Cilk Plus and Intel TBB, in both single-programming and multiprogramming scenarios. We show that GPRM not only performs well for single workloads, but also outperforms the other models for multiprogramming workloads. There are three considerations regarding our task-based scheme: (i) It is implemented inside the parallel framework, not as a separate layer; (ii) It improves the performance without the need to change the number of threads for each application (iii) It can be further tuned and improved, not only for the GPRM applications, but for other equivalent parallel programming models.

Keywords Parallel programming · Multiprogramming · Task stealing · Manycore processors · GPRM · Intel Xeon Phi

A. Tousimojarad (✉) · W. Vanderbauwhede
School of Computing Science, University of Glasgow, Glasgow, UK
e-mail: a.tousimojarad.1@research.gla.ac.uk

W. Vanderbauwhede
e-mail: wim@dcs.gla.ac.uk

1 Introduction

Improving the performance of multiple parallel applications running together on the same machine is equally important as improving the performance of a stand-alone application. Thread and task scheduling in such a dynamic environment is a significant challenge, and scheduling algorithms designed for single-programming environments are no longer efficient. The problem lies in the assumption that a dedicated set of execution resources are fully available to the program, which is not always the case [1]. Therefore, in the presence of an external workload, such algorithms may lead to a significant drop in performance.

There are various programming models and runtime libraries that help developers to move from sequential to parallel programming. In this paper, we have chosen three well-known parallel programming approaches to compare their performance (in both single- and multi-programming cases) on a modern manycore machine. We have used three benchmarks with different features which exercise different aspects of the system performance. Moreover, two multiprogramming scenarios are used to compare the behaviours of these models when multiple applications reside in the system. Before going into the details of these models, we briefly introduce the manycore platform used in this work.

1.1 Intel Xeon Phi

The Intel Xeon Phi coprocessor 5110P used in this study is a Symmetric Multiprocessor (SMP) on-a-chip which is connected to a host Xeon processor via a PCI Express bus interface. The Intel Many Integrated Core (MIC) architecture used by the Intel Xeon Phi coprocessors gives developers the advantage of using standard, existing programming tools and methods. Our Xeon Phi comprises 60 cores (240 logical cores) connected by a bidirectional ring interconnect. More details can be found in Sect. 5.

1.2 Parallel Programming Models

All of the chosen programming models are supported by icc (Intel's C/C++ Compiler). The GPRM virtual machine, which is based on Pthreads, can be cross-compiled by icc for the Xeon Phi without any additional library.

1.2.1 OpenMP

OpenMP is the de-facto standard for shared-memory programming, and is based on a set of compiler directives or pragmas, combined with a thread management API. OpenMP has been historically used for loop-level and regular parallelism through its compiler directives. Since the release of OpenMP 3.0, OpenMP also supports task parallelism [2]. It is now widely used in both task and data parallel scenarios.

The Intel OpenMP runtime library (as opposed to the GNU implementation) allocates a task list per thread for every OpenMP team. Whenever a thread creates a task

that cannot not be executed immediately, that task is placed into the thread's deque (double-ended queue). A random stealing strategy balances the load [5].

1.2.2 Cilk Plus

Cilk Plus has evolved from Cilk [3], and is an extension to C/C++ with additional keywords and an array section notation. It comes provides very simple but powerful ways of specifying parallelism, as it is integrated into the compiler. It features a fork-join model to support irregular patterns and nesting. Cilk Plus provides the `_cilk_spawn` and `_cilk_sync` keywords to spawn and synchronise tasks; `_cilk_for` loop is a parallel replacement for sequential loops in C/C++. The tasks are executed within a work-stealing framework. The scheduling policy provides load balance close to the optimal [18]. The Intel implementation of Cilk Plus ensures that by running a program on one processor the same order of operations as the equivalent sequential program is produced [15].

1.2.3 TBB

Intel Threading Building Blocks (TBB) is another well-known approach for expressing parallelism [16]. TBB is an object-oriented C++ template library that contains data structures and algorithms to be used in parallel programs. It supports both regular and irregular parallelism, and has direct support for a various parallel patterns, such as task graphs, map, pipelines, etc. TBB abstracts the low-level thread interface. However, conversion of legacy code to TBB requires restructuring certain parts of the program to fit the TBB templates. Similar to Cilk Plus, a common thread pool is shared by all tasks and load balancing is achieved by work-stealing. Each worker thread in TBB has a deque of tasks. Newly spawned tasks are put at the back of the deque, and each worker thread takes the tasks from the back of its deque to exploit temporal locality. If there is no task in the local deque, the worker steals tasks from the front of the victims' deques [13].

1.2.4 The Glasgow Parallel Reduction Machine (GPRM)

The Glasgow Parallel Reduction Machine (GPRM) [21] provides a task-based approach to manycore programming by structuring programs into *task code*, written as C++ classes, and *communication code*, written in GPC, a restricted subset of C++. The communication code describes how the *task*s interact. GPC is a simple functional language with parallel evaluation, and using a C++ syntactic veneer. What this means is that it is possible to compile task code and GPC communication code with a C++ compiler and get correct functionality, but without the parallelism.

The GPC compiler compiles the communication code into the Glasgow Parallel Intermediate Representation (GPIR) (the *task description* code), which is an S-expression based, bare-bones functional languages inspired by Scheme [19], e.g. $(S_1\ (S_2\ 10)\ 20)$ represents a task S_1 taking two arguments, the first argument is the task S_2 which takes as argument the numeric constant 10, and the second argument is

the numeric constant 20. GPIR is further compiled into lists of *bytecode*s, which the GPRM virtual machine executes with concurrent evaluation of function arguments—in other words the VM is a coarse-grained parallel reduction machine where the methods provided by the *task code* constitute the instructions.

2 GPRM Architecture

The number of threads in GPRM is set to the number of logical cores of the underlying hardware. Each thread runs a *tile*, which consists of a *task manager* and a *task kernel* (Fig. 1). The *task kernel* is typically a self-contained entity offering a specific functionality to the system, and on its own is not aware of the rest of the system. The task kernel has run-to-completion semantics. The corresponding *task manager* provides an interface between the kernel and other *tile*s in the system. Since threads in GPRM correspond to execution resources, for each processing core there is a thread with its own *task manager*. The GPRM system is conceptually built as a network of communicating sequential *tile*s that exchange packets to request computations and deliver results.

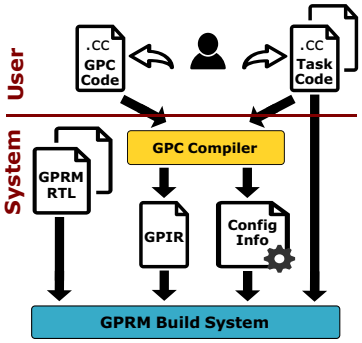
At first glance, the GPRM model may seem static and intolerant to runtime changes. However, in this paper, we discuss how it can efficiently balance the load and thrive in dynamic environments. For load balancing, compile-time information about the task dependencies is combined with an efficient task-stealing mechanism. Moreover, by keeping track of the applications that reside simultaneously in the system, it improves the performance of those applications markedly.

2.1 GPRM Model of Parallel Execution

The fundamental difference between GPRM and the other models is that GPRM does not use the fork-join model for task management. As a consequence, most of the techniques used in the GPRM runtime system, e.g. the stealing mechanism, are rather different from what readers might have in mind, and therefore need more explanation. In the fork-join model, at a *fork* point, new serial control flows are branched from an existing serial control flow. At a *join* point, these control flows can (possibly selectively) synchronise and merge.

GPRM, on the other hand, analyses all the C++ classes and methods used in the GPC program at compile-time, and maps them to numeric constants. These constants are used in a wrapper function to match the operation from the GPIR code with the actual method call to be executed. This task-specific generated code is combined with the generic GPRM runtime library and the source code for the task classes This process is summarised in Fig. 1a. Unlike the fork-join model, GPRM has no keywords for task creation or synchronisation, as all the functions in the GPC code are evaluated in parallel, unless otherwise stated (i.e. if a `seq` pragma is used).

When launching GPRM, a pool of POSIX threads (typically equal to the number of available cores) is created before the execution of the actual program starts. The tasks are initially assigned to the threads, based on the indices provided by the GPC compiler into the GPIR code (the indices of the tasks in Fig. 1c). Because of the restrictions

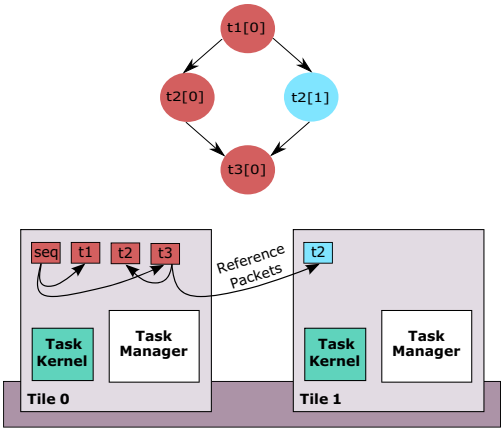


(a) User's view v.s. system's view

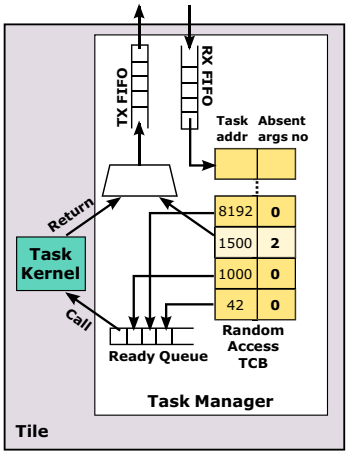
```
int x,y;
GPRM::Kernel::Body B;
#pragma gprm seq
{
  B.t1(x,y); //Init
  B.t3(B.t2(x), B.t2(y));
}
```

```
class Body {
 public:
   int t1(int,int);
   int t2(int);
   int t3(int,int);
 };
```

(b) GPC - Task code(.h)



(c) Task dependencies - Mapping to the tiles



(d) Abstract view of tiles

Fig. 1 **a** Users write GPC and Task codes. The rest is handled by the GPRM framework. **b** Sample GPC code and the C++ header file for the Task code. **c** Task dependencies for the example code in (**b**), and the allocation of the tasks on tiles. Four reference packets requesting computations are shown. **d** Internal structure of tiles. If all the arguments of a task in the TCB table become ready, it will be pushed into the *Ready Queue*. Otherwise, references will be sent to the others

imposed on the GPC language, its abstract syntax is that of a functional language, and hence the indices can be automatically generated based on the dependencies in the call tree; tasks that depend on one another cannot run in parallel and therefore can have similar indices (can be mapped onto the same cores). The parallel execution is achieved by parallel evaluation of the bytecode, as follows:

- Computations are triggered by the arrival of a *reference packet*, a packet which contains a reference to a *task*, i.e. a piece of bytecode representing an S-expression, e.g. the first reference packet for the example in Fig. 1b will be a reference to the built-in task `seq`, which sequences the operations.
  - Each argument in this S-expression is either a reference or a constant, e.g. both arguments of the task t1 in Fig. 1b are constants and both arguments of the task t3 are references.

  - References are sent out to other tiles for computation; values are stored. 4 reference packets are shown in Fig. 1c (curved arrows). As another example, in Fig. 1d, the task with address 1500 sends a reference through the tile's TX FIFO to another tile.
  - The leaf subtasks of the computational tree have either no arguments or constant values as arguments, so no references need to be sent.
- Once all arguments have been evaluated, the reduction engine passes the evaluated arguments of the S-expression to the *task kernel* which performs the actual computation. This is shown as a part of the tile structure in Fig. 1d with the *Call* and *Return* arrows.
- The result of the computation is returned to the caller, i.e. the sender of the reference packet. The result of the computation from the *task kernel* in Fig. 1d (the tile structure) is sent to the caller through the tile's TX FIFO.

There are other components in the tile structure in Fig. 1d that need to be explained. Each *tile* receives packets from others in its *task manager*'s RX FIFO. On the receipt of a *reference packet*, its corresponding *task record* is created. The newly created *task record* is stored in a random-access table called *Task Control Block (TCB)*,[1] which stores information about the tasks, particularly the number of their absent arguments. If the number of absent arguments is non-zero, references will be sent to other tiles in order to request computations, otherwise, as soon as all arguments of a *task* become ready, it will be pushed into the *ready queue* for computation. Therefore two cases would result in sending a packet to the other tiles: (i) a reference packet from a task with absent arguments requesting computations, (ii) a data packet containing the pointer to the result of the computation to the caller tile. Therefore, scheduling in our system, similar to other reduction machines (although it has coarser granularity at task level, rather than instruction level) is based on the need for data; this is known as the demand-driven model [26].

3 Task Stealing

The two features of the GPRM runtime system of specific interest to this work are *Task Stealing* and *Global Sharing*. [2] *Task Stealing* is the process of stealing *task*s from the *ready* tasks queues of other threads. If enabled, it allows threads to steal *task*s from each other when they become idle after finishing their jobs. This can balance the load if there are enough tasks in the ready queues. We denote GPRM with stealing enabled as *GPRM-Steal* (or *GPRM-S*).

3.1 Comparison of the Stealing Techniques

At first sight, the stealing mechanism may seem quite similar to the classical work-stealing approaches [1,3,4], but there are fundamental differences, due to the nature

[1] TCB is called "subtask list" in the previous GPRM papers.

[2] These features can be enabled via command-line switches when compiling the GPRM runtime system. It that sense they can be considered as runtime support features.

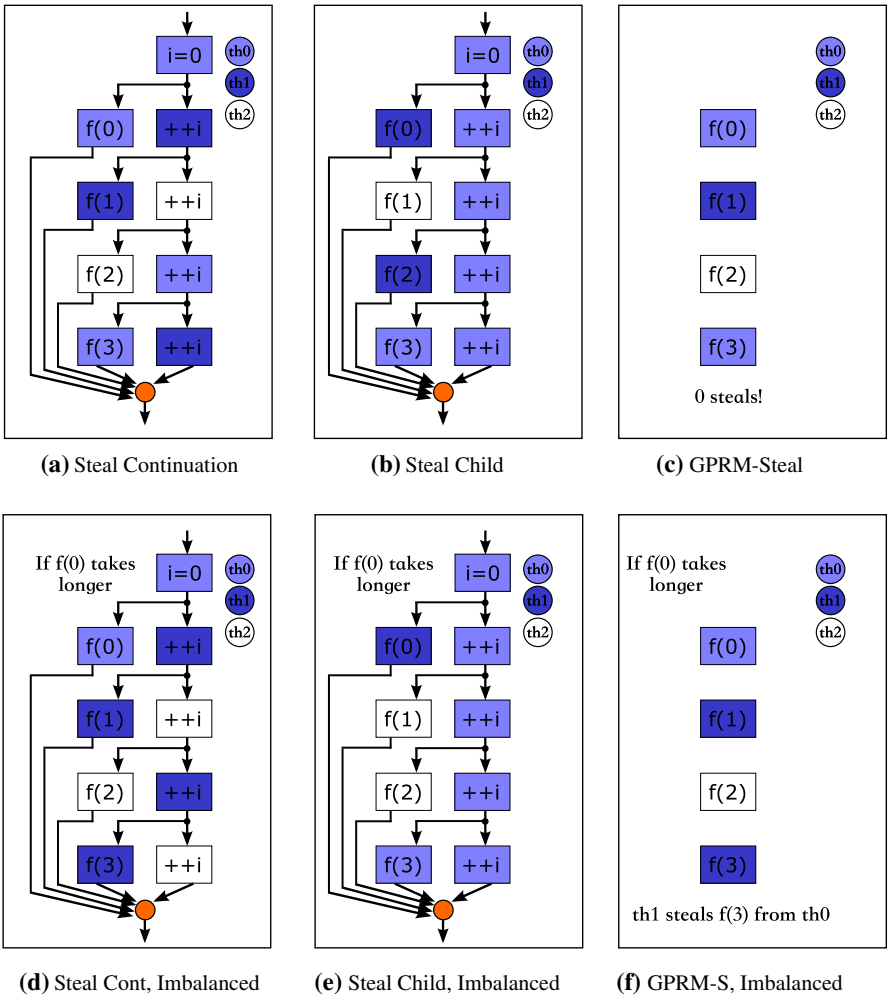
of our parallel programming model. In a fork-join model, when control flow forks, the master thread executes one branch and the other branch can be stolen by other threads (thieves). Multiple branches can be generated as the program is executed. This classical approach needs double-ended queues (deques), such that the workers work at the back of their own deques, while thieves can steal from the front of the others' deques. *Steal child* (used by TBB)—the newly created child becomes available to the thieves—and *steal continuation* (used by Cilk Plus)—the continuation of the function that spawned new task becomes available to the thieves—are two variations of the conventional work-stealing approach [15].

In GPRM, we do not use a fork-join model for task creation, hence there is no concept such as task spawning.: the C++ methods used in the GPC code are compiled into tasks. At compile-time, the compiler specifies the initial mapping between tasks and threads (even if the creation of a task is conditional, its initial host thread is specified). The parent tasks in the GPRM model are not the same as the parents in the Directed Acyclic Graph (DAG) as shown in Fig. 1c: rather, the parent tasks are the ones that request computations from their children, hence will depend on their children, e.g. in the DAG in Fig. 1c, t3 is the parent of the t2 tasks, following the order of the function calls: `B.t3(B.t2(x), B.t2(y));`.

With this background information, it is more clear what we mean by *task stealing*. Our stealing mechanism is about stealing the individual tasks, rather than the whole branch. In the conventional work-stealing approaches, the stolen branch would create more tasks during the execution of the program, and they would be executed by the thief (unless other workers become free and steal from that thief). The GPRM-specific task stealing mechanism is useful because all the tasks are initially allocated to threads (*tile*s). The stealing mechanism only tunes the initial allocation set by the compiler. Therefore, assuming that all the tasks are exactly the same and the number of them is a multiple of the number of the processing cores in the system, most probably no stealing occurs. In order to illustrate the differences between the stealing techniques in details, consider the following program written in Cilk Plus and GPRM. Rewriting it with other approaches is straightforward.

```
1 void f(int i) {
2  if(i==0) sleep(2);
3  else sleep(1);
4 }
5 ...
6 /* Cilk Plus */
7 for(int i=0; i < N; ++i)
8     cilk_spawn f(i);
9 cilk_sync;
10
11 /* GPRM */
12 #pragma gprm unroll
13  for(int i=0; i < N; ++i)
14      f(i);
```

Listing 1 Micro-benchmark to illustrate the differences between the stealing techniques



(a) Steal Continuation **(b)** Steal Child **(c)** GPRM-Steal

(d) Steal Cont, Imbalanced **(e)** Steal Child, Imbalanced **(f)** GPRM-S, Imbalanced

Fig. 2 **a** Steal continuation (used by Cilk Plus), balanced load: 3 steals. **b** Steal child (used by TBB), balanced load: 3 steals. **c** GPRM with stealing enabled, balanced load: 0 steals. **d** Steal continuation, imbalanced load: 4 steals. **e** Steal child, imbalanced load: 3 steals. **f** GPRM-steal, imbalanced load: 1 steal

1. Suppose we have only three threads in the system and $N = 4$. For the first case, assume that calling the function `f(i)` with different `i`s results in the same runtime:
   - In the *steal continuation* technique shown in Fig. 2a, `th0` (thread0) sets `i=0`, spawns `f(0)`, and immediately start executing `f(0)`, leaving the continuation of the loop available for stealing. `th1`—as an example—steals the continuation, updates `i`, and executes `f(1)`. `th2` could be the next thief that steals the further continuation and executes `f(2)`. Theoretically, `th0` finishes its work before the other threads, hence executes the next iteration and `f(3)`. the last

iteration can be stolen by th1. Therefore, 3 steals[3] can be considered for this simple case.
- For the *steal child* technique shown in Fig. 2b, th0 executes all iterations of the loop, spawns all f(i)s, and leaves them available to steal. However, since newly spawned tasks are put at the back of the deque and each worker thread takes the tasks from the back of its own deque (like TBB), therefore after all iterations, th0 executes f(3). f(2) can be stolen by th1. There are also 3 steals in this case.
- Since the GPC compiler unrolls the task creation loop and assigns the tasks to the worker threads, f(0) to f(2) will be assigned to th0 to th2, and f(2) will be assigned to th0. Theoretically, no stealing occurs, because if all threads finish their work at the same time, th0 would execute its next assigned task before others enter their stealing phase and steal it.

2. For the second case, consider the definition of f() in Listing 1, where executing f(0) takes more time:
   - The number of steals for the *steal continuation* becomes 4, as th1 and th2 can steal more continuations before th0 finishes its first job.
   - The number of steals can remain 3 for the *steal child* technique. th1 steals the first child and th2 the second. Assuming that th2 has started executing f(1) an epsilon before th0 reaches f(3), f(2) becomes available for th2 (note that th1 is still busy executing f(0)).
   - In the *GPRM-Steal* technique in Fig. 2f, assuming that th1 has started its work an epsilon before th2, it can steal f(3) from the *ready* queue of the busy thread (*tile*), th0.

Figure 2 shows that what we mean by task stealing is actually the minimum number of steals required to balance the load. Even for tiny micro-benchmarks, the techniques as well as the number of steals are quite different.

3.2 Implementation of Task Stealing in GPRM

As described above, inside GPRM every thread runs a *tile* with its own *task manager*. Each task manager consists of multiple queues for different purposes, such as exchanging packets (RX_fifo and TX_fifo) or storing results. As stated, one of these queues is the ready_fifo. All tiles are initially in the *non-stealing state*, which means their task managers take tasks from their own ready_fifo. If STEAL is enabled, then after running the last task in its ready_fifo, the task manager searches for jobs in the ready_fifos of other tiles (excluding the control tile). The tile remains in the *stealing state* and repeats this task-stealing process, unless there is no more job to steal after probing all ready_fifos of all tiles once. Then it goes to the *sleeping state*, and waits for a *reference packet* to wake it up and start a new computation. This mechanism is shown in List 2.

[3] We use the noun "steal" (OED "the act of stealing") rather than "theft".

```
1 Task TaskManager::taskSteal() {
2 for(int j=tileAddr; j < tileAddr+NTILES; ++j) {
3     int i=(j
4     Tile& victim = *(system.tiles[i]);
5     if(victim.TaskManager.ready_fifo.size() != 0) {
6         Task stolen = victim.TaskManager.lockReadyQ(1);
7         // 1 means a thief is locking
8         if(stolen != EMPTY) { // EMPTY is an int for null tasks
9             ... // registers it to the TCB of the thief
10            return stolen;
11        }
12    }
13 } // else, loop continues
14 return EMPTY;
15 }
```

Listing 2 Task Stealing inside the task manager

The `stolen` task shown in List. 2 can be null (`EMPTY`), because we do not lock the queues only to check whether their sizes are greater than 0. This way, we do not interrupt the routine operations of other tiles. However, inside the member function called `lockReadyQ()`, we check whether the size is still greater than 0, or the owner has processed all of its tasks already. If the latter is the case, the loop continues in order to find another victim.

The member function `lockReadyQ()` used in both Lists. 2 and 3 is responsible for locking the `ready_fifo` and returning the top element, if any. As shown in the code, it gets an argument which specifies whether a thief has locked the FIFO or not. As a result, we can define different actions for the owners and thieves inside this member function. For instance, for research purposes, it is possible to define conditions for task stealing. We can set rules for thieves to steal only from the queues with larger sizes than X. However, for the experiments in this study we did not set any rules.

Another member function of the class `TaskManager` is called `coreControl`, which is the centre for main operations inside the task manager. As shown in List. 3, a macro called `STEAL` is defined to ensure that if task stealing is disabled, no one locks the `ready_fifo`. This way, we can measure the real overhead of stealing, compared to when tiles only take tasks from their own queues. Three cases are shown in this function:

1. If `STEAL` is defined, the tile is either in its *stealing state* or not. If it is, `taskSteal()` will be called, and if successful the `status` will be set to `ready` for processing.
2. The second case is when `STEAL` is defined, and the tile is in the *non-stealing state*. If the task manager is idle, it will lock its own `ready_fifo` and take a task to process.
3. The third case is when the `STEAL` macro is not defined, and no stealing occurs in the system. Therefore, if the `status` is idle and the `ready_fifo` is not empty, its top element will be processed. There is no need to lock the `ready_fifos`, as every tile only uses its own queue in this case.

```
1 void TaskManager::coreControl(bool is_steal_state) {
2 // is_steal_state comes from the Tile class
3 #ifdef STEAL
4  if(is_steal_state) {
5     Task stolen_task = taskSteal(); // defined earlier
6     if(stolen_task != EMPTY) {
7        current_task = stolen_task;
8        status = ready; // the status of the task manager
9     }
10  }
11  else if(status == idle) {
12          // if STEAL, the queue size will be examined later
13          Task next = lockReadyQ(0);
14          // 0 means the owner is locking
15          if(next != EMPTY) {
16             current_task = next;
17             status = ready;
18          }
19  }
20 #else // STEAL is disabled
21  if(status == idle and ready_fifo.size() > 0) {
22     current_task = ready_fifo.front();
23     ready_fifo.popFront();
24     status = ready;
25  }
26 #endif
27 ...
28 }
```

Listing 3 Core Control inside the task manager

4 Global Sharing

The *Global Sharing* feature is intended for use in dynamic environments, where multiple parallel workloads compete for the resources. The proposed method uses a globally shared data structure that keeps track of thread mapping information. This data structure can be implemented in a runtime system as in our work, or could be embedded in the OS kernel.

Every instance of GPRM (every application) maps this shared data structure to its own memory space, and uses it to share information with others. The information we share, although quite minimal, is crucial to achieve good performance. In GPRM, all sequential tasks run on a specific tile. Generally, sequential tasks are those responsible for initialisation, or the ones that have to run alone after a synchronisation point. In either case, they cannot be stolen, because they are the only ready-to-run tasks existing in the system, and except the tile they are attached to, all other tiles are in the *sleeping state*.

In order to avoid running the sequential parts of different workloads on the same core, the corresponding `tileAddr` is shared between all GPRM applications present in the system. Therefore, every newly arrived GPRM application maps its threads to

cores such that its "sequential tile"[4] is pinned to the first available position that is not devoted to sequential tiles of other applications. All other threads of that application will be arranged in order. On the Xeon Phi, where four logical cores form one physical core, the target candidate will be the next physical core.

Although more information could be shared between concurrently present applications, it is important to keep the overhead low. Moreover, even if other information such as the number of active/asleep threads is shared, there is no clue as to whether they would remain the same or not. Thus, such information would have to be shared frequently. We will show that with the small amount of information that we share currently, a noticeable performance improvement can be obtained. Whether or not sharing more information results in better performance remains to be investigated.

5 Experimental Setup

The Xeon Phi provides four hardware threads sharing the same physical core and its cache subsystem in order to hide the latency inherent in in-order execution. As a result, the use of at least two threads per core is almost always beneficial [12]. The Xeon Phi has eight memory controllers supporting 2 GDDR5 memory channels each. The clock speed of the cores is 1.053GHz. Each core has an associated 512KB L2 cache. Data and instruction L1 caches of 32KB are also integrated on each core. Another important feature of the Xeon Phi is that each core includes a SIMD 512-bit wide VPU (Vector Processing Unit). The VPU can be used to process 16 single-precision or 8 double-precision elements per clock cycle. The third benchmark (Sect. 6.3) utilises the VPUs.

All the benchmarks are implemented as C++ programs, and all speedup ratios are computed against the running time of the sequential code implemented in C++ (which means they are directly proportional to the absolute running times). The benchmarks executed natively on the Intel Xeon Phi. For that purpose, the executables are copied to the Xeon Phi, and we connect to it from the host machine using `ssh`. The Intel compiler `icpc (ICC) 14.0.2` is used with the `-mmic` and `-O2` flags for compiling the benchmarks for native execution on the Xeon Phi. The OpenMP programs should be compiled with the `-openmp` flag. The TBB programs need the `-ltbb` flag. For all approaches excluding GPRM, some shared libraries must be copied to the MIC. For the OpenMP applications, the `libiomp5.so` library is required. The `libcilkrts.so.5` is needed for Cilk Plus applications and the `libtbb.so.2` library is required for the TBB programs.

In this study, we aim to show that by using our programming model, it is possible to achieve superior performance without changing the number of threads, and only by specifying the number of tasks, and hence, we do not change the number of threads for the GPRM approaches.

[4] The "sequential tile" is responsible for the sequential tasks, but also contributes to the parallel execution, whenever required.

6 Benchmarks

We have used three benchmarks for the purposes of this study. They are intentionally simple to make it possible to reason about the observed differences in performance between the selected models. We first compare the results for each single program

Two different comparisons are shown for every benchmark. The first comparison shows the speedup for varying numbers of threads. Since GPRM creates a pool of threads at the beginning of its execution, we only show the results with the default number of threads (as many as the number of cores), which is 240 on the Xeon Phi. Most of the time, users do not change the default settings. We show that for GPRM, this default choice always results in very good performance.

The second comparison illustrates the speedup with the default number of threads and varying cutoffs. Basically, the cutoff determines the number of tasks (or chunks). Choosing a small cutoff can restrict parallelism, but choosing a very large cutoff can saturate the system with a massive number of fine-grained tasks. The decision often depends on the input data set [7]. Usually, the cutoff value can be controlled by the user code. Leaving the decision to the runtime system has been proposed as an alternative.

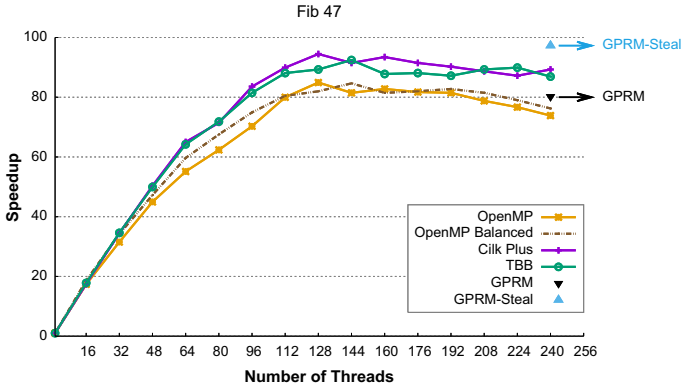
Adaptive Task Cutoff (ATC) [6] implemented in the Nanos [20] runtime system—a research OpenMP runtime system—is a scheme to modify the cutoff dynamically based on profiling data collected early in the program's execution.

In GPRM, setting the cutoff value less than the number of cores is fairly similar to having a smaller number of threads, because in this situation, tasks are assigned to a fraction of threads and the remaining threads are asleep and will remain so to the end of program. Larger cutoff values lead to the creation of more fine-grained tasks. Up to some point, the stealing mechanism can benefit from more numbers of tasks to balance the load. But above this threshold having lots of fine-grained tasks can impose too high overhead on the system. We have discussed such scenarios in [21] and [24], and proposed efficient solutions in GPRM (compared to GNU OpenMP) in order to control the number of tasks.

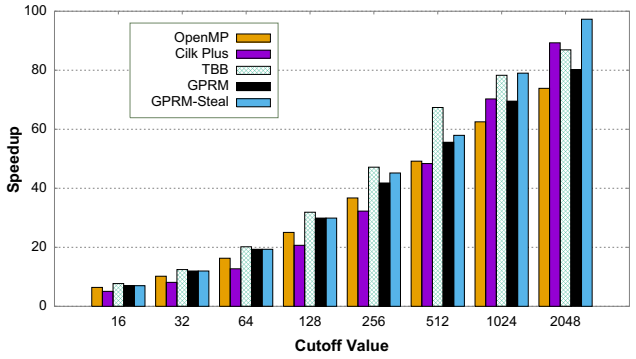
6.1 Parallel Fibonacci Benchmark: Fibonacci

We consider a parallel Fibonacci benchmark as the first testcase. The Fibonacci benchmark (calculating the Fibonacci numbers $fib(i) = fib(i-2)+fib(i-1)$ by concurrent recursion) has traditionally been used as a basic example of parallel computing. Although it is not an efficient way of computing Fibonacci numbers, the simple recursive pattern can easily be parallelised and is a good example of creating unbalanced tasks, resulting in load imbalance. In order to achieve desirable performance, a suitable cutoff value for the recursion is crucial. Otherwise, too many fine-grained tasks would impose an unacceptable overhead to the system. The cutoff limits the tree_depth in the recursive algorithm, which results in generating 2^{tree_depth} tasks.

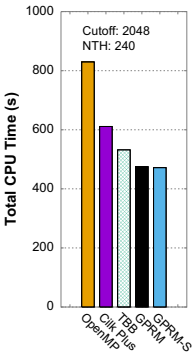
Figure 3 shows all the results taken from running this benchmark with different programming models. Figure 3a shows the speedup chart for $fib(47)$ with 2048



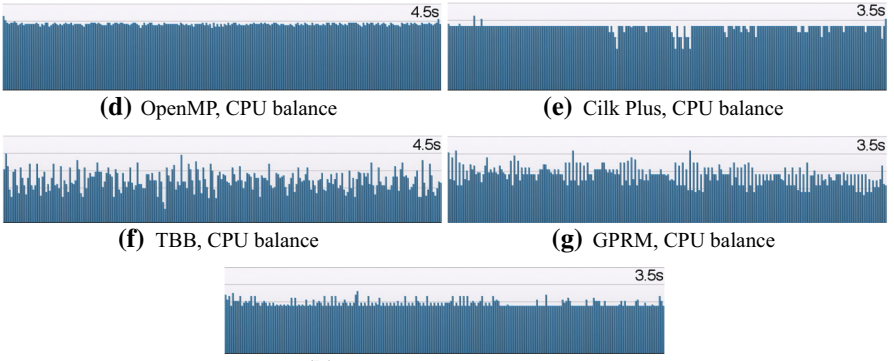
(a) Speedup, cutoff 2048, varying numbers of threads



(b) Speedup, NTH 240, varying cutoffs



(c) Total CPU Time



(d) OpenMP, CPU balance

(e) Cilk Plus, CPU balance

(f) TBB, CPU balance

(g) GPRM, CPU balance

(h) GPRM-Steal, CPU balance

Fig. 3 **a** The speedup of the parallel Fibonacci benchmark for the integer number 47. **b** Choosing a proper cutoff value is key to good performance. **c** Total CPU time consumed by this benchmark on the Xeon Phi. **d**–**h** A detailed breakdown of overall CPU time for the case with 240 threads and cutoff value 2048 is illustrated for each approach. The x-axis on these charts represents the logical cores and the y-axis is the time per (logical) core, from 0 to the maximum number specified in seconds

unbalanced tasks at the last level of the Fibonacci heap. Increasing the number of threads causes visible performance degradation for OpenMP. Setting the thread affinity `KMP_AFFINITY=balanced` results in a negligible improvement of the OpenMP performance. Cilk Plus and TBB show similar results. Cilk Plus with 128 threads comes close the GPRM-Steal performance, but it is not possible for the programmer to determine this number before running the experiment.

Figure 3b shows that larger cutoffs lead to better performance. It is due to the fact that the workload is unbalanced, and hence creating more tasks can result in a more even distribution of them. GPRM-Steal can balance the load even more. This is evident by comparing Fig. 3g, h.

Total CPU Time is a lower-better metric that shows the total CPU time (over all CPUs) consumed by the program. Intel TBB and both of the GPRM approaches have a better *Total CPU Time* compared to the other approaches in Fig. 3c. One reason is that in a model like GPRM, threads go to sleep immediately after finishing their jobs, while e.g. in the Intel OpenMP, they spin-wait for 200 ms before going to sleep [11]. Although sometimes in solo execution of the programs, these extra CPU cycles (and generally the overhead of the runtime libraries) have negligible influence on the running time (wall time), they affect other programs under multiprogrammed execution considerably [25,27].

Figure 3d–h show a detailed breakdown of CPU times taken from the Intel VTune Amplifier XE 2013 performance analyser [14] when running Fib 47 with cutoff 2048 natively on the Xeon Phi. The x-axis shows the logical cores of the Xeon Phi (240 cores), and the y-axis is the CPU time, from 0 to the maximum number specified in seconds.[5]

It might be argued from Fig. 3a that the performance achieved by Cilk Plus with a smaller number of threads is close to GPRM-Steal with 240 threads, meaning that similar performance can be achieved with half the number of threads, which might be beneficial in terms of energy consumption.

This would be a valid criticism if GPRM threads were making the cores busy by spin-waiting, which is not the case. Instead, the GPRM threads go to sleep if they have no work to do, and hence do not consume CPU time. In order to corroborate this claim, we measured the average power consumption of the best result achieved by each model; all measurements fall in the range of 130–135 Watt. Moreover, there is no clue for the programmer on how to determine the optimal number of threads before running the experiment, while GPRM-Steal can provide the best performance with the default number of threads. We have shown that our stealing approach which results in load balancing is effective, while keeping the overhead low.

6.2 Parallel Merge Sort Benchmark: MergeSort

This benchmark sorts an array of 80 million integers using a parallel merge sort algorithm. The ith element of the array is initialised with the number $i * ((i\%2) + 2)$.

[5] For all experiments, results from the benchmarks kernel are considered in the figures (a) and (b), while in the other results taken from the VTune Amplifier, all information from the start of the application, including its initial phase and the CPU time consumed by the shared libraries is taken into account.

The cutoff value determines the place after which the operation should be performed sequentially. For example, cutoff 2048 means that chunks of 1/2048 of the 80M array should be sorted sequentially, in parallel, and afterwards the results will be merged two by two, in parallel to produce the final sorted array.

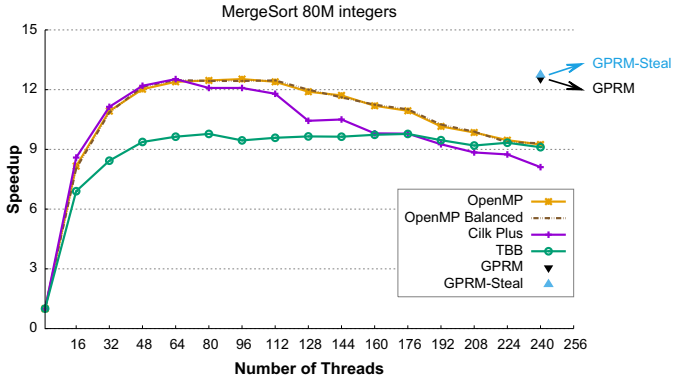
In the Fibonacci benchmark, the parent tasks were lightweight integer additions. But for the MergeSort benchmark, the parent tasks are heavyweight merge operations. Moreover, the children tasks at the leaves—the chunks that need to be sorted sequentially—are almost equal in size.

As shown in the Fig. 4a, this memory-intensive benchmark does not scale well [2]. The GPRM approaches have significantly better performance with the default number of threads. Both OpenMP and Cilk Plus perform well with smaller numbers of threads, but their performance drops as the number of threads increases. Using the `balanced` thread affinity for OpenMP has no noticeable effect. Since the child tasks are almost equal in size, a small cutoff would suffice, as shown in Fig. 4b. Larger cutoffs, though, do not cause a notable slowdown (as long as one does not create a very large number of tasks). Since the amount of work carried out for each task is fairly equal, there is no noticeable difference between both GPRM approaches. The speedup charts demonstrate how regular task-based applications similar to this reduction example are well suited to the GPRM model.

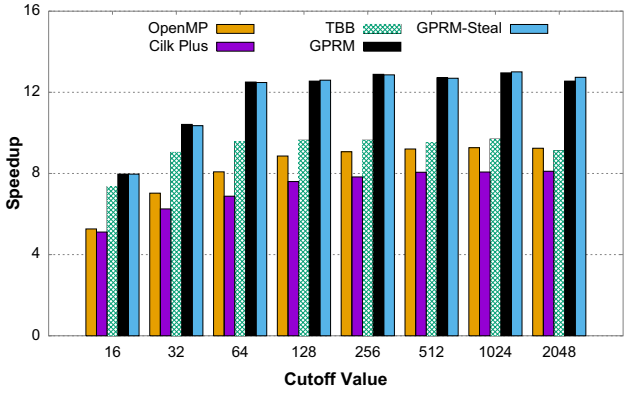
Figure 4c–h are again based on the results obtained by the intel VTune Amplifier. For this benchmark there is a significant difference between the *Total CPU Time* of TBB and GPRM on one side and OpenMP and Cilk Plus on the other side. Since all merges in a branch of the task tree can run on the same tile, one should not expect to see a balanced distribution for the GPRM approaches, and the balanced look in the cases of Cilk Plus and OpenMP is mostly because of the CPU time wasted by their runtime libraries [22].

6.3 Parallel Matrix Multiplication Benchmark: MatMul

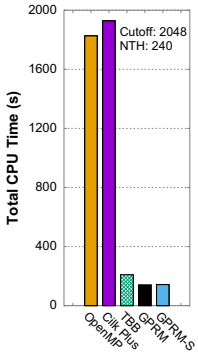
This benchmark performs a naive matrix multiplication by a triple nested loop with ikj loop ordering—for caching benefits—on square matrices of $N \times N$ double-precision floating point numbers. This is a data parallel problem which fits very well to OpenMP and its `for` worksharing construct. There is a concept similar to the cutoff in the loop parallelism context to control chunking. It specifies the size of chunk for each thread in a data parallel worksharing scenario. If the cutoff value is assumed as the number of chunks, the chunk (grain) size can be specified for the OpenMP `for` as follows: `#pragma omp for schedule(static, N/cutoff)`. The `static` keyword can be replaced by `dynamic` as well. Grain size in the Cilk Plus is similarly specified via a pragma: `#pragma cilk grainsize = N/cutoff`. GPRM uses its `par_cont_for` (parallel contiguous for) worksharing construct, which is a task-based approach to this problem, and distributes the chunks based on their indices amongst the working threads. If the cutoff value is assumed as the number of *task*s in GPRM, the chunk size will be `N/cutoff`. Parallel loops in GPRM are described in [24]. Intel TBB has a similar template function called `parallel_for`, which can be called with `simple_partitioner()` to control the grain size. In



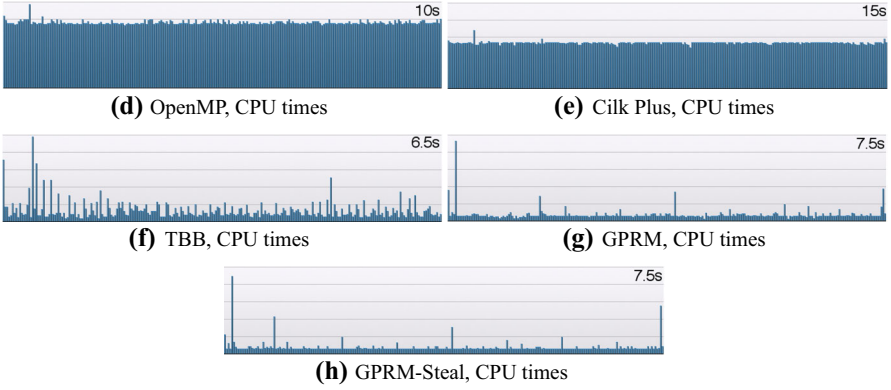
(a) Speedup, varying numbers of threads



(b) Speedup, varying cutoffs



(c) Total CPU Time



(d) OpenMP, CPU times

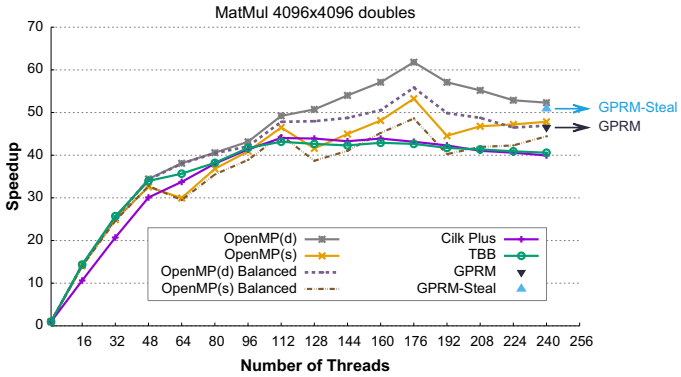
(e) Cilk Plus, CPU times

(f) TBB, CPU times

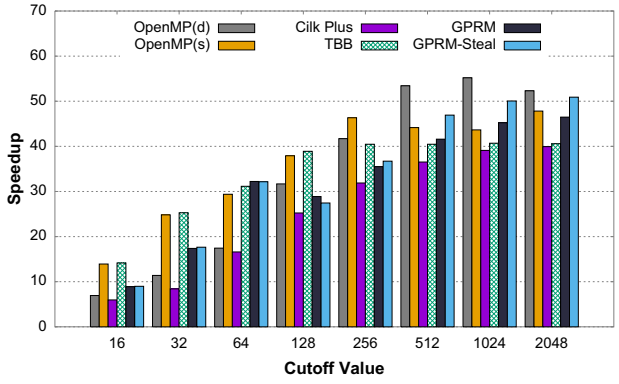
(g) GPRM, CPU times

(h) GPRM-Steal, CPU times

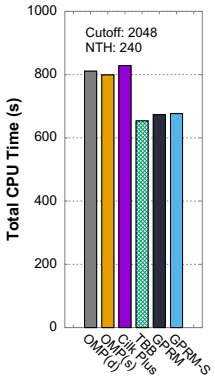
Fig. 4 **a** Speedup of the parallel MergeSort benchmark for an array of 80 million integers. This benchmark does not scale well. **b** Cutoff values greater than 64 are enough to lead to desirable performance with 240 threads. **c**–**h** For the OpenMP and Cilk Plus approaches, more than 75 % of the CPU cycles are consumed by their runtime libraries [22]



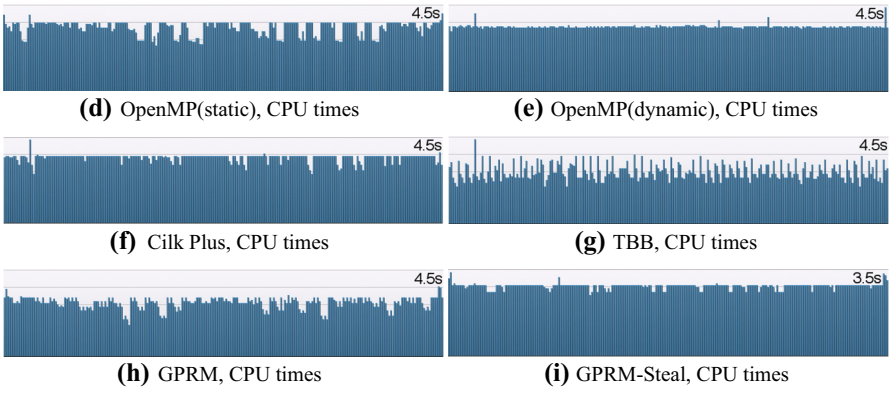
(a) Speedup, varying numbers of threads



(b) Speedup, varying cutoffs



(c) Total CPU Time



(d) OpenMP(static), CPU times **(e)** OpenMP(dynamic), CPU times

(f) Cilk Plus, CPU times **(g)** TBB, CPU times

(h) GPRM, CPU times **(i)** GPRM-Steal, CPU times

Fig. 5 **a** Speedup of the parallel MatMul benchmark on a 4096×4096 matrix of double numbers. OpenMP with dynamic scheduling has the best scaling amongst all, and both GPRM approaches scale better than Cilk Plus and TBB. **b** For cutoff values larger than 256, OpenMP (dynamic scheduling) has the best performance. **c** Again the Total CPU Time of TBB and GPRM is the least amongst all. **d**–**i** There is an evident distinction between the distribution of CPU times in the charts (**d**) and (**e**) that shows how dynamic scheduling in OpenMP leads to better performance

order to achieve automatic vectorization on the Xeon Phi, the Intel TBB and OpenMP codes have to be compiled with `-ansi-alias` flag.

For the GPRM approaches, all the tasks are the same, having fairly the same size. However, 4096 is not a factor of 240 (number of threads). Moreover, `cutoff=256` (making 256 tasks) makes 16 cores busier than the others. Although we could choose the `cutoff=240` to improve the performance, for consistency with other experiments, we have limited ourselves to powers of two. By increasing the cutoff, there will be more tasks and a better distribution, hence better speedup (Fig. 5).

7 Multiprogramming

In this section, we consider two multiprogramming scenarios to see how these models behave in multiprogramming environments. The metric that is used for the comparison is the user-oriented metric *Turnaround Time* [9], which is the time between submitting a job and its completion in a multiprogrammed system.

7.1 Case 1: Multiple Instances of a Single Program

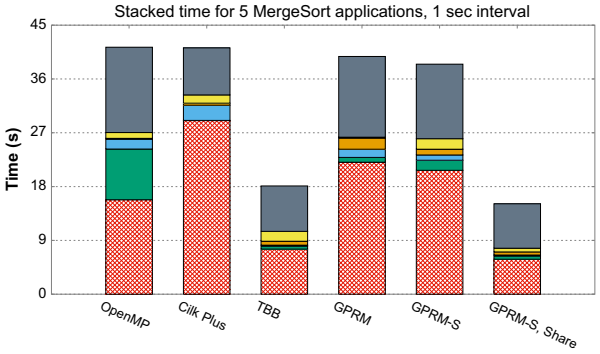
In order to show the effect of *Global Sharing* between different GPRM applications, consider 5 MergeSort applications entering the system with an interval of 1 s. Again, an array of 80 million integers with the cutoff value 2048 and the default number of threads (240) is considered.

The results are illustrated in Fig. 6. The stacked representation is used firstly to illustrate the difference between the kernel times (only the parallel parts) of the applications, e.g. in the OpenMP case, the difference between the best and worst kernel time is around 11.2 s, while for the "GPRM-Steal with Global Sharing", the difference is less than 2 s. The second use of the stacked representation is to show all other consumed time in the system at the top of the stacked column chart. This includes the time before job submission (interval) as well as the time spent on initialisations (sequential time). It is evident that for OpenMP, or GPRM approaches without Global Sharing, this time is also larger, which serves as an indication of the amount of overlap of the sequential parts of the applications. It can be seen in Fig. 6h how the bottleneck is removed with the help of *Task Stealing* and *Global Sharing*.

Beside the *Total CPU Time*, the hardware event *Instructions Executed*, estimated by multiplying sample count by 10M events per sample (obtained by the VTune Amplifier in Fig. 6b) can be used as another metric for comparison. Based on $(|(V1 - V2)|/((V1 + V2)/2)) * 100$ formula, for sum of the turnaround times, the difference between "GPRM-Steal with Global Sharing" and TBB as the second best result is around 20 %.

7.2 Case 2: Single Instances of Multiple Programs

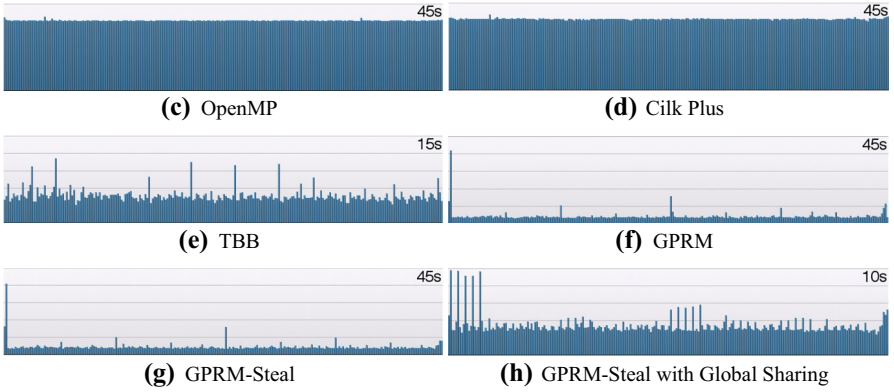
For this case, we consider another multiprogramming scenario, which consists all three benchmarks described before. The three benchmarks have the same input sizes as the



(a) Stacked Time

Approach	Total CPU Time(s)	Inst. Executed
OpenMP	8.7K	21.2K × 10M
Cilk Plus	8.8K	16.0K × 10M
TBB	1.1K	4.6K × 10M
GPRM	0.9K	3.8K × 10M
GPRM-S	0.9K	3.9K × 10M
GPRM-S, Share	0.8K	3.8K × 10M

(b) Performance Summary



(c) OpenMP **(d)** Cilk Plus

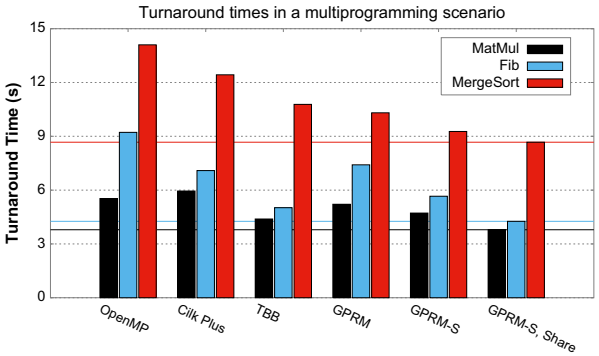
(e) TBB **(f)** GPRM

(g) GPRM-Steal **(h)** GPRM-Steal with Global Sharing

Fig. 6 A Multiprogramming case with 5 MergeSort applications with 1 s interval. The stacked column chart shows the mean time for each application's kernel, followed by the remaining time spent from the start of the first application to the end of the last one. GPRM-Steal with Global Sharing has the best performance; Cilk Plus has the worst

single-program cases with the cutoff value 2048 and the default number of threads 240. We do not start all of them at the same time. Rather, we want the parallel phases to start almost simultaneously, such that the threads of all applications compete for the resources. For that purpose, the MergeSort benchmark enters the system first. Two seconds later the MatMul benchmark enters the system, and half a second after that, the Fib benchmark starts.[6] The results are shown and discussed in Fig. 7. In sum of

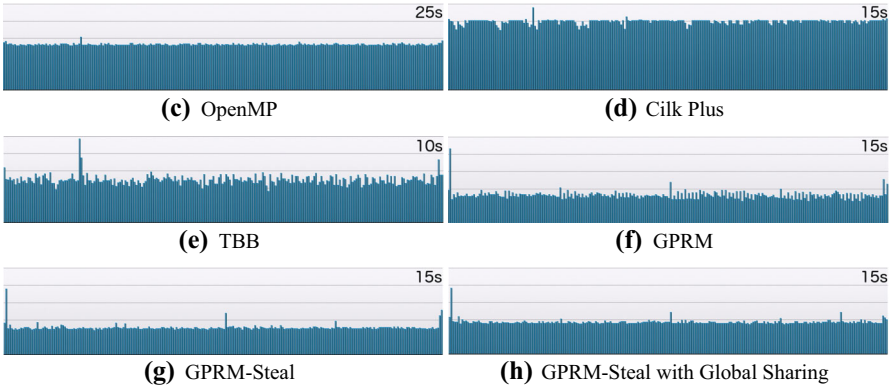
[6] The sequential phase of the MergeSort benchmark with the input size 80M is around 2 s, and the initial phase of the MatMul benchmark with the input size 4096×4096 is about half a second.



(a) Turnaround times

Approach	Total CPU Time(s)	Inst. Executed
OpenMP	3.2K	8.5K × 10M
Cilk Plus	2.9K	7.5K × 10M
TBB	1.2K	4.3K × 10M
GPRM	1.1K	4.1K × 10M
GPRM-S	1.1K	4.1K × 10M
GPRM-S, Share	1.2K	4.2K × 10M

(b) Performance Summary



(c) OpenMP **(d)** Cilk Plus

(e) TBB **(f)** GPRM

(g) GPRM-Steal **(h)** GPRM-Steal with Global Sharing

Fig. 7 A multiprogramming scenario with all the three benchmarks. The best turnaround times are obtained with "GPRM-Steal with Global Sharing". OpenMP has the worst performance amongst all. Performance of GPRM is improved by stealing tasks locally inside each application and sharing information globally across multiple applications

the turnaround times, the difference between "GPRM-Steal with Global Sharing" and TBB as the second best result is about 17 %.

Although the *Total CPU Time* is a key performance metric, it cannot be used solely to interpret the results. A sequential program can have the same value for the *Total CPU Time* as a parallel program. Therefore, it is also important to find out how evenly the tasks are distributed across the system. As we have observed in the multiprogramming cases, compared to other GPRM approaches, the efficiency of the "GPRM-Steal with Global Sharing" comes from its better load balancing. However, the wasted CPU cycles by the runtime libraries, as for OpenMP and Cilk Plus, which can have a

significant impact on the results can be detected by *Total CPU Time* and *Instructions Executed*.

8 Related Work

Saule and Catalyurek [18] have compared OpenMP, Cilk Plus, and TBB on the Intel Xeon Phi. They have focused on the scalability of the approaches for single-program graph algorithms. We have added GPRM to the comparison and have also targeted multiprogramming situations.

Callisto [10] is a user-mode shared library for co-scheduling multiple parallel runtime systems. Although there is no need to modify the high-level applications or the OS, it has to be linked with the Callisto-enabled versions of the runtime systems. The current version does not support OpenMP tasks. Furthermore, the authors have used pairs of benchmarks on a 2-socket machine. It needs more investigation to find out whether Callisto can be still effective if more benchmarks are run together, or if one moves from a socket-based machine to a modern architecture such as MIC.

Emani et al. [8] used predictive modelling techniques for OpenMP programs to determine an optimal mapping of a program in the presence of external workload. Dynamic runtime information is combined with the compile-time knowledge of the program to decide about the best adaptive mapping of programs to execution resources. Their purpose is to maximise the performance of a target program with minimum impact of the performance of the workloads.

Sasaki et al. [17] developed a sophisticated scheduling scheme to co-schedule multiprogram workloads, which predicts the applications' scalability dynamically, and allocates the optimal number of cores to applications in order to maximise the system utilisation. They have proposed a technique called *Core Donation* that maximises the CPU utilisation while keeping the scalability of the programs into account.

In our previous work [23], a thread mapping method based on the system's load information is developed for OpenMP programs. Performance of multiprogram workloads in Linux can be improved by sharing the load information and using it for thread placement. However, for this method to be effective, the optimal number of threads for each single program has to be known to the programmer. Most of time, though, the programs are run with the maximum number of threads, with the expectation of achieving the desired performance, and that is the case we have targeted in this paper.

9 Conclusion

In this paper, we have introduced a strategy for parallel programming called *Steal Locally, Share Globally*, implemented in GPRM, our task-based parallel programming framework. The idea is to steal tasks locally (from within the same application) only if the initial task assignment is not optimal, and to share the least amount of information about the system's load globally (between different applications). We have shown that our strategy is highly competitive against well-known approaches, namely OpenMP, Cilk Plus and TBB for all testbenches, and achieves the top performance on the Intel Xeon Phi in almost all cases.

Our main objective is to provide a low-overhead solution which can be efficient in different possible scenarios occurring inside a general-purpose system. The overhead of the runtime libraries has a serious impact on the performance of multiprogram workloads. On the other hand, the performance of single programs should be considered as well. For example, we have shown that TBB has a good performance for multiprogram workloads, but not the best performance for single programs. GPRM combines compile-time information about tasks with an efficient task stealing strategy and a very low-overhead sharing mechanism between different programs in order to achieve high performance.

Without a precise cost model, the programmer cannot determine the optimal number of threads for each application, hence the programming framework should attempt to deliver optimal performance with the default number of threads. The ideal situation for the programmer is to only have to express the parallelism, relying on the runtime system to get the expected speedup. As we have shown, GPRM delivers this ideal in almost all cases, and where it does not, it comes very close. Thus, GPRM combines an intuitive task-based programming model with excellent performance, without the need to tune the number of threads.

References

1. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. Theory Comput. Syst. **34**(2), 115–144 (2001)
2. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The design of openmp tasks. IEEE Trans. Parallel Distrib. Syst. **20**(3), 404–418 (2009)
3. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
4. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5), 720–748 (1999)
5. Clet-Ortega, J., Carribault, P., Pérache, M.: Evaluation of openmp task scheduling algorithms for large numa architectures. In: Euro-Par 2014 Parallel Processing, pp. 596–607. Springer, New York (2014)
6. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. pp. 1–11. IEEE (2008)
7. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona openmp tasks suite: a set of benchmarks targeting the exploitation of task parallelism in openmp. In: International Conference on Parallel Processing, 2009. ICPP'09. pp. 124–131. IEEE (2009)
8. Emani, M.K., Wang, Z., O'Boyle, M.F.: Smart, adaptive mapping of parallelism in the presence of external workload. In: 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–10. IEEE (2013)
9. Eyerman, S., Eeckhout, L.: System-level performance metrics for multiprogram workloads. IEEE Micro **28**(3), 42–53 (2008)
10. Harris, T., Maas, M., Marathe, V.J.: Callisto: co-scheduling parallel runtime systems. In: Proceedings of the 9th European Conference on Computer Systems, p. 24. ACM (2014)
11. Hofmeyr, S., Iancu, C., Blagojević, F.: Load balancing on speed. In: ACM Sigplan Notices, vol. 45, pp. 147–158. ACM (2010)
12. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High Performance Programming. Newnes (2013)
13. Kim, W., Voss, M.: Multicore desktop programming with intel threading building blocks. IEEE Softw. **28**(1), 23–31 (2011)
14. Lubin, M., McMillan, S., Kruse, C.G., Del Vento, D., Montuoro, R.: Efficient software development: 4 Whats new in intel parallel studio xe 2013 service pack (2013)
15. McCool, M., Reinders, J., Robison, A.: Structured Parallel Programming: Patterns for Efficient Computation. Elsevier (2012)

16. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism. O'Reilly Media, Inc. (2007)
17. Sasaki, H., Tanimoto, T., Inoue, K., Nakamura, H.: Scalability-based manycore partitioning. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pp. 107–116. ACM (2012)
18. Saule, E., Catalyurek, U.V.: An early evaluation of the scalability of graph algorithms on the intel mic architecture. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) pp. 1629–1639. IEEE (2012)
19. Sussman, G.J., Jr., G.L.S.: Scheme: an interpreter for extended lambda calculus. In: MEMO 349, MIT AI LAB (1975)
20. Teruel, X., Martorell, X., Duran, A., Ferrer, R., Ayguadé, E.: Support for openmp tasks in nanos v4. In: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research, pp. 256–259. IBM Corp. (2007)
21. Tousimojarad, A., Vanderbauwhede, W.: The Glasgow Parallel Reduction Machine: Programming Shared-Memory Many-Core Systems Using Parallel Task Composition. EPTCS 137, 79–94 (2013). doi:10.4204/EPTCS.137.7
22. Tousimojarad, A., Vanderbauwhede, W.: Comparison of three popular parallel programming models on the Intel Xeon Phi. In: Euro-Par 2014: Parallel Processing Workshops, pp. 314–325. Springer, New York (2014)
23. Tousimojarad, A., Vanderbauwhede, W.: An efficient thread mapping strategy for multiprogramming on manycore processors. In: Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing, vol. 25, pp. 63–71. IOS Press (2014). doi:10.3233/978-1-61499-381-0-63
24. Tousimojarad, A., Vanderbauwhede, W.: A parallel task-based approach to linear algebra. In: 2014 IEEE 13th International Symposium on Parallel and Distributed Computing (ISPDC), pp. 59–66. IEEE (2014)
25. Tucker, A.: Efficient Scheduling on Multiprogrammed Shared-memory Multiprocessors. Ph.D. thesis, Stanford University (1994)
26. Veen, A.H.: Dataflow machine architecture. ACM Comput. Surv. (CSUR) **18**(4), 365–396 (1986)
27. Yan, J., He, J., Han, W., Chen, W., Zheng, W.: How openmp applications get more benefit from many-core era. In: Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, pp. 83–95. Springer, New York (2010)