A Toolchain for Deterministic Parallelism on an Embedded Bare-metal Platform

Kenelm Louetsi, Christophe Negre, David Parello

A Toolchain for Deterministic Parallelism on an Embedded Bare-metal Platform

1st Kenelm Louetsi
DALI Univ. Perpignan
Perpignan, France
LIRMM Univ. Montpellier
Montpellier, France
kenelm.louetsi@univ-perp.fr

2nd Christophe Negre
DALI Univ. Perpignan
Perpignan, France
LIRMM Univ. Montpellier
Montpellier, France
christophe.negre@univ-perp.fr

3rd David Parello
DALI Univ. Perpignan
Perpignan, France
LIRMM Univ. Montpellier
Montpellier, France
david.parello@univ-perp.fr

***Abstract*—Bare-metal platforms, as well as many other platforms, can make great gains in performance through parallel programs. However, running such kind of programs is often harder on bare-metal platforms. This leaves the management of parallelism on bare-metal platforms under the purview of the programmer, thus increasing the program complexity. Furthermore, this also makes the program specific to the target bare-metal platform once ported. This paper presents a set of tools used to build and execute parallel programs on custom bare-metal platforms as painlessly as possible. The proposed toolchain behaves like a regular compilation toolchain, taking standard C OpenMP code as input and building platform-specific, parallel bare-metal binaries as output. We evaluated the behavior of standard C OpenMP code executed on our platform both in terms of correctness and in terms of speedup. Our observations show that it is possible to preserve the performance speedup of parallel execution on a bare-metal platform, without having to modify the program source code. Furthermore, these changes allow programs to exhibit some degree of determinism, which is apropos for bare-metal platforms.**

***Index Terms*—OpenMP, parallelism, compilation, bare-metal, embedded, multicore**

I. Introduction

Embedded systems require performance along with safety, security and energy efficiency. This concerns for among others, IOT, nomadic devices, autonomous vehicles, etc. Increases in performance used to be easily provided through Moore's law hardware speedup. With its slowdown [11], today's hardware performance is mostly achieved by exploiting parallelism. This is true for embedded critical systems, which have to balance this performance against safety. These systems often cannot exploit the full performance of parallelism, as it makes safety harder to guarantee.

Increasing the parallelism of a program can be achieved either through local parallelism or by distant parallelism. Local parallelism consists in concurrently executing instructions which are close inside the program code, but it is known to be severely bounded [13] and non-deterministic. Distant parallelism is obtained by having threads executing distant instructions working concurrently on independent data, it is known to be unbounded [13]. But this requires synchronisation elements to handle concurrent access on shared data which are left under the purview of an overarching program, such as an operation system or an hypervisor, which will guarantee the smooth operation of the system.

Developing a parallel program is simple on an embedded device with an operating system since it handles parallel synchronisation, but this results in a loss of performance and a loss of determinism. Another strategy would be to develop the parallel program to run on a bare-metal platform: in this case the management of parallelism is left under the purview of the programmer, increasing the program complexity. Furthermore, this also makes the program specific to the target bare-metal platform once ported.

In this paper we present a modification of the fork-join model of parallelism for embedded bare-metal platforms. This modified model of parallelism exhibit deterministic properties useful for facilitating the programing of bare-metal platforms. This model is implemented through a set of modifications to open-source tools for compiling and running parallel programs on a bare-metal platform. The implementation is targeted towards a specific platform, a RISC multicore processor stripped-down of most of the usual hardware necessary for supporting parallelism. Our goal is twofold, first is to show that it is possible to adapt a "standard" shared-memory parallel program to a platform without explicit synchronisation support by changing the program expected memory model. Second is to simplify the development of parallel programs on bare-metal platforms through the use of open-source tools, without sacrificing performance and safety. The proposed tools take regular C/C++ programs that use a subset of OpenMP to manage parallelism. We chose OpenMP since it is one of the most popular way of parallelising a program, as it ensures the portability of a program. We propose a model of OpenMP-like parallelism, where some OpenMP-like pragma are exposed to the programmer, but with a private memory model, where each thread has a private copy of all program data and work independently from the other threads. Synchronization is only possible between "parent" and "child" threads through forking and joining instructions. This model of parallelism only applies to data-parallel programs, but simplifies the thread management and guaranties some determinism of execution.

In summary, the contributions of the paper are the following:

- a model of parallelism for deterministic execution on bare-metal platforms,
- a toolchain for implementing the model of parallelism,
- experiments to characterize the model.

Organization of the paper. In Section II we discuss the problematic involved in the development of parallel programs on a bare-metal platform. We then present in Section III the model of parallelism used in the proposed tools along with our implementation of a subset of OpenMP's API. In Section IV we present the platform and the set of parallel benchmarks we compiled for our experiments. In Section V we present experimental results, then close with a few concluding remarks in Section VII.

II. OpenMP on bare-metal

Adapting OpenMP for bare-metal in a platform-agnostic way means, in practical terms, that a regular (non-bare-metal) parallel OpenMP (C) program should be compilable for, and executable to a set of bare-metal targets, with only the minimum possible amount of platform-specific code required (e.g. low-level HW access, etc.).

OpenMP uses the *fork-join* model of parallelism [9] [22], where one parent thread, upon entering a parallel region, will dispatch (*fork*) shared work to a team of concurrent threads. Once the work is done, all threads in the team are terminated (*join*) and the parent alone resumes the sequential execution.

As a standard [7], a typical implementation of OpenMP consists of a set of parallel directives and a runtime library. An OpenMP directive indicates how to compile part of a program in compliance with OpenMP API. In practice, a targeted section of code is marked by a special compiler pragma, which specifies the OpenMP directive. An OpenMP directive itself can convey various data to the compiler. When this data leads to the production of implementation code or impacts how the user code is executed, the lexical scope of the directive is called a construct. These special directives are embedded inside the program source code, processed during the compilation and transformed to calls to the runtime library. Said library is then dynamically linked by the OS during the program execution. The challenge of using OpenMP on a bare-metal platform resides in that, most of the necessary parallelism management operations are done by the operating system, which is not always present in bare-metal.

Considering this, the task of managing parallelism is left to the program itself. This adds all the complexity of having to manage parallelism, on top on the program initial purpose, making it harder to implement, and also impacting performance. Furthermore, to improve performance through parallelism, the code which controls the parallelism should also be effective itself [1].

Indeed, parallel performance is often constrained by the need to share the same resources across multiple parallel entities, hence the need for efficient control mechanisms. This is a hard problem [15], which usually requires synchronisations to resolve. For bare-metal these synchronisations have to be implemented on a per-platform basis. Consequently, to exploit a *fork-join* model of parallelism, bare-metal programs have to either embed their own platform-specific parallelism management code, or run under another dedicated program. Both these options incur potential losses of performance, be it from unoptimized parallel management code or through the dependency upon another program.

To alleviate this loss of performance, one option is to modify the implemented model of parallelism so as to reduce the number of unavoidable parallelism management operations as much as possible. We propose one such modification, which allows the removal of most of the explicit synchronisation operations, while keeping an OpenMP-like model of parallelism, and without changing a program original logic flow. To achieve this, we impose specific constraints over the OpenMP model of parallelism for bare-metal platforms. These constraints are expressed in our implementation of OpenMP low-level threading back-end, which stays platform agnostic. Associated with these constraints is also some platform specific code, for the basic hardware-access functionalities.

In summary, the challenge of using OpenMP on a bare-metal platform resides in that, most of the necessary parallelism management operations are traditionally done by the operating system or some kind of hypervisor, of which the existence is not a guarantee on bare-metal.

Proposed strategy. This paper details how a constrained variant of OpenMP's model of parallelism can be exploited on bare-metal platforms to inherently remove the need for explicit synchronisations, all the while keeping some of the performance gains from parallelism. In Section III, we present a library implementing our constrained model of parallelism through a subset of OpenMP. This library produces multi-threaded code for a bare-metal platform described in Section IV-A. As shown in [3], a large proportion of parallel codes can be run with the proposed strategy through isolated thread memory and in a deterministic manner. Thus, the experiments described in this work have been selected in accordance with the proposed strategy, so as to clearly display the relevance of our approach.

III. Bare-metal OpenMP toolchain

To produce relevant experiments for our model of parallelism, our target platform must exhibit certain characteristics. The hardware must be as minimal as possible, while still supporting some degree of multithreading. For this paper experimental platform, it means a bare-metal environment with strict in-order execution, no speculation, no caches and hardware support for multithreading, with memory-isolated threads. Described below is how we tailored our toolchain for this kind of target.

A. OpenMP support

Representative programs for the kind of embedded platform we target in this work are well suited to regular parallelism. This turned us to mostly regular data-parallel workloads, rather than task parallel ones. As such, we left OpenMP explicit task

manipulation constructs for future works. Furthermore, one of our aim is to dispense with any explicit synchronisations through our model of parallelism. Thus, we also did not need OpenMP synchronisation constructs. This left us with OpenMP work-sharing constructs, which we discuss in this paper.

B. OpenMP execution

OpenMP uses the fork-join model of parallelism. An OpenMP program starts as a single parent thread. This parent executes sequentially until it reaches a `# pragma omp parallel` directive. From there, a team of threads comprised of the parent thread, and any required number of other threads will execute in parallel a set of tasks derived from the code inside the parallel construct. Most implementation of OpenMP do this by outlining the code inside the parallel construct into compiler-generated functions. At the end of the parallel construct, all the threads synchronise on a barrier. Then, only the parent thread resume its sequential execution, until it reaches another construct or the program ends. This execution model relies on the compiler to create a set of tasks from the code inside the `parallel` construct, and on platform-specific code necessary for the dynamic execution of those tasks. In summary, the OpenMP standard can be implemented through the combination of a code translator and a runtime library. The toolchain presented in this paper is based on the implementation of OpenMP version 4.5 by LLVM. We chose LLVM's implementation for its extensive documentation and its simple frontend compilation approach, rather than a source-to-source such as in Nanos Mercurium [5] or OpenUH [19]. GCC's implementation of the OpenMP specification was not chosen because of its strong integration into GCC's compilation pipeline, which we did not need.

LLVM implements four distinct types of OpenMP runtimes, the host runtime (*libomp*), the target offloading runtime (*libomptarget*), the target offloading plugin (*libomptarget.rtl.XXXX*), and the target device runtime (*libomptarget-ARCH-SUBARCH.bc*). The *libomptarget** runtimes are mainly used for supporting *libomp* heterogeneous devices offloading capabilities. It is the (*libomp*) host runtime that implements the OpenMP specification, with the other runtimes for offloading support. Thereafter, we will only discuss the (*libomp*) host runtime. (*libomp*) utilizes the Pthreads library to instantiate the outlined functions dynamically. To allow tasks to communicate, Pthreads necessitate multiple layers of abstraction, which means notable added overhead [8]. Additionally, our model of parallelism manages synchronisations in a very specific way, which is not directly compatible with OpenMP synchronisation constructs.

For these reasons, we do not use LLVM's *libomp* library, but instead we re-designed and re-implemented the runtime around our model of parallelism from scratch. We implemented our target subset of OpenMP API in a lightweight custom library which preserves the compiler interface, the *liblbmcomp* runtime library. A common way for OpenMP implementations to reduce dynamic thread creation overhead on entering a parallel region, is to do some form of thread parking [18]. This optimization allows threads to be "docked" once a parallel region is exited. These threads can then be quickly re-activated if needed. As our target platform is bare-metal, and its parallelism comes from hardware threads, we had to use something other than Pthreads to manage threads. We implemented a custom library, the *liblbmcio*, to pilot our hardware threads indistinguishable from software threads for the *liblbmcomp* runtime library.

C. Synchronization/Parallelism Model

As we target hardware supported threads (known as harts) with private memory, we can do a fixed allocation of the parent and child threads over all the harts. This allows us to guarantee absolute memory isolation between each thread, and so avoid common concurrent access problems. Effectively, both program and libraries are loaded into all harts code memory on boot. Each hart then executes the same initial configuration routines from the libraries. The harts can then busy-wait to be sent the context for a parallel work, which is a full image of the parent thread data memory. To reduce the number of data transmission, harts can also start to execute the sequential part of the program, until the first parallel region is reached. This ensures that all harts reach the parallel region with the same memory state, thus canceling the need to synchronise parent and child memory state through data transfers. Once a parallel region is reached, all harts but the one hosting the parent thread start busy-waiting. The parent thread will enter alone the parallel region; thereupon it will set to itself a part of the data to process, then inform one child thread, its "child", of what data is left to be processed. The "child" thread will carry out the same operation for its own child, and so on until all threads have work to do. When a thread is done, it will wait until its child signals that it itself is done. Once the signal received, it can then retrieve the work done by its child, afterwards which it will signal its own parent that it itself is done and terminates.

Figure 1 illustrates the deployment of parallel threads following the proposed parent-child scheme. The strict hierarchy between threads over the attribution of work, ensure that no discrete work can belong to two threads at the same time. Furthermore inter-thread communications only go through a one-way, parent-to-child channel. Over the kind of channel, the parent is the parent and its child the child. This allows us to ensure the only potential concurrent access conflicts are between a parent and its child, where the parent has then clear superiority. With these properties we can enforce a strict degree of hierarchy and isolation between each thread, thus forfeiting the need for locking mechanisms which would otherwise be needed.

By combining these two libraries, we can build OpenMP programs over our platform hardware threads, without having any hardware-management code exposed to said programs. This allows us to build bare-metal platform compatible OpenMP programs with no source code modifications.

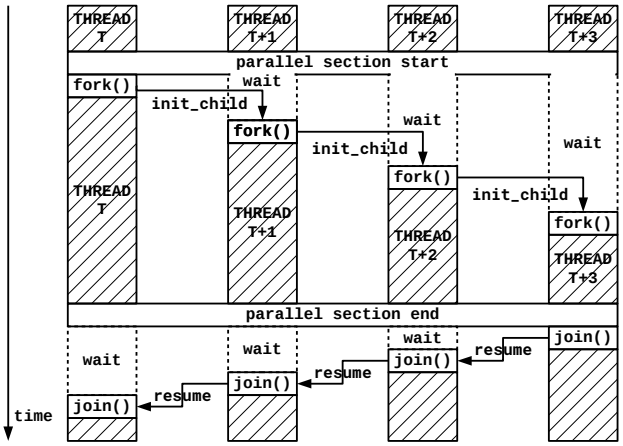

Fig. 1. Threads fork-join

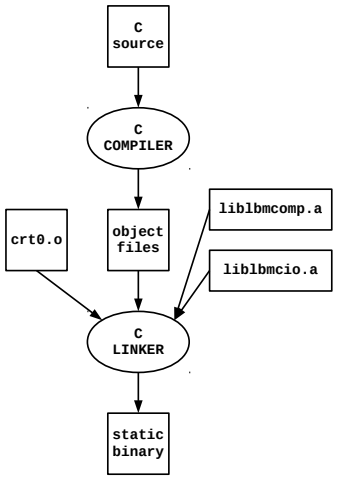

Fig. 2. Fork/join static parallel binary build stages

D. Determinism

An added benefit of our model of parallelism is the introduction of some degrees of determinism. Indeed, inter-thread communications follow a strict hierarchy determined during compilation, with said compilation being static for all parts of the program on bare-metal.

1) Execution-flow determinism: This leads to the program having a fixed parallel execution-flow, which means that from one execution to the next, the sequence of parallel control routines calls will stay the same, whatever the processed data. This allows the exhibition and potential subsequent exploitation of determinism in the sequence of parallel communications/control routines in a parallel OpenMP program, which otherwise would need explicit synchronisation mechanisms (e.g. mutex, barriers, etc.) to enforce any degree of parallelism.

2) Time determinism: Timings of explicit parallel synchronisation mechanisms are notoriously difficult to measure without external (OS or HW) support, with the ever-present possibility of deadlocks [10] [31]. As such they are often part of the longest path [12] of a program, and can even prevent the computation of a bounded WCET (in the case of deadlocks). Hence, their removal allows for an easier computation of a program longest path, as well as its WCET.

E. Generation and execution

Building the binary In a standard OpenMP C code, all OpenMP directives will be processed at compile time and will generate the standard OpenMP entrypoints into the program object file.

Figure 2 illustrates the different stages of the building of a bare-metal binary with our toolchain. Our first main difference with a "traditional" non-bare-metal compilation flow occurs at link time; we enforce static linking for all dependencies of the program as it targets bare-metal. We then link against *liblbmcomp*, instead of against a "regular" OpenMP library (e.g. LLVM's libomp). Obviously, we also provide the necessary utilities for bare-metal C code execution such as the memory-mapping information through the linker script, the C runtime initialization tools (crtX.o) and basic I/O capabilities through *liblbmcio*. Once the program ELF file has been generated, we strip and flatten it into a custom binary.

Bootstrapping Our flat binary is loaded in memory by an auxiliary control processor (ARM, sim, etc.) after the core bootloading code. In a typical parallel execution, a parent thread will make some pre-computations, then pass the parallel context and other relevant data to its children. Since our memory model only allows passage of information through copy, inter-thread communications always have a high cost. Due to this, we favored thread-local work as much as possible, to the point of choosing to duplicate some work rather than wait for its remote transmission and completion. Hence, each hart/thread is pre-loaded with a copy of the program before the boot, and all will execute the sequential pre-computation identically. Upon arriving at the entry of the parallel section, all threads but the parent one will stop and wait. Then, the parent thread will run the parallel initialization routines and set the work context for the poll of threads. Despite this duplication, the overall performance gains from parallelism remain. As such, for this work we focused on highly parallel applications, with as much intra-thread/thread-local work and as little inter-thread communications as is possible.

Execution Concerning the execution with our model of parallelism, it is identical to a regular OpenMP execution at a high level. The program will run sequentially on a parent thread upon coming upon the start of a parallel section. The parent thread will then proceed to initialize and start the thread pool for parallel work. Once the parallel work is completed on all threads the parent thread will resume sequential execution, until either encountering another parallel

section or the program end.

Tools For actual software used, the toolchain is based around open-source tools. Since we build from a standard x86_64 machine to a bare-metal RISC-V on an FPGA, we need the relevant cross-compilation capabilities. Those were provided by the LLVM project tools and libraries. The relevant RISC-V nano-libc came from the riscv-gnu-gcc project and we rewrote enough of libgloss to be able to setup a working C runtime environment on bare-metal. These tools, when combined allow for the building of C programs, statically linked with our *liblbmcomp* and *liblbmcio*.

IV. Experimental environment

In this section we describe the experimental environment for the evaluation of the proposed toolchain. First we describe the main features of the bare-metal platform. Afterwards, we describe the set of benchmarks used for our experiments.

A. Bare-metal target platform

Our target platform is the **L**ittle **B**ig **M**icro**C**ontroller (LBMC). It is a multicore computing platform proposed by Goossens et al. [14] composed of a highly scalable grouping of multithreaded RISC cores. Most of its architectural characteristics match the requirements of our model of parallelism: no speculation, no caches, and some multithreading support with isolated per-hart memory.

The number of harts per core is configurable, and the cores implement the rv32im extension of the RISC-V ISA [27]. As this target does not support floating point operations, we limit ourselves to use integers only in our experiments. While using soft-float libraries is an option, we felt it would not meaningfully contribute to the goal of this paper.

For our experiments, the LBMC platform is implemented on an FPGA, which allows us to execute the actual process of programming an embedded bare-metal platform as close as possible.

All experiments have been realized on the LBMC in different configurations described in Table I.

#HARTS	IMEM		DMEM	
	total	per-hart	total	per-hart
2	32ko	16ko	32ko	16ko
4	32ko	8ko	32ko	8ko
8	32ko	4ko	32ko	4ko

TABLE I
LBMC core specifications

These configurations have been deployed on an FPGA (Xilinx ZYNQ XC7Z020-1CLG400C) to provide physical platform for our experiments, rather than limiting ourselves to simulation. LBMC's cores were clocked at 50MHz, used only programming logic and were controlled through an embedded ARM processor.

B. Benchmarks

The parallel codes which could be run on a bare-metal platform with the proposed toolchain should satisfy several constraints. At first they should fit in the proposed parallelism model: they should have parallel section working independently on private data. This means that no synchronisation is done in the middle of a parallel section. This is the most important constraint since it is a restriction at the algorithmic level. Secondly, the constraints inherent to LBMC integer-only operations and hart-private memory architecture, the programs selected for benchmarking had to share a common profile: integer (or fixed-point) dataset, no dynamic memory management, few dependencies outside the libc, no reentrancy, and high parallelism.

These constraints prevent the use of popular parallel benchmark suites such as PARSEC [6], SPLASH [28] and NPB [4]. Indeed these benchmark suites were designed mainly to evaluate the performance of HPC platforms, and were not created with embedded bare-metal platforms in mind. In particular, they include system calls, unpredicatble parallel synchronisations and floating-point operations. As such, it is exceedingly difficult to source OpenMP parallel codes specifically tuned for an embedded platform such as in [17], and even more so for our bare-metal platform.

Our solution was to select specific experiments: some were selected and recoded from ([6], [28] [4] and [16]). Other programs are reimplemented directly from variants of the algorithms used in these benchmark suites, in order to avoid floating point arithmetic or to have a parallel section which fits the requirement of the proposed parallelism model.

Algorithm	Input size	Overall cost	Seq. cost	Par. cost
AES-OFB	n blocks	$n+1$ block-ciph.	1	$\frac{n}{k}$
edge detec.	n^2 pixels	$O(n^2)$ add/mul	0	$O(\frac{n^2}{k})$
LZSS	n char.	$O(n)$ comp.	0	$O(\frac{n}{k})$
qsort	n int	$O(n\log_2(n))$ comp.	$\log_2(k)n$	$O(\frac{n}{k}\log(\frac{n}{k}))$
merge sort	n int	$O(n\log_2(n))$ comp.	$\log_2(k)n$	$O(\frac{n}{k}\log(\frac{n}{k}))$
mat. vector	n^2+n int	$O(n^2)$ add/mul	0	$O(\frac{n^2}{k})$
mat. mult.	$2n^2$ int	$O(n^3)$ add/mul	0	$O(\frac{n^3}{k})$
mat. expo	n^2 int	$O(e\times n^3)$ add/mul	0	$O(e\times\frac{n^3}{k})$
polymult.	n int	$O(n^2)$ add/mul	0	$O(\frac{n^2}{k})$
NTT	n int	$O(n\log(n))$ add/mul	$\log_2(k)n$	$O(\frac{n}{k}\log(\frac{n}{k}))$

TABLE II
Characteristics of parallel codes for k threads used for the experiments

The considered algorithms are described in Table II. For each one we provide the size of the input data, and the complexities of sequential part and the parallel part. We can notice that the considered algorithms show a large variety of workload and input data sizes. All algorithms are fully parallel: the sequential work load is always really small compared to the one of the parallel part. These algorithms concern linear

algebra calculations (matrix and polynomial multiplications), cryptographic computation (AES, Number Theoretic Transform (NTT)), image processing (edge detection), compression (LZSS) and sorting (qsort, merge sort).

V. Results

This section describes the execution characteristics of our selected benchmarks on the LBMC platform for differer configurations.

LBMC cores are in-order barrel processors where eac OpenMP software thread is attached to one of a core's har Each hart has one fixed slot on the core pipeline, and eac slot priority set through a round-robin policy. This means tha any hart can at maximum, occupies the pipeline one *Nth* o the time, where *N* is the total number of harts of the core One restriction of LBMC is that it was made specifically fo parallel execution, hence it does not support a configuratio with only one hart. To realize our experiments in sequentia mode, we had to use the two-harts configuration with one ha deactivated.

We provide an evaluation of the performance of the code produced by our toolchain as well as some of its deterministic characteristics. All the measures were collected only after proper program termination.

A. Homogeneous workload

Table III illustrates the performance in ipc (instructions per cycle) of the benchmarks when executed on LBMC configured with one, two or four threads. In these experiments, the parallel workload is homogeneous across all harts, that is, in each benchmark the work is uniformly spread out across all harts. Each hart performs the same computation over a distinct part of the data.

This kind of execution allows us to see how our restricted model of parallelism performs compared to the "classical" model of parallelism. In particular, our aim is to show that the restrictions imposed by our model do not majorly degrade performance. Given Amdahl's law, the maximum speedup in pleasingly parallel problems such as matmul should increase linearly with the number of deployed threads.

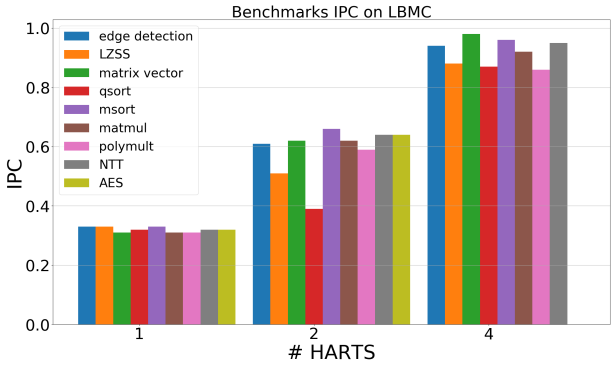

Fig. 3. Homogeneous work IPC

Figure 3 displays the observed performance per benchmark between different numbers of threads. As mentioned before, the maximum ipc of the mono-threaded execution of the benchmarks is only half that of the two-hart platform, so 0.5. As expected, with more threads, the benchmarks are executed faster.

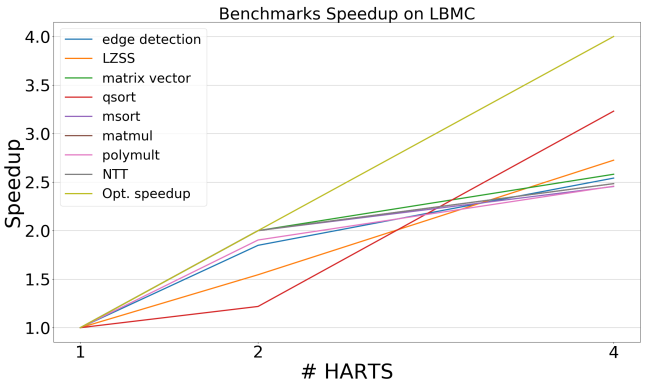

Fig. 4. Homogeneous work speedup

Figure 4 illustrates the ratio in ipc between the different platform configurations, as well as the theoretical optimal speedup for reference. From one to two harts the speedup ranges between times 1.2 to 2.0, and from one to four harts it ranges between times 2.4 to 3.2. The speedup drop-off in ratio from two to four harts is due to the expected diminishing returns coming from "parallelism".

We can observe that, despite the stricter inter-thread communication policies, the ipc still increases with the number of threads.

While these kinds of workload are useful for observing raw performance, they are not the best suited for observing harder parallel computing problems such as inter-threads communications.

a) Inter-thread communication costs: Indeed, the overhead for memory communications inside a whole pool of threads increases linearly with the number of threads. This is an inherent consequence of the way communications are enforced in our model of parallelism. Doubly so as a common pattern of parallel work-sharing requires two-ways communications.

To alleviate this, we tried to remove as many communications as possible, which led to the pre-loading of sequentially processed data before the first parallel region. However, this method is not adapted for programs with intensive re-entrant parallel sections. This is due to the fact that, restoring an identical memory image among all the threads equates having to copy the whole memory of the initial thread to all the other threads.

Table IV illustrates how these memory copies impact the overall performance with the matrix multiplication benchmark. A chain of forks distributes the sub-matrices to each thread. Each fork copies of the matrix from parent to child, until all

benchmark	input size	#inst	#cycles	ipc	#inst	#cycles	ipc	#inst	#cycles	ipc
		1 thread 2 harts			2 threads 2 harts			4 threads 4 harts		
AES	n=128	3194614	10001115	0.32	3195405	5001744	0.64	-	-	-
edge detec.	n=32	200219	612837	0.33	201691	328989	0.61	204391	218277	0.94
LZSS	n=1024	27949	84849	0.33	6563450	12907090	0.51	7143393	8083457	0.88
qsort	n=512	225072	708655	0.32	250879	646116	0.39	8082552	9315412	0.87
mergesort	n=32	17329	52719	0.33	39206	59661	0.66	163184	170257	0.96
matvect	n=32	72525	233745	0.31	74322	119567	0.62	87809	90027	0.98
matmul	n=32	2466365	7986917	0.31	2467389	3994992	0.62	2469437	2686876	0.92
polymult.	n=256	4752372	15307185	0.31	4761579	8126396	0.59	4779993	5576958	0.86
ntt	n=512	189881	592669	0.32	218061	341956	0.64	326733	344092	0.95

TABLE III
HOMOGENEOUS BENCHMARKS

work is assigned. Then, the subsequent chain of joins copies the resulting sub-matrices from the last child back to the first parent.

matrix multiplication 4 threads 4 harts			
with loading in fork	with loading in join	#inst	#cycles
YES	YES	573563	593341
YES	NO	110370	114183
NO	YES	508934	526252
NO	NO	61635	63238

TABLE IV
IMPACT OF PRE/POST LOADING ON MAT. MUL.

We observe that, as described precedently, the costs of inter-thread communications is high. The performance of the same program, between inter-thread data loading and data pre/post-loading is about an order of magnitude. Between fork and join, the data transfers linked to the join operations are slower, this is due to the very nature of our memory model. Indeed, our communications follow a strict parent to child descending hierarchy. As such, any communications initiated by a parent to a descendant child has a clear priority among the whole chain of threads. However, this is not the case for the reverse, when a child needs to communicate with an ancestor parent, its message has to go back up the whole chain of threads with a low priority.

Since our target LBMC platform aims to emulate very small embedded hardware, we were limited in the size of matrices that fit in memory. This exacerbate the apparent cost of thread communications, relative to the amount of arithmetic operations.

Comparison. We did not compare the experimental results of the proposed approach with classical approaches of the literature for deterministic execution of parallel program on bare-metal platform. Indeed, deterministic approaches of the literature require an operating system, but this would require to have a platform equipped with an OS and with characteristics similar to our platform LBMC: we could not find any platform like this. Second, the approaches of the literature for running parallel program on bare-metal platforms are, to the best of our knowledge, non-deterministic and crafted to specific platform.

VI. RELATED WORK

There is an increasing demand for parallel execution on critical embedded devices which require predictable time and functional behavior. Many works of the literature focus on designing/extending parallel programming language to be executed on platform equipped with an operating system.

For example, in [29], [30] the authors propose a C-based parallel programming language called ForeC for synchronous program. The ForeC language provides a semantic for parallel section for data sharing, along with combine function executed at synchronisation.

OpenMP was considered for the development of parallel programs for critical real-time systems. In [25] the authors propose a scheduling algorithm for tied tasks in OpenMP: a tied task is assigned to be executed to a fixed thread/core. In [23] the authors propose to extend OpenMP with new clauses (deadline and event) and they also show that current OpenMP task scheduler can be used to implement real-time task scheduling policy.

Functional determinism of parallel OpenMP program was studied in [3]. The authors provide a deterministic version of OpenMP: the adopted strategy is to run parallel threads on isolated copies of data and merge these copies at synchronisation points. This eliminates unpredictable read/write/conflicting access on shared data inducing non deterministic execution of parallel programs. In [2] the same authors use similar strategy to design an OS enforcing the determinism of parallel programs.

We can also mention several works which consider to ease the porting of parallel OpenMP program from platforms with an operating system to bare-metal. Mariongiu et al. [21] [20] demonstrated that OpenMP can be implemented for an embedded platform with distributed memory.

We can also report two works which provide a runtime library for bare-metal platforms. The first one [24] provides an OpenMP runtime for C66X a multicore platform with RISC and DSP for HPC computation. The runtime in [24] is implemented with the OpenEM API of the platform. The second work [26] provides a runtime library for embedded devices based on standardised API (MRAPI) for communication designed by MultiCore Association (MCA). They adapt the

main constructs of OpenMP to handle the memory model of the MRAPI which includes remote and shared memories.

VII. Conclusion

We considered in this paper the development of parallel application over bare-metal platform. Our goal was to get a simplified thread management and have a determinism in time and in execution. For this reason we used a parallel model which isolates the threads memory: all threads are working only on private data. Thread synchronisations and data sharing are only done only at fork and join between parent and child threads.

We presented a set of tools based on this parallel model, to aid in the execution of parallel OpenMP C code on a bare-metal platform. These tools takes parallel OpenMP C codes, compile them and port them to a prescribed bare-metal platform. We presented experimental results for OpenMP parallel codes compiled with the proposed tools and run on the LBMC [14] platform. The results obtained show that the parallel model preserves the expected speed-up when increasing the number of threads. We also noticed that determinism in time and in execution are also provided by the proposed approach.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS 1967 (Spring)*, page 483–485. Association for Computing Machinery, 1967.

[2] A. Aviram. Efficient System-Enforced Deterministic Parallelism. *Commun. ACM*, 55(5):111–119, 2012.

[3] A. Aviram and B. Ford. Deterministic OpenMP for Race-Free Parallelism. In *3rd USENIX Workshop on Hot Topics in Parallelism*. USENIX Association, 2011.

[4] D. Bailey. *NAS Parallel Benchmarks*, pages 1254–1259. Springer US, 2011.

[5] J. Balart, A. Duran, M. Gon, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: A research compiler for OpenMP. *Proceedings of the European Workshop on OpenMP*, 8, 01 2004.

[6] C. Bienia, S. Kumar, J. Singh, and K.Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of PACT 2008*, pages 72–81, 2008.

[7] O. A. R. Board. *OpenMP Application Programming Interface Specification 5.2*. 2021.

[8] B. M. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing OpenMP on a high performance embedded multicore MPSoC. In *Proceedings of IPDPS 2009*, pages 1–8, 2009.

[9] M. E. Conway. A multiprocessor system design. In *Proceedings of AFIPS 1963 (Fall)*, page 139–146. Association for Computing Machinery, 1963.

[10] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, sep 1965.

[11] L. Eeckhout. Is moore's law slowing down? what's next? *IEEE Micro*, 37(4):4–5, 2017.

[12] S. Eyerman and L. Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Proceedings of ISCA 2010*, page 362–370. Association for Computing Machinery, 2010.

[13] B. Goossens and D. Parello. Limits of Instruction-Level Parallelism Capture. In *Proceedings of ICCS 2013*, volume 18 of *Procedia Computer Science*, pages 1664–1673. Elsevier, 2013.

[14] B. Goossens, D. Parello, and D. Bikov. A Multicore and Multithreaded Microcontroller. In *Proceedings of IC-AIRES 2022*, volume 591 of *LNNS*, pages 22–36. Springer, 2022.

[15] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. *SIGMOD Rec.*, 25(2):173–182, jun 1996.

[16] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of WWC-4*, pages 3–14, 2001.

[17] T. Hanawa, M. Sato, J. Lee, T. Imada, H. Kimura, and T. Boku. Evaluation of Multicore Processors for Embedded Systems by Parallel Benchmark Program Using OpenMP. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 15–27, 2009.

[18] S. Karlsson. An Introduction to Balder — An OpenMP Run-time Library for Clusters of SMPs. In *OpenMP Shared Memory Parallel Programming*, pages 78–91. Springer Berlin Heidelberg, 2008.

[19] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. *Concurrency and Computation: Practice and Experience*, 19:2317–2332, 12 2007.

[20] A. Marongiu and L. Benini. An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs. *IEEE Transactions on Computers*, 61(2):222–236, 2012.

[21] A. Marongiu, P. Burgio, and L. Benini. Supporting OpenMP on a multi-cluster embedded MPSoC. *Microprocessors and Microsystems*, 35(8):668–682, 2011. Design and Verification of Complex Digital Systems.

[22] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

[23] M. A. Serrano, S. Royuela, and E. Quiñones. Towards an OpenMP Specification for Critical Real-Time Systems. In *Proceedings of IWOMP 2018*, volume 11128 of *LNCS*, pages 143–159. Springer, 2018.

[24] E. Stotzer, A. Jayaraj, M. Ali, A. Friedmann, G. Mitra, A. P. Rendell, and I. Lintault. OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip. In *9th International Workshop on OpenMP, IWOMP 2013*, volume 8122 of *LNCS*, pages 114–127. Springer, 2013.

[25] J. Sun, N. Guan, X. Wang, C. Jin, and Y. Chi. Real-Time Scheduling and Analysis of Synchronous OpenMP Task Systems with Tied Tasks. In *Proceedings of DAC 2019*, page 94. ACM, 2019.

[26] C. Wang, S. Chandrasekaran, B. M. Chapman, and J. Holt. libEOMP: a portable OpenMP runtime library based on MCA APIs for embedded systems. In *International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2013*, pages 83–92. ACM, 2013.

[27] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović. The risc-v instruction set manual, volume i: User-level isa, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[28] S. Woo, M. Ohara, E.Torrie, J. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995.

[29] E. Yip, A. Girault, P. Roop, and M. Biglari-Abhari. The ForeC Synchronous Deterministic Parallel Programming Language for Multicores. pages 297–304, Sept. 2016.

[30] E. Yip, A. Girault, P. Roop, and M. Biglari-Abhari. Synchronous Deterministic Parallel Programming for Multi-Cores with ForeC. *ACM Trans. on Programming Languages and Systems*, 45, 2023.

[31] D. Zöbel. The deadlock problem: a classifying bibliography. *SIGOPS Oper. Syst. Rev.*, 17(4):6–15, oct 1983.