The Glasgow Parallel Reduction Machine: Programming Shared-memory Many-core Systems using Parallel Task Composition

Ashkan Tousimojarad, Wim Vanderbauwhede
School of Computing Science
University of Glasgow
Glasgow, UK
a.tousimojarad.1@research.gla.ac.uk, wim@dcs.gla.ac.uk

We present the Glasgow Parallel Reduction Machine (GPRM), a novel, flexible framework for parallel task-composition based many-core programming. We allow the programmer to structure programs into task code, written as C++ classes, and communication code, written in a restricted subset of C++ with functional semantics and parallel evaluation. In this paper we discuss the GPRM, the virtual machine framework that enables the parallel task composition approach. We focus the discussion on GPIR, the functional language used as the intermediate representation of the bytecode running on the GPRM. Using examples in this language we show the flexibility and power of our task composition framework. We demonstrate the potential using an implementation of a merge sort algorithm on a 64-core Tilera processor, as well as on a conventional Intel quad-core processor and an AMD 48-core processor system. We also compare our framework with OpenMP tasks in a parallel pointer chasing algorithm running on the Tilera processor. Our results show that the GPRM programs outperform the corresponding OpenMP codes on all test platforms, and can greatly facilitate writing of parallel programs, in particular non-data parallel algorithms such as reductions.

1 Introduction

As processor clock speeds have stagnated, processor manufacturers have moved to many-core devices in an attempt to perpetuate Moore's law. Processors with tens of cores are already commercially available and soon will be the norm, even in laptops. However, most programming languages were originally intended for single-core processors and consequently most software is written for single-core processors. Efficient utilisation of many-core platforms is a great challenge. POSIX threads enable parallel programming but they are difficult to use and put the burden on the programmer, even when using compiler directives such as OpenMP; there are number of languages where parallelism can be expressed natively without the need for explicit thread creation [17, 15, 1, 9], but compared to mainstream languages such as C++ and Java, none of them have found widespread adoption. Even if a multicore programming language would find wide adoption, it would in the short term obviously be impossible to rewrite the vast amount of single-core legacy code libraries, nor would it be productive. For many applications, especially computationally-intensive ones, sequential algorithms are extremely efficient. We therefore propose an approach to parallel programming based on parallel composition of (sequential) tasks. Our task composition approach can be integrated into existing code and has the advantage of providing a parallel task composition mechanism embedded in an existing language.

In order to express irregular parallelism, the concept of *tasking* is introduced in OpenMP 3.0 [2]. Although with the OpenMP tasks, one can express more parallelism, or write parallel codes more easily,

N. Yoshida, W. Vanderbauwhede (Eds.): Programming Language
Approaches to Concurrency- and Communication-cEntric
Software 2013 (PLACES'13)
EPTCS 137, 2013, pp. 79–94, doi:10.4204/EPTCS.137.7

it sometimes results in a situation in which lots of fine-grained tasks are created and the performance degrades dramatically [8]. We will show such a situation later in section 6.

Intel Threading Building Blocks (TBB) is a famous approach for expressing task-based parallelism [11]. Intel TBB is a C++ runtime library that contains data structures and algorithms to simplify parallel programming. It abstracts the low-level threading details required to utilise the multi-core systems, similar to the approach that we are using. However, there are still some important issues that put burden on the programmer, such as dealing with the mutual exclusion.

Cilk++ [5], which is a variation of Cilk that supports C++, uses some keywords to express task parallelism. In the Cilk++ the scheduling of tasks are predefined, while our approach allows for different task scheduling strategies to be defined statically or dynamically at run-time.

The SMP superscalar (SMPSs) project from the Barcelona Supercomputing Center [13, 6] also allows programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different processing cores by means of an automatic parallelisation at run time. The SMPSs runtime builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies. The SMPSs program code must be annotated using special preprocessor directives.

Our approach is to provide a language with default parallel evaluation implemented using a restricted subset of C++ which is familiar and easy to use for the end users.

2 A Task Composition Framework for Many-core Platforms

To facilitate reuse of existing single-core code libraries we propose a *task-based* approach to many-core programming, in particular for computationally intensive tasks. In our parlance, a *task node* consists of a *task kernel* and a *task manager*. A *task kernel* is typically a complex, self-contained unit offering a specific functionality. Such a kernel is said to provide one or more *services* to the system. A *task kernel* on its own is not aware of the rest of the system. The *task manager* provides the task composition interface to the kernel.

To implement this paradigm, we have created the Glasgow Parallel Reduction Machine (GPRM), a lightweight, distributed reduction engine for shared-memory and NUMA-style many-core and multiprocessor systems. The GPRM executes a strict functional machine language with concurrent evaluation of function arguments. As the name suggests, the GPRM is similar in spirit, if architecturally very different, to reduction machines such as Alice [4]. The concepts behind reduction machines are very well explained in [16]. Essentially, the GRPM is a coarse-grained reduction machine as we only reduce the task composition graph. The GPRM uses β-reduction and performs string reduction at bytecode level.

Conceptually, the GPRM consists of a set of *tiles* connected over a network. Each *tile* consists of a *task node* and a FIFO queue for incoming packets. Every *tile* runs in its own thread and blocks on the FIFO. The system is event driven, with two possible types of events: arrival of a packet and events generated by the kernel. The latter is either creation of a packet or modification of the local state.

The reduction engine (i.e. the task manager) evaluates the GPIR bytecode via parallel dispatch of packets requesting computations to other *tiles*.

The Glasgow Parallel Reduction Machine (GPRM) is a Virtual Machine in the sense that it evaluates bytecode; it could equally be considered a runtime library as every instance of the GPRM is compiled based on the source program, and linked with the original source. The GPRM evaluates the bytecode in parallel, as follows:

- On startup, the GPRM creates a thread pool.

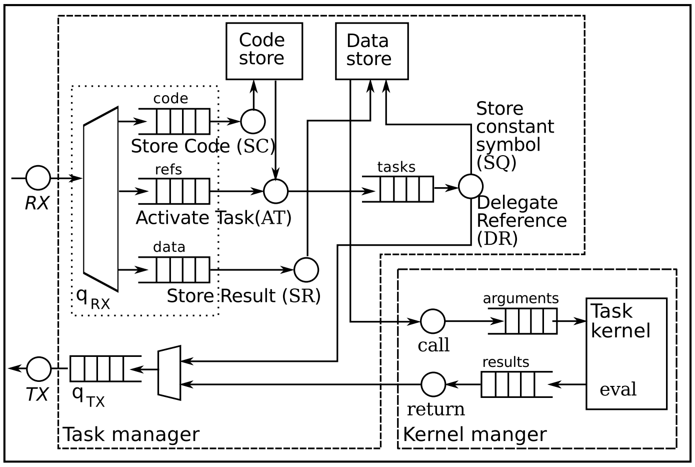


Figure 1: GPRM Task Manager

  - The threads exchange packets via FIFO queues.
  - Each thread runs a *tile*.

- Computations are triggered by the arrival of a *reference* packet, a packet which contains a reference to a *subtask*, i.e. a piece of bytecode representing an S-expression.
  - Each argument in this S-expression is either a reference or a value.
  - References are sent out to other tiles for computation, values are stored.
  - 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.
- The result of the computation is returned to the caller, i.e. the sender of the reference packet.

Rather than the conventional stack used by other virtual machines, we use a random-access *subtask list* to store the state of each parsed subtask. The addresses of the entries in this list (the *subtask records*) are managed using a stack. The reduction engine keeps track of which arguments have been evaluated using the *subtask records*: arrival of a reference packet results in creation of a new subtask record in the subtask list. The address is taken from the *subtask stack*. The process is shown in pseudo-code in Algorithm 1(a). When the computation by the kernel is finished, the process is essentially to dispatch the result and clean up, as shown in Algorithm 1(b).

The combined operation of all reduction engines in all threads results in the parallel reduction of the entire program. Because the communication between the tasks is expressed using a pure functional language, the Church-Rosser theorem [3] guarantees that the parallel evaluation is correct, and we can easily express complex communication patterns between the tasks. The programmer does not need to deal with creating and managing threads: parallel execution is the default and the GPRM manages the thread pool.

Algorithm 1 (a) Subtask allocation and parsing; (b) Result dispatch and clean-up

```
# (a) On receipt of subtask reference
addr = subt_stack.pop()
subt_list[addr]= subt_rec.new()
# caller id is taken from the request packet
subt_list[addr].caller = caller_id
subt_list[addr].operation = bytecode.shift()
for arg in bytecode:
  if arg.kind==reference
   status = requested
   val = nil
   dispatch_ref_packet(subt_addr, arg)
  else
   status = present
   val = arg
  end
  subt_list[addr].args[arg]=(val,status);
end

# (b) After computation by kernel
dispatch_result_packet(caller_id, result)
subtask_stack.push(addr)
```

3 Programming Model

Essentially, the GPRM programming model is one of computational tasks that communicate using a function call mechanism. The function call tree is evaluated in parallel. This mechanism can support various types of parallelism such as data parallelism, reduction, and pipeline parallelism.

3.1 Kernel Tasks and Wrapper Architecture

The computational kernel tasks are written as C++ classes. This means that the end user simply creates classes in the *GPRM::Kernel::*namespace. The only requirement on the class is that it does not instantiate another class in the *GPRM::Kernel::*namespace. Execution of code in these classes follows C++ semantics, including the possibility to create threads etc.

The compiler wraps every user class in a pure functional interface which provides the actual *task* abstraction, essentially by generating a switch/case statement to select methods corresponding to byteword values.

3.2 Communication Code

The communication between the tasks is expressed in a restricted subset of C++11 with parallel evaluation. This communication code is compiled into the Glasgow Parallel Intermediate Representation language (GPIR), a small, S-expression based functional language.

In practice, the communication code is a function in the GPRM:: namespace, called from a (sequential) C++ program. The restrictions are mainly determined by the requirement that the communication code must be pure and functional, so they imply single assignment and no dynamic memory allocation. However, because the wrapper functions around the user classes can actually contain arbitrary C++ code, there are very few other restrictions. Examples are given below.

4 Glasgow Parallel Intermediate Representation Language

The Glasgow Parallel Intermediate Representation (GPIR) is based on the untyped lambda calculus [7], with extensions similar to Scheme’s [14]: numbers, conditionals, lists. GPIR is more regular than Scheme in that every expression must be either a constant, a lambda variable or an operation-operands sequence, i.e. any list must always start with an operation. Furthermore, the GPIR has one additional syntactic construct, the quote ’. Quoting an expression defers evaluation from the Reduction Engine to the task kernel. This *deferred evaluation* is the basic mechanism used in the GPRM to implement control features. The GPIR syntax is given by:

<*s-expr*> S-expression of the form
’(’ operation <expr$_1$*> ... <expr*$_m$*> ’)’*
operation A literal (in practice representing the instance of a class and the method used, e.g. $t_1.m_2$))
<*expr*$_i$> Either an S-expression, or a literal (e.g. *x* or *42*) that is not an operation. Can be quoted, i.e. preceded by a quote, e.g.: $'(t_1.m_1\,'42)$

4.1 Compilation of GPIR into Bytecode

The GPIR compiler converts nested S-expressions into a map of flat S-expressions by substituting a reference for every non-literal expression, and then converts the flat S-expressions into bytecode, essentially by assigning a 64-bit number to every reference and literal. The keys in the map are the memory addresses where the bytecodes are stored. For example,

```
(t1.m2
  (t2.m3 ’42)
  (t3.m4)
)
```

is converted into a map:

$$r_1 \Rightarrow (t_1.m_2\, r_2\, r_3)$$
$$r_2 \Rightarrow (t_2.m_3\,'42)$$
$$r_3 \Rightarrow (t_3.m_4)$$

The GPIR compiler creates a packet which contains the reference to the root of the computation (i.e. r_1 in the example) and sends it to the corresponding tile for evaluation.

4.2 Minimal GPIR Subset

For the purpose of this paper, we consider the GPIR subset consisting of following types of expressions:

4.2.1 Lambda expression

$$(\lambda' x'_1 x_2 ...' x'_n < s\text{–}expr >)$$

All arguments are quoted to defer evaluation, because evaluation of a lambda expression by the task manager would be meaningless as the arguments x_i are not constants or references, so their evaluation is not defined. The compiled bytecode contains a list of flat S-expressions representing the original nested expression. For example, the lamdba expression

$$(\lambda' x' (* (- x' 1)\, (+ x' 1)))$$

is compiled into

$$\begin{aligned} r_1 &\Rightarrow (\lambda' x' r_2' r_3' r_4) \\ r_2 &\Rightarrow (* \, r_3 \, r_4) \\ r_3 &\Rightarrow (- x' 1) \\ r_4 &\Rightarrow (+ x' 1) \end{aligned}$$

4.2.2 Beta reduction expression

$$(\beta < \lambda - expr >< expr_1 >< expr_2 > ... < expr_n >)$$

The beta reduction, i.e. substitution of the lambda variables by their corresponding argument values, is performed at the level of the bytecode. After beta reduction, the bytecode will be evaluated by the Reduction Engine. There are two points worth noting:

- First, the arguments are evaluated in parallel. However, achieving sequential evaluation is very simple: it suffices to convert

  $$(\lambda' x'_1 x_2 ...' x'_n < s\text{–}expr >)$$

  into

  $$(\lambda' x'_1 (\lambda' x'_2 (\lambda \; ...' (\lambda' x'_n < s\text{–}expr >))...))$$

- Second, if the expressions are not quoted, they will be evaluated, i.e. they will return values. However, as GPRM tasks are C++ objects, the methods will return either numerical values (e.g. int or float) or pointers. The wrapper methods return these wrapped in a bytecode container, so that the beta-substitution results in valid GPIR bytecode. However, if the expressions are quoted, then the quoted byteword is obviously valid bytecode; the quote is removed during the substitution, so that the references will be evaluated.

4.2.3 Conditional expression

$$(if < cond\text{–}expr >' < if\text{–}true\text{–}expr >' < if\text{–}false\text{–}expr >)$$

In the *if* expression, if the *if-true* and *if-false* expression are quoted, only the expression corresponding to the value of the condition is evaluated.

4.2.4 List expressions

The list expressions are defined in the usual functional style, based on the empty list and the cons operation.

4.2.5 Expression labels

Any expression in the GPIR can be explicitly labeled and called by its label:

$$(label\,L < expr_1 >)$$
$$(op_2\,L)$$

is equivalent to

$$(op_2 < expr_1 >)$$

4.2.6 Additional expressions

For reasons of efficiency, in practice there is a larger set of GPIR expressions, for example to support sequencing of operations without the need for chained function calls. However, all of these can theoretically be expressed in terms of the above minimal set. For example:

(return <expr>)= (if ’1 ’<expr> ’0)

(begin <expr$_1$><expr$_2$>...<expr$_n$>) =
(β (λ′x$_1$...′x$_n$ ’(return x$_n$))<expr$_1$><expr$_2$>...<expr$_n$>)

(let (assign ’x <x-expr>) ’<in-expr>) =
(β (λ ’x ’<in-expr>) <x-expr>)

4.3 Compiling GPRM C++ Code to GPIR

Code for the GPRM is C++ code. We use the name GPC for the language of the sections of the code that are executed on the GPRM. The GPC compiler separates the task code from the communication code based on the namespace. In this paper, the GPC language is not our primary concern. As a simple example, the following code

```
GPRM::Kernel::Task1 t1;
GPRM::Kernel::Task2 t2;

int GPRM::compute(int v0) {
int v1 = t1.m1(v0);
int v2 = t2.m1(v1);
int v3 = t2.m2(v1);
return t1.m2(v2,v3);
}
```

will be compiled into GPIR as

```
(β
  (λ ’v1
    ’(t1.m2
     (t2.m1 v1)
     (t2.m2 v1)
  ))
 (t1.m1 (ctrl.arg ’0)))
```

where *(ctrl.arg ’0)* is a method of the *ctrl* kernel to handle the arguments.

The compilation is conceptually straightforward: because of the restrictions imposed on the communication language, its abstract syntax is that of a functional language, and hence it can be compiled easily into GPIR. The object instance declarations are mapped to tiles based on the dependencies in the call tree: tasks that depend on one another can’t run in parallel and hence can be mapped onto the same tile.

4.4 Build Process

The object instances, together with system information – in particular the maximum number of hardware threads in the system – are used to dimension the thread pool. Furthermore, the compiler analyses all the classes and methods used in the task description code and maps them to numeric constants. These constants are used in the 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 code and the source code for the task classes and compiled into a library. The original program is adapted by adding an instance declaration for the GPRM and replacing the call to the task code (GPRM::compute in the above example) by a call to the GPRM’s run method, with the name of the compiled bytecode file as an argument. The program is then compiled and linked with the GPRM library.

4.5 Scheduling of Work on Threads

Dynamic scheduling of work on threads is performed using a control kernel: whereas a task *(t1.m1 ...)* will be executed on a compile-time assigned thread, using the *ctrl.run* service, a task can be scheduled on a run-time computed thread: *(ctrl.run ’(t1.m1 ...) (thread-id-expression)).* The *thread-id-expression* can be a compile time constant, a run-time computed value or a value returned at run time by a dynamic scheduler.

4.5.1 Deferring Evaluation

The mechanism used to perform the scheduling is based on the GPRM's ability to defer evaluation to the kernels by quoting. The bytecode for the above example is:

$$
\begin{aligned}
&r_1 \Rightarrow (ctrl.run'r_2\, r_3)\\
&r_2 \Rightarrow (s_1.m_1 \ldots)\\
&r_3 \Rightarrow (thread\text{–}id\text{–}expression)
\end{aligned}
$$

The evaluation of r_1 results in two argument values: the quoted reference $'r_2$ and the number *thread-id.*

4.5.2 Restarting Evaluation

The discussion in Section4.1 glossed over the actual structure of the reference byteword: apart from containing the address of the corresponding bytecode, it also contains the identifier of the tile on which this code should be run, in other words the reference is a tuple of two integers:

$$r = (code\text{–}addr, tile\text{–}id)$$

The *ctrl.run* kernel substitutes the compile-time computed *tile-id* for r_2 by the run-time computed value, i.e. *thread-id*. It then removes the quote and restarts the evaluation. As a result, a reference packet is dispatched to the run-time computed tile.

5 Example: Parallel Merge Sort

To illustrate the programming model, we consider an implementation of the Merge Sort algorithm which is a good example of parallel reduction. We will discuss different aspects of this algorithm in detail. In the next section, we will illustrate the power of the GPRM for task management using a pointer chasing algorithm.

5.1 GPRM Implementation

We use two tasks, *leaf* and *stem*, implemented as methods of a MergeSort class. The GPC task composition code uses a recursive tree:

```
GPRM::Kernel::MergeSort ms;

void ms_rec(int n,int nmax, int* a) {
  if (n>=nmax) {
    ms.leaf(n,a);
  } else {
    ms.stem(
      ms_rec(2*n,nmax),
      ms_rec(2*n+1,nmax),
      a);
  }
}
int* GPRM::merge_sort (int* a) {
  ms_rec(1,NUM_THREADS,a);
  return a;
}
```

In GPIR, this becomes:

```
(β
 (λ 'f 'n 'nmax 'a (β f n nmax a))
 (λ 'n 'nmax 'a
       (if (>= n nmax)
          '(ctrl.run
            '(ms.leaf n a)
             n)
          '(ctrl.run
            '(ms.stem
              (β f (* '2 n) nmax)
              (β f (+ '1 (* '2 n)) nmax) )
              a)
            n)
          )
     )
 '1  NUM_THREADS (ctrl.reg '0))
```

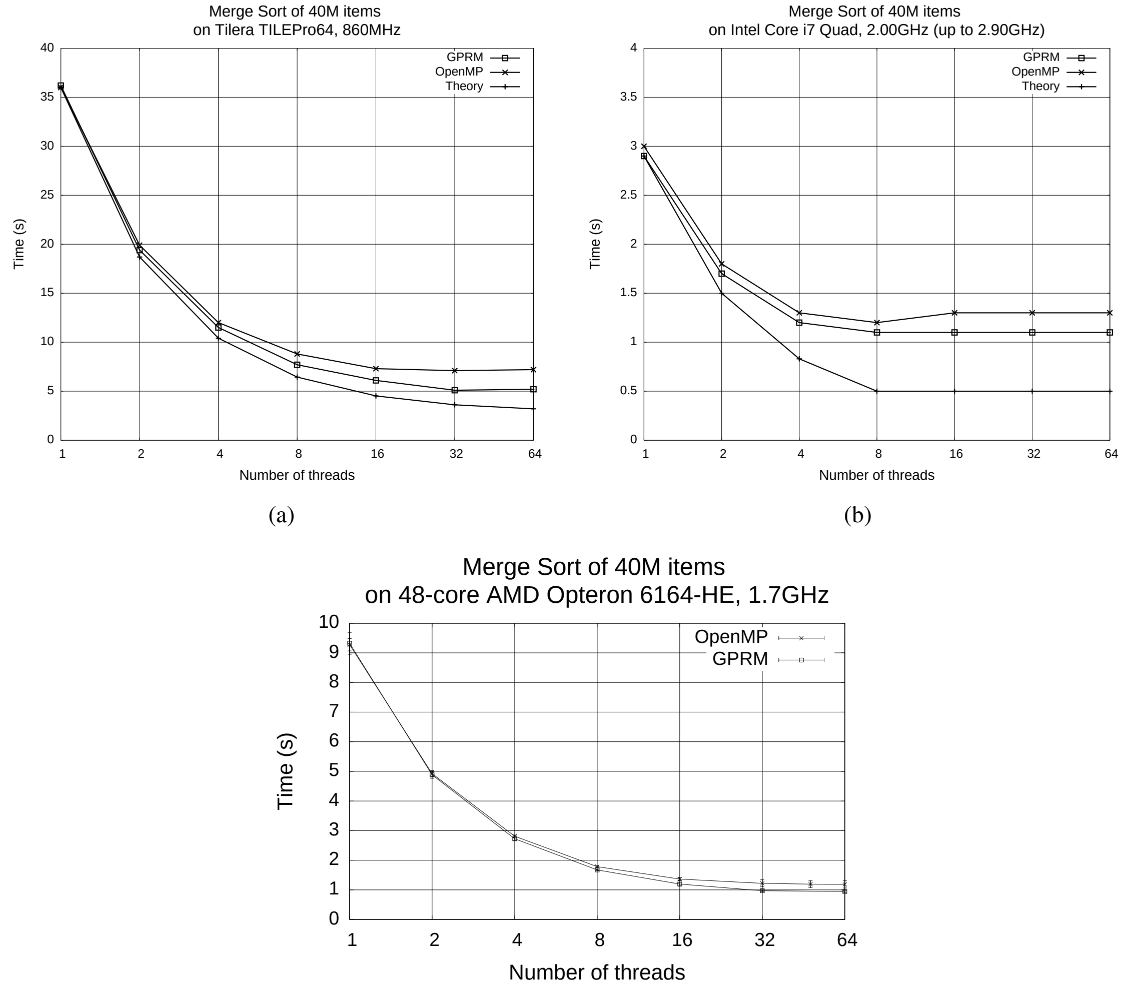
For simplicity, we omitted the GPIR code for argument handling. Note the explicit run-time thread allocation, which allocates the computations *ms.leaf* and *ms.stem* to the GPRM nodes with address *n*.

5.2 Evaluation and Discussion

5.2.1 Parallel Execution Time

Figure 2 shows the performance of the GPRM compared to a high-performance OpenMP implementation [10] for sorting an array of 40M 32-bit integers. We used the Tilera TILEPro64 64-core processor [1], as well as on two more conventional platforms: a quad-core Intel Core i7-2630QM running at 2 GHz, with Turbo Boost up to 2.9 GHz, and a 4-socket, 12-core AMD Opteron 6164 HE, running at 1.7 GHz. Note that the Tilera processor runs at only 860MHz. Differences in CPU speed and memory specifications account for the difference in performance at low numbers of threads. The theoretical model shown in Figure 2 is based on the fact that the execution time for a single-threaded Merge Sort algorithm is $k.n.log(n)$, in which k is the time required for each operation and n is the number of elements. Therefore, the execution time on e.g. 2 cores will be $k\frac{n}{2}\log\frac{n}{2}+kn$. On the TILEPro64, the maximum number of available cores is 63 (one is required for the PCIe link between the card and host), which would result in a critical path and thus some irregularity at the end of the plot, compared to the theoretical model. Also, with the hyper-threading technology, the number of available cores on the Intel Core i7 platform is 8 which restricts the performance gain at higher numbers of threads.

[1] http://www.tilera.com/products/processors/TILEPRO64



(a) (b)

(c)

Figure 2: Performance of merge sort in GPRM and OpenMP on (a) Tilera TILEPro64, (b) Intel Core i7 Quad CPU and (c) AMD Opteron 6164-HE

It is clear from these results that the GPRM is very competitive on all platforms, indeed the GPRM slightly outperforms the OpenMP version for larger numbers of threads. Furthermore, the example illustrates how easy it is to create complex parallel programs using this approach. The corresponding OpenMP code (see Appendix) uses parallel OpenMP sections to assign recursive calls to threads. It is noticeably longer and more complex than the GPC code.

5.2.2 GPRM Overhead

As the GPRM is a virtual machine interpreting bytecode, it introduces some overhead. To explore this we varied the array size from 4K words to 400M words. In this experiment, we used 32 threads on the 48-core AMD system. As can be seen from Figure 3, for very small arrays (4K) the GPRM code is 2.4× slower than the OpenMP code. For 4M, the performance is the same and for 400M, the GPRM is 1.4× faster. We are confident that we can reduce the overhead of the GPRM as up to now no special care was taken to optimise the performance of the reduction engine.

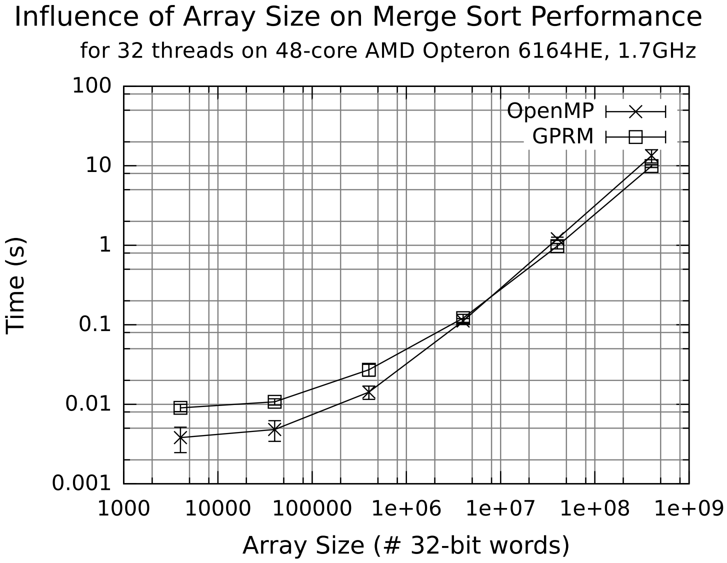


Figure 3: Influence of Array Size on Performance

5.2.3 Impact of Caching

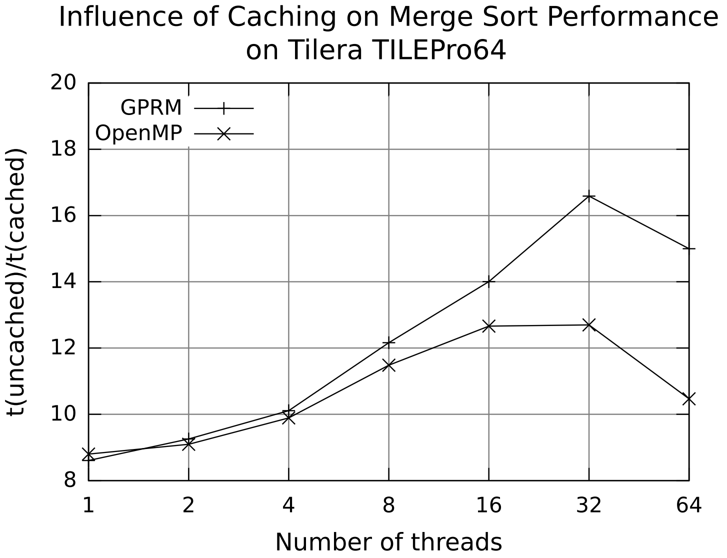


Figure 4: Influence of Caching on performance on Tilera TILEPro64

An important point is the effect of the memory architecture on the performance: typically, the off-chip memory (DRAM) has limited parallelism, so any parallelism provided by the cores could be undone by the memory bottleneck. To evaluate this, we used the TILEPro64 with caching disabled. The Tilera platforms provide fine-grained control over caches and memory, making them ideal for this type of experiments [18]. Figure 4 shows the ratio of the execution time for uncached to cached execution: for the GPRM the performance without caching is 8× worse for a single thread, and 17× for large numbers of threads. We also see that the GPRM makes better use of the caches than OpenMP, which results in better performance for larger numbers of threads.

6 Example2: Parallel pointer chasing

As another example, we compare the performance of our framework with different OpenMP implementations of a parallel pointer chasing algorithm. In [2], pointer chasing algorithm is stated as one of the motivations behind tasking implementation in OpenMP. We use two different approaches that are used in that paper to show how they scale compared to the GPRM implementation. The first OpenMP approach uses the `single nowait` construct inside a `parallel` region. Each thread needs to traverse the whole list and determine in each step whether another thread processed the current element or not. In order to do some work at each element, we have used the Ackermann function [12]. In the second OpenMP approach, a single thread traverses the list and creates one task for each element.

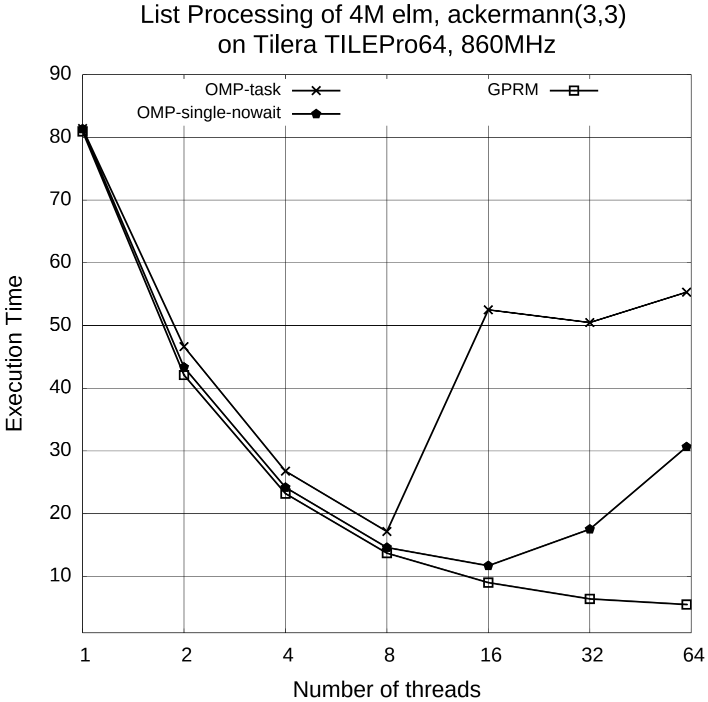


Figure 5: Performance of list processing in GPRM and two OpenMP implementation on the Tilera TILEPro64

Figure 5 shows that GPRM scales very well when the number of threads becomes larger. It is also evident that the OpenMP tasking approach performs poorly when so many tasks are created. Although in the GPRM implementation, every thread traverses the whole list, it does not need to check whether other threads have processed the element or not. The reason is that the whole work is divided between threads in such a way that there is no contention between them. For example the thread k is responsible to process the element k, NTH+k, 2NTH+k, and so on.

7 Conclusion and Future Work

We have presented a new approach towards many-core programming, based on a functional task composition language with parallel reduction, implemented as the Glasgow Parallel Reduction Machine. We have shown that our approach slightly outperforms OpenMP for the merge sort algorithm on large arrays. For the pointer chasing algorithm, we have demonstrated that out approach works so much better than the corresponding OpenMP implementations. It does not have the high cost of the `single` construct, or the overhead of having so many fine-grained tasks. We are currently working on a data-parallel version of the GPRM, for use on GPUs and, eventually, FPGAs. In this way we aim to create a unified programming framework for heterogeneous many-core systems.

Acknowledgement We thank Prof. Hameed of the University of Aizu, Japan, for kindly granting us the use of the AMD system for our experiments. Ashkan Tousimojarad would like to thank SICSA (Scottish Informatics and Computer Science Alliance) for supporting his research.

References

[1] Joe Armstrong (2007): *Programming ERLANG: software for a concurrent world*. Pragmatic programmers, Pragmatic Bookshelf, doi:10.1017/S0956796809007163. Available at `http://www.oreilly.com/catalog/`.

[2] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan & Guansong Zhang (2009): *The design of openmp tasks*. *Parallel and Distributed Systems, IEEE Transactions on* 20(3), pp. 404–418, doi:10.1109/TPDS.2008.105.

[3] A. Church & J.B. Rosser (1936): *Some properties of conversion*. *Transactions of the American Mathematical Society* 39(3), pp. 472–482, doi:10.1090/S0002-9947-1936-1501858-0.

[4] P Harrison & Mike Reeve (1987): *The parallel graph reduction machine, Alice*. In: *Graph Reduction*, Springer, pp. 181–202, doi:10.1007/3-540-18420-1_55.

[5] Charles E Leiserson (2010): *The Cilk++ concurrency platform*. *The Journal of Supercomputing* 51(3), pp. 244–257, doi:10.1145/1629911.1630048.

[6] Josep M Perez, Rosa M Badia & Jesus Labarta (2008): *A dependency-aware task-based programming environment for multi-core architectures*. In: *Cluster Computing, 2008 IEEE International Conference on*, IEEE, pp. 142–151, doi:10.1109/CLUSTR.2008.4663765.

[7] B.C. Pierce (2002): *Types and programming languages*. MIT press.

[8] Artur Podobas & Mats Brorsson (2010): *A comparison of some recent task-based parallel programming models*. In: *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers,(MULTIPROG'2010), Jan 2010, Pisa*.

[9] R.F. Pointon, P.W. Trinder & H.W. Loidl (2001): *The design and implementation of Glasgow Distributed Haskell*. *Implementation of Functional Languages*, pp. 53–70, doi:10.1.1.20.3631.

[10] A. Radenski (2011): *Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs*. In: *Proc. PDPTA'11, the 2011 international conference of parallel and distributed processing technique and applications*, CSREA press, pp. 367–373, doi:10.1.1.217.7866.

[11] James Reinders (2010): *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc.

[12] Yngve Sundblad (1971): *The Ackermann function. a theoretical, computational, and formula manipulative study*. *BIT Numerical Mathematics* 11(1), pp. 107–119, doi:10.1007/BF01935330.

[13] SMP Superscalar (2008): *User's Manual, Version 2.0*. *Barcelona Supercomputing Center*.

[14] Gerald Jay Sussman & Guy L Steele Jr. (1975): *Scheme: An interpreter for extended lambda calculus*. In: *MEMO 349, MIT AI LAB*, doi:10.1.1.128.80.

[15] W. Thies, M. Karczmarek & S. Amarasinghe (2002): *StreamIt: A language for streaming applications*. In: *Compiler Construction*, Springer, pp. 49–84, doi:10.1007/3-540-45937-5_14.

[16] Willem Gerard Vree & Universiteit van Amsterdam (1989): *Design considerations for a parallel reduction machine*. Sneldruk Enschede.

[17] M. Weiland (2007): *Chapel, Fortress and X10: novel languages for HPC*. *The University of Edinburgh, Tech. Rep., October*.

[18] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown & Anant Agarwal (2007): *On-chip interconnection architecture of the tile processor*. *Micro, IEEE* 27(5), pp. 15–31, doi:10.1109/MM.2007.89.

Appendix 1. OpenMP Code for Merge Sort

Algorithm 2 Parallel Merge Sort using OpenMP sections [10]

```
#include <omp.h>

int mergesort_serial(int* input,
  int* scratch, int size);

void merge( int* input1,
  int size1, int* input2,
  int size2, int* scratch);

int mergesort_parallel_omp (int* input,
  int* scratch, int size, int threads) {
     if (threads == 1) {
      int r = mergesort_serial(input,
           scratch, size);
      return r;
     }
     else {
#pragma omp parallel sections
      {
#pragma omp section
       {
        mergesort_parallel_omp(input,
             scratch, size/2, threads/2);
       }
#pragma omp section
       {
        mergesort_parallel_omp(input+size/2,
             scratch+size/2, size-size/2,
             threads-threads/2);
       }
      }
      merge(input, size/2,
       input+size/2, size-size/2, scratch);
    }
  return 0;
}

int main() {
    omp_set_nested(1);
    omp_set_num_threads(2);
    int* array1 = new int[ARRAY_SZ];
    int* scratch1 = new int[ARRAY_SZ];
    mergesort_parallel_omp(array1,
      scratch1, ARRAY_SZ, NUM_THREADS);
  return 0;
}
```