Technical Report

Towards Transparent Parallel/Distributed Support for Real-Time Embedded Applications

Ricardo Garibay-Martínez

Luis Lino Ferreira

Cláudio Maia

Luis Miguel Pinho

CISTER-TR-130202

Version:

Date: 2/14/2013

Towards Transparent Parallel/Distributed Support for Real-Time Embedded Applications

Ricardo Garibay-Martínez, Luis Lino Ferreira, Cláudio Maia, Luis Miguel Pinho

CISTER Research Unit

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail: rgmaz@isep.ipp.pt, llf@isep.ipp.pt, clrrm@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

An increasing number of real-time embedded applications present high computation requirements which need to be realized within strict time constraints. Simultaneously, architectures are becoming more and more heterogeneous, programming models are having difficulty in scaling or stepping outside of a particular domain, and programming such solutions requires detailed knowledge of the system and the skills of an experienced programmer. In this context, this paper advocates the transparent integration of a parallel and distributed execution framework, capable of meeting real-time constraints, based on OpenMP, and using MPI as the distribution mechanism. The paper also introduces our modified implementation of GCC compiler, enabled to support such parallel and distributed computations, which is evaluated through a real implementation. This evaluation gives important hints, towards the development of the parallel/distributed fork-join model for real-time embedded applications.

Towards Transparent Parallel/Distributed Support for Real-Time Embedded Applications

Ricardo Garibay-Martínez, Luis Lino Ferreira, Cláudio Maia and Luís Miguel Pinho
CISTER/INESC-TEC, ISEP
Polytechnic Institute of Porto, Portugal
{rgmz, llf, crrm, lmp}@isep.ipp.pt

Abstract—**An increasing number of real-time embedded applications present high computation requirements which need to be realized within strict time constraints. Simultaneously, architectures are becoming more and more heterogeneous, programming models are having difficulty in scaling or stepping outside of a particular domain, and programming such solutions requires detailed knowledge of the system and the skills of an experienced programmer. In this context, this paper advocates the transparent integration of a parallel and distributed execution framework, capable of meeting real-time constraints, based on OpenMP programming model, and using MPI as the distribution mechanism. The paper also introduces our modified implementation of GCC compiler, enabled to support such parallel and distributed computations, which is evaluated through a real implementation. This evaluation gives important hints, towards the development of the parallel/distributed fork-join framework for supporting real-time embedded applications.**

Keywords—parallel execution, real-time, distributed embedded systems, compiler support, GCC, OpenMP, MPI.

I. INTRODUCTION

Real-time embedded systems are present in our everyday life. These systems range from safety critical ones to entertainment and domestic applications, presenting very diverse set of requirements. Although diverse, in all these areas, modern real-time applications are becoming larger and more complex, thus demanding more and more computing resources.

By using parallel computation models, the time required for processing computational intensive applications can be reduced, therefore, gaining flexibility. This is a known solution in areas that require high performance computing power, and real-time systems are not the exception. Therefore, the real-time community has been making a large effort to extend real-time tools and methods to multi-cores [1], and lately to further extend them considering the use of parallel models [2].

Nevertheless, these parallel models do not take into consideration heterogeneous architectures, which mix both data-sharing and message passing models. In [3], we introduced a solution for parallelising and distributing workloads between neighbouring nodes based on a hybrid approach of OpenMP [4] and Message Passing Interface (MPI) programs [5], which can be used in this context. Furthermore, we presented a timing model, which enables the structured reasoning on the timing behaviour of such hybrid parallel/distributed programs.

However, one main disadvantage of the use of such approaches comes from the programmer's point of view. Coding parallel programs is not a straight-forward task, even more, when programming for distributed memory; including real-time constraints adds even more programming complexity. This complexity usually requires detailed knowledge of the system and the skills of an experienced programmer; constraints that may not always be affordable (in cost or time).

In this context, this paper presents a transparent parallel distributed fork-join execution model intended to support real-time constraints. Furthermore, we introduce our modified implementation of GNU Compiler Collection (GCC) compiler [6]; enabled to support parallel and distributed computations in a transparent manner. We also show through a real implementation, how the execution times of parallel/distributed applications can be reduced by following our execution model. We also derive some conclusions on the importance of considering the transmission time (e.g. implicit transmission delay) when developing distributed applications.

II. PARALLEL/DISTRIBUTED REAL-TIME EXECUTION MODEL

The parallel/distributed fork-join model for distributed real-time systems has been introduced in [3]. One of our main goals is to be able to model the parallelisation and distribution of computations of real-time tasks. Thus, the generic operation of the local process/thread which performs a parallel/distributed fork-join execution is as follows: i) The local process/thread initialize the distributed environment (e.g. by calling `MPI_Init()`); ii) it determines the data to be sent and sends it using a communication mechanism (e.g. by using `MPI_Send()`); iii) the data gets transmitted through the network with a certain implicit delay; iv) the data is received on the remote neighbour node (e.g. by using `MPI_Recv()`) and processed (in parallel if more than one core is available in the remote node); v) when the execution in the remote node is finished, the results are sent back through the network to the local process/thread; vi) finally, the results are gathered and the final result produced.

To implement this model we propose the introduction of a new `#pragma omp` **`parallel distributed`** `for` in OpenMP, which removes the burden of coding the parallel/distributed application from the programmer. Also, a **`deadline()`** clause can be associated to this pragma, passing as parameter the number of milliseconds on which the application is expected to finish its parallel/distributed part of code. Figure 1 depicts a fragment of code using this pragma, indicating that the code embraced within the *for* loop can be executed on distributed nodes within no more than 200 milliseconds.

```
#pragma omp parallel distributed for
 deadline (200) num_threads(3){
    for (i = 0; i < 4; i++)
        loopCode();
}
```

Figure 1. #pragma omp parallel distributed *for* deadline pragma example.

Figure 2 is a timeline representation of the execution of parallel/distributed fork-join, where the horizontal lines represent threads/processes and the vertical lines represent forks and joins. In this case, the main thread splits in three threads (creates a team of OpenMP threads), two are executed in parallel on the local node and another is, mainly, executed on a remote node, hereafter we call such kind of execution as *remote execution*. Furthermore, we also assume that it is possible to split the *remote execution* in two threads, one for each core on the remote node. This is done automatically by the framework. Thus, taking as example the code in Figure 1, the timeline in Figure 2 assumes that two threads $\sigma_{1,1,1}$ and $\sigma_{1,1,2}$ execute locally one *for* loop iteration each, the distributable thread $\sigma_{1,1,3}$ executes the remaining two iterations. Then, thread $\sigma_{1,1,3}$ is hosted in a remote node and further split into two threads, by adding thread $\sigma_{1,1,4}$, each one of these threads is executing one iteration of the *for* loop. By looking at Figure 2, it is also considered the inherent transmission delay of transmitting and receiving code and data, this is the transmission delay $\mathcal{TD}_{1,3}$ and $\mathcal{TD}_{3,1}$, respectively. In this case note that the MPI framework places the code on distributed nodes and starts the remote programs.

Also, it is important to notice that the fork-join execution is divided in a sequence of serial and parallel segments (e.g. parallel regions in OpenMP), which implies precedence constraints between the segments of a fork-join task. Furthermore, the execution of a fork-join real-time task has associated Worst-Case Execution Time C_i, a task period T_i which indicates the minimum interval time in which a task is periodically invoked, and a task deadline D_i which defines the maximum possible time a task can take to be completed.

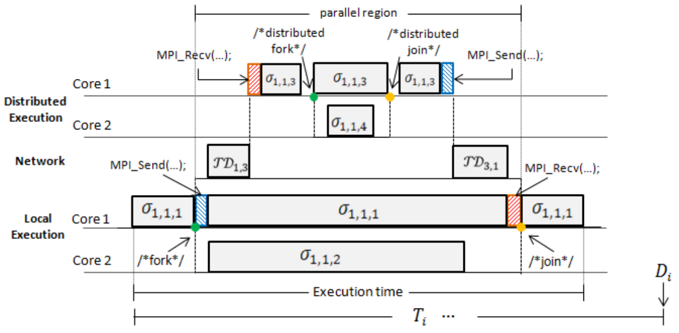

Figure 2. Timeline of a real-time task using the parallel/distributed fork-join model.

III. Towards Parallel/Distributed Real-Time Compiler Support

A. *GCC Compiler and OpenMP*

GCC is structured in three modules, the *front-end* (also known as parser) which is responsible for identifying and validating the source code (e.g. lexical analyses), the *middle-end* which has the main objective of simplifying and optimizing the code, and the *back-end* which is in charge of transforming the final optimised code to assembly code, by taking into consideration the destination platform. The modifications required for implementing our parallel/distributed model only require changes on the GCC front-end and middle-end.

The front-end parses the code in a recursive manner and performs sanity checks of the code. The main objective is to identify the language *keywords* that affect the execution of the program (including the OpenMP keywords). It is in this stage, where the new **`distributed`** clause is added to the existing set of OpenMP clauses. The result of applying the parsing process is an intermediate code called GENERIC, which is later propagated to the middle-end for further processing.

The middle-end transforms the GENERIC code into the intermediate representation called GIMPLE, this process is usually refereed as the *"gimplification"* step. During this step all implicit data sharing clauses are made explicit and atomic directives are transformed into the corresponding atomic update functions. Similarly to the front-end, the middle-end starts a process of simplification and optimizations of the code. Each optimization or simplification is defined as a *pass*.

The *pass manager* is in charge of calling the set of passes which are included in the *passes.c* source file. One of those passes is the OpenMP lowering pass. The objective of the OpenMP lowering pass is to identify the OpenMP clauses, transform them in equivalent simplified code (GIMPLE code), and initialize the auxiliary variables of the expansion phase [7]. After the lowering pass, the pass manager invokes the OpenMP *expansion pass*. The expansion pass is in charge of outlining the OpenMP parallel regions and introducing the built-in function to be called by the *libgomp* library. *libgomp* is the GNU implementation of the OpenMP API for shared memory platforms.

In our implementation, a new function has been created in *libgomp*. The `GOMP_Distributed_Parallel_start()` function initialises the MPI environment; from that point, the code is simultaneously executed by all MPI processes. The number of created MPI processes is equal to the number of hosts in mpd.hosts configuration file of MPI, which indicates the number of nodes in the distributed system (e.g. a cluster of embedded computing devices). All MPI processes receive the data to execute using MPI broadcast primitives. In our current implementation the workload is evenly divided among nodes in the system.

After all processes have received their corresponding data, each process creates a team of OpenMP threads. By default a team of OpenMP threads is then created, having the same number of threads as the number of processors/cores in the node they are going to be executed.

Once the team of threads has been created, the workload can be re-assigned by using the standard sharing mechanisms implemented by *libgomp*. Then, whenever the execution of a parallel region is finalised, the execution of the OpenMP team of threads and MPI processes is automatically terminated by calling the proper termination routine.

B. *Towards Real-time Libgomp Implementation*

Currently, we have an implementation, which is able to parallelise and distribute computing workloads transparently. However, our final objective is to be able to support real-time processes/threads based on the OpenMP programming model.

In order to support real-time processes/threads, an underlying real-time operating system must be used. We are mainly interested on the use of two available real-time kernels for Linux: the SCHED_DEADLINE [8] and the SCHED_RTWS [9]. The first of them, implements partitioned, global and clustered scheduling algorithms that allows the cohabitence of hard and soft real time tasks. The second one, implements a combination of the Global EDF and a priority-based work-stealing algorithm, which allows executing parallel real-time tasks in more than one core at the same time instant whenever an idle core is available.

libgomp implements threads by using the POSIX threads library [10] (also known as *pthreads*). This is done by calling `gomp_team_start()` function. This function initializes the data structures and passes the required parameters to the `phtread_create()` function. Is in this point where we can use the `phtread_ create()` function, to create the new threads according to the received real-time parameters extracted from the `deadline()` clause. The implementation of the `deadline()` clause can be done by repeating a similar procedure as when implementing the `distributed` clause.

In the following section, we present the performance evaluation of our `distributed` clause implementation, of the non-real-time version of the software. Although, the real-time support is being currently implemented, the results help us to confirm that we are following the correct path for supporting real-time processes/threads based on the OpenMP programming model.

IV. EXPERIMENTAL EVALUATION

The experiments reported in this paper were conducted in a cluster with 5 identical machines. Each machine is equipped with a dual-core Intel(r) Celeron(r) CPU, with a clock rate of 1.9 GHz and 2 GB memory. Communications were supported by 8-port 100 Mbps n-way fast Ethernet switch.

The experiments are based on the execution of the `#pragma omp` **`parallel distributed`** `for` pragma, and having a variable number of iterations. This number of iterations represents different sizes of a synthetic workload that need to be processed within some time constraints. For example, in Figure 3, iterations contained in the variable N_ITER are to be divided among the nodes/cores in the system. Each iteration takes in average, approximately 0.016 ms to be executed. The iterations inside the *for* loop are considered to be independent between them, and therefore, each iteration only produces changes on an independent part of the data.

The work-sharing mechanism used for these experiments, divides the iterations contained inside the *for* loop between the MPI processes (one process per node) and they further split on the number of OpenMP threads (one thread per core). We had chosen this approach since is one of the most general work-sharing mechanisms used for the Single Program Multiple Data (SIMD) paradigm. On the other hand, it has the disadvantage of not being suitable for handling more dynamic patterns of parallelism (e.g. variable real-time patterns).

```
1.    #pragma omp parallel distributed for
2.       for (i = 0; i < N_ITER; i++)
3.          loopCode();
4.    }
```

Figure 3. Distributed *for* clause example.

The data collected is related to experiments with a different number of iterations (100, 1000 and 10000) – variable N_ITER in Figure 3. Each experiment consists on averaging the execution time of the distributed parallel loop over one thousand measurements. I.e. measuring the time from before the execution of line 1 to after line 4. When conducting the experiments, we were interested in estimating the reduction of the execution time as a function of the number of utilised nodes, and estimating the *maximum measured execution time*.

Figure 4, depicts the *average execution time* and the maximum measured execution time of three experiments, having 100, 1000 and 10000 iterations, respectively. It is important to note that the vertical axis in Figure 4 has a logarithmic scale. It is also possible to observe that the execution time (numerically presented above the bars) and the maximum measured execution time (represented as error bars) are reduced almost in a linear way (considering the logarithmic scale) for the cases of 10000 and 1000 iterations. This is an expected speedup for parallel programs when computations realised in different nodes are independent. However, this is not the case for the execution with 100 iterations; where the execution times are kept almost the same, regardless the number of nodes utilised during execution. The reason for this behaviour is that the time saved on the parallel task execution, is consumed in transmission time.

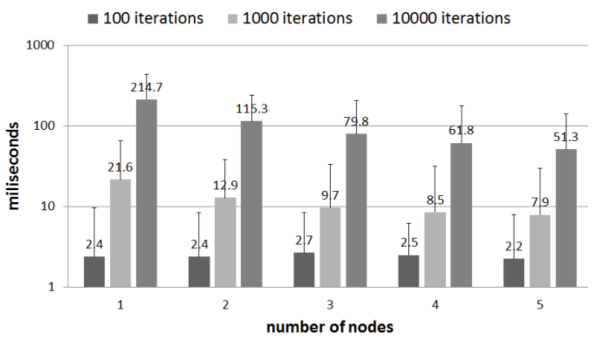

Figure 4. Execution times with 100, 1000 and 10000 iterations.

Also, the *average transmission time* and the *maximum mesuared transmission time* are important parameters to take into account when scheduling parallel tasks in a distributed environment, because they add a considerable extra time (delay), to the total execution, which is usually not considered when scheduling tasks in shared-memory platforms. For example, the size of the data to be transmitted is proportional to the number of iterations contained in the N_ITER variable. During the initial fork operation the data size to be transmitted is approximately 6.25 Kb, 62.5 Kb and 625 Kb, when N_ITER variable is equal to 100, 1000 and 10000 iterations, respectively. The same amount of data is later transmitted to the main node during the join operation. Figure 5 shows the total transmission time (e.g. the sum of $\mathcal{TD}_{1,3}$ and $\mathcal{TD}_{3,1}$ in Figure 2) for the different values of N_ITER mentioned above.

The results in Figure 5, show that the number of nodes in the cluster does not affect the transmission times. The reason is that the size of the data to be transmitted is constant and transmitted using MPI broadcast. The last, holds because the workload to be transmitted is considerably small in comparison with the total capacity of the network. It is also important to note that these measurements had been done in a closed network where the only traffic is related to our experiments.

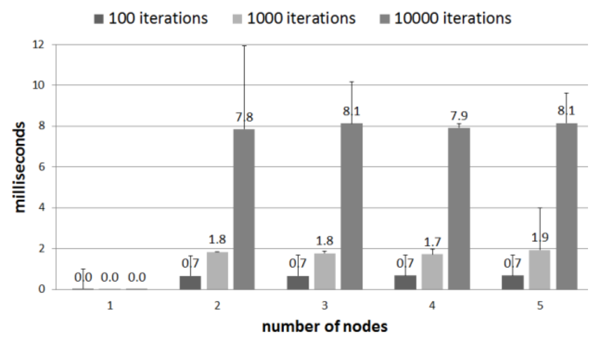

Figure 5. Transmission times with 100, 1000 and 10000 iterations.

Therefore, we can confirm that when the execution time increases, the transmission time becomes more negligible. Thus, in order to correctly address parallel/distributed applications, a trade-off between local/distributed computations (execution time) and the cost of transmitting computations (transmission time) over networked devices, needs to be carefully considered. This is one of the main differences of our model, when compared to pure shared-memory approaches. By observing the preliminary results in Figures 4 and 5, we can derive two remarks from these plots.

The first one is that total execution time needs to be pondered, in order to decide if it is worth to apply a workload distribution, or not. For example, when running applications with execution times of a few milliseconds, the total execution time becomes more susceptible to delays (e.g. the implicit transmission delay). Therefore, it may be not be worthwhile to realize workload distribution. It is possible to observe this effect in Figure 4 for the case of 100 iterations. The second remark is that the existing parallel/distributed algorithms implemented by OpenMP and MPI are not suitable for handling real-time workloads since no resources are reserved in the nodes neither in the network. Therefore, this opens opportunities to explore different and more complex work-sharing mechanisms and scheduling algorithms.

V. Conclusions and Future Work

This paper presented the parallel/distributed fork-join real-time execution model. We proposed an automatic code generation by modifying the GCC compiler and enabling it to transparently support MPI messages based on OpenMP programming model. To do so, we proposed an extension to the OpenMP specification by adding the `#pragma omp` **`parallel distributed`** `for`. Also, we describe the modification done on the GCC compiler to support transparent parallel/distributed executions.

We are currently working on the implementation and integration of the **`deadline`**() clause into *libgomp*. This will allow *libgomp* to be able to generate real-time threads. We also plan to join *libgomp* with the new real-time SCHED_RTWS scheduler for Linux. Then, it will be possible to execute parallel/distributed threads on a real-time kernel and therefore, having real-time guarantees on the execution of parallel threads.

Acknowledgments

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology), by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within VipCore project, ref. FCOMP-01-0124-FEDER-015006, and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/71562/2010.

References

[1] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” *ACM Comp. Surv.,* vol. 43, no. 35, p. 1–44, 2011.

[2] A. Saifullah, K. Agrawal, C. Lu and C. Gill, “Multi-core Real-Time Scheduling for Generalized Parallel Task Models,” in *Proc. of the IEEE 32st Real-Time Systems Symposium (RTSS 2011)*, 2011.

[3] R. Garibay-Martinez, L. L. Ferreira and L. M. Pinho, “A framework for the development of parallel and distributed real-time embedded systems,” in *Proc. of 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2012)*, 2012.

[4] OpenMP Architecture Review Board, "OpenMP application program interface V3.1 July 2011," www.openmp.org/wp/openmp-specifications/, online: last accessed April 2012.

[5] Message Passing Interface Forum, “MPI: A Message-Passing Interface standard version 2.2,” online: http://www.mpi-forum.org/docs/docs.html, online: last accessed April 2012.

[6] GCC Internals, http://gcc.gnu.org/onlinedocs/gccint/, online: last accessed September 2012.

[7] D. Novillo, "Openmp and automatic parallelization in gcc," in *In the Proceedings of the GCC Developers’ Summit*, June 2006.

[8] N. Manica, L. Abeni, L. Palopoli, D. Faggioli and C. Scordino, “Schedulable device drivers: Implementation and experimental results,” in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, 2010.

[9] L. Nogueira, J. C. Fonseca, C. Maia and L. M. Pinho, “Dynamic Global Scheduling of Parallel Real-Time Tasks,” in *Proc.10th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC’12)*, 2012.

[10] POSIX Threads Programming, https://computing.llnl.gov/tutorials/pthreads/, online: last accessed, September 2012.