Technical Report

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

Ricardo Garibay-Martínez

Luis Lino Ferreira

Luis Miguel Pinho

HURRAY-TR-120403

Version:

Date: 3/16/2012

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

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

IPP-HURRAY!

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: rgmz@isep.ipp.pt, llf@isep.ipp.pt, lmp@isep.ipp.pt

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

Abstract

Embedded real-time applications increasingly present high computation requirements, which need to be completed within specific deadlines, but that present highly variable patterns, depending on the set of data available in a determined instant. The current trend to provide parallel processing in the embedded domain allows providing higher processing power; however, it does not address the variability in the processing pattern. Dimensioning each device for its worst-case scenario implies lower average utilization, and increased available, but unusable, processing in the overall system. A solution for this problem is to extend the parallel execution of the applications, allowing networked nodes to distribute the workload, on peak situations, to neighbour nodes. In this context, this report proposes a framework to develop parallel and distributed real-time embedded applications, transparently using OpenMP and Message Passing Interface (MPI), within a programming model based on OpenMP. The technical report also devises an integrated timing model, which enables the structured reasoning on the timing behaviour of these hybrid architectures.

© IPP Hurray! Research Group

A Framework for the Development of Parallel and Distributed Real-Time Embedded Systems

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

Abstract—Embedded real-time applications increasingly present high computation requirements, which need to be completed within specific deadlines. But, those applications present highly variable patterns, depending on the data set in a determined instant. The current trend to provide parallel processing in the embedded domain allows providing higher processing power; however, it does not address the variability in the processing pattern. Dimensioning each device for its worst-case scenario implies lower average utilization, and increased available, but unusable, processing in the overall system. A solution for this problem is to extend the parallel execution of the applications, allowing networked nodes to distribute the workload, on peak situations, to neighbour nodes. In this context, this paper proposes a framework to develop parallel and distributed real-time embedded applications, transparently using OpenMP and Message Passing Interface (MPI), within a programming model based on OpenMP. The paper also devises an integrated timing model, which enables the structured reasoning on the timing behaviour of these hybrid architectures.

Keywords-real-time; parallel execution; distributed embedded systems; hybrid programming model; OpenMP; MPI.

I. INTRODUCTION

Nowadays, real-time embedded systems are part of our everyday life. These systems range from the traditional areas of military and mission critical to, domestic and entertainment applications. One of the main characteristics of real-time applications is the need to perform computations within deadlines, for which an extensive bulk of work has been provided for several decades. However, more and more, real-time applications are evolving to become larger and more complex, generating larger and more dynamic workloads which are not easy (or efficiently) dealt with traditional approaches.

The use of parallel models can reduce the time required for processing computational intensive applications, and it is currently the general trend to increase processing in all areas, and real-time systems are not an exception. Therefore, the real-time community has been making a large effort to extend real-time tools and methods to multi-cores, lately considering the use of parallel models at the application level [1-3].

Despite the fact that parallel processors can offer increased processing capacity, dimensioning each computing system for the maximum local worst-case scenario (the traditional real-time systems model) does not deal efficiently with the high variability of application loads, particularly in more dynamic scenarios. Furthermore, in some embedded applications, the use of powerful enough multi-core processors, is prohibited due to energy, space or cost constraints. Example of such type of applications are, for instance, image processing for obstacle detection in cooperating robots, where the computation requirements of the detection algorithms are highly dependent on the robot's current velocity, surrounding scenario and obstacles [4-6].

Consequently, whenever it is possible to connect an embedded system through a local network, it might be possible to perform part of those computations, on neighbouring nodes with available processing capabilities. Frameworks are thus required which allow to dynamically manage globally the resources of the system, allowing peak situations to be distributed cooperatively by the nodes [7].

For this purpose, this paper proposes a framework, based on the OpenMP programming model [8], transparently integrating an underlying distribution framework using Message Passing Interface (MPI) [9]. This hybrid computation model allows program blocks to be transparently distributed, to be executed in neighbour nodes. This transformation must nevertheless be able to provide real-time behaviour. Examples of such efforts are described in [4, 10].

This model considers programs written using the fork/join programming model, with extensions to support distribution and real-time requirements. As with OpenMP, the compiler can then generate the distributed code, to be executed in neighbouring nodes, using the MPI library. If the neighbour nodes support parallel computations, the offloaded component might again be parallelized, using OpenMP.

The paper then puts forward a timing and execution model, which is able to map the code structure of the applications into the underlying parallel and distributed behaviour, which is amenable to timing analysis.

The remainder of the paper is structured as follows. Section II presents the motivation and related work, whilst Section III overviews the OpenMP and MPI programming models. Section IV then presents the general model for supporting parallel/distributed execution using the hybrid OpenMP/MPI model. The timing model for OpenMP/MPI

execution is then put forward in section V. Finally, in Section VI we draw some conclusions and propose future work.

II. Motivation and Related Work

This work targets systems which include embedded computing platforms (e.g. smartphones, small and medium robots, home and factory automation devices, etc.), which are network connected to cooperating neighbour nodes. These platforms can include single or multi-core processors, ranging from resource-scarce to more powerful nodes.

In particular, we have been considering collaborative robotics applications, where dynamic workloads can be parallelized and distributed over the different robots. For instance, the authors of [5] propose an algorithm which can be used for the real-time 3D map generation for robotic applications, by applying parallel techniques to traditional algorithms for solving the Simultaneous Localization and Map Building (SLAM) problem.

Object recognition and tracking is also a computational intensive problem, which cannot be solved on resource scarce platforms in real-time. In [4] the authors propose the use of code offloading techniques which also take into account the tasks' deadlines. Also, [10] presented an adaptive offloading approach for soft real-time systems, which is capable of monitoring the execution time of a specific task, and offload it to other nodes when it predicts that the local node will not be capable of guaranteeing the task's deadline in the future.

Another problem that requires real-time computations in robotics and autonomous vehicles is the self-location problem for large environments; to which the authors of [6] propose a distributed and parallel processing technique, based in the Monte Carlo Technique location algorithm which is implemented in a combination of OpenMP and MPI.

The use of OpenMP and MPI has been considered also in other application domains, particularly in the area of high performance computing, for example [11]. These works show the advantage of the combinations in relation to pure message passing solutions, nevertheless they are only focused on performance and not in the real-time behaviour.

Simultaneously, the real-time community has been analyzing the timing behaviour of parallel and distributed applications. In particular, in the last few years the real-time schedulability theory has been extensively extended to consider multi-core platforms [12]. But, only recently more attention has been given to the case of multithreading parallel task models. In [1], the authors introduced the parallel synchronous task model, where every task in the system is a sequence of serial and parallel segments. Nevertheless, such a model is highly restrictive, and for overcoming these limitations the authors of [2] have introduced its generalization. Both works [1, 2] are based on a method of imposing artificial deadlines to segments belonging to a real-time task to derive new deadlines for the task's segments.

The work of [2] was then extended in [3] by modelling a real-time task as Directed Acyclic Graph (DAG), where vertices represent threads and edges represent possible dependencies between threads.

The integration of distributed computations within real-time applications also requires that the schedulability analysis is extended to the network and the overall distributed application. Network schedulability has been much harder to achieve and depends heavily on the underlying technology. Examples are [13], which provides the foundations of soft real-time scheduling for IEEE 802.11 wireless networks, and [14], which addresses the same problem, but for Ethernet switches. Integrating both network and CPU scheduling can lead to even more complex analysis (e.g. [15]), although the current evolution of multi-core technologies to include on-chip networks will eventually make indistinct the parallel and distributed analysis.

Our work builds upon the base concepts of [2, 3] but it differentiates in: i) we propose a framework and characterization of hybrid OpenMP/MPI programs, considering the possibility of distributed computations and; ii) we focus in the model that is able to map the behaviour of OpenMP constructs (code blocks) and MPI constructs (send/receive operations) into threads, enabling the timing analysability of such kind of programs, instead of focusing on the analysis itself.

III. OpenMP and MPI Programming Models

A. OpenMP Programming Model

The OpenMP API has been developed to provide portability and a user-friendly environment for programming shared memory multiprocessor machines [7]. The great success of OpenMP as a programming model relies on the simplicity of generating parallel programs. Furthermore, it is very efficient for creating incremental parallelism from existing sequential code.

OpenMP programs follow the fork-join paradigm. The structure of a parallel OpenMP program contains: i) a sequential part (e.g., some C/C++ variable initializations); ii) some parallel constructs (e.g. parallel sections), that are inserted in the code for realizing a parallel execution (fork) and finally; iii) the set of generated threads are reduced (e.g. the reduction clause) to generate the final result (join).

In this paper, we only analyse the most relevant OpenMP constructs, which are sufficient for parallelizing the majority of applications. Even more, one of the contributions in this paper is to derive a generic real-time model for OpenMP programs.

In OpenMP, the *parallel construct* defines a segment of the code to be executed in parallel; this segment is known as *parallel region*. The construct is defined by the `#pragma omp parallel` directive (Figure 1). Whenever a thread

encounters a *parallel construct*, this thread becomes the *master thread* and it creates a team of threads, which will run the code inside that code segment.

```
1. #pragma omp parallel sections num_threads(2){
2.    #pragma omp section{
3.       #pragma omp single
4.          /*variable initialization*/;
5.    }
6.    #pragma omp section{
7.       function_1();
8.    }
9. }
```

Figure 1. Parallel sections and single construct example.

At the end of a *parallel region* there is an implicit synchronization mechanism called a *barrier* (line 9, Figure 1). There is a possibility of using a `nowait` clause which inhibits the *barrier* and allows continuing the execution, but it is not considered in this paper. Each thread executes its associated code and waits at the end *barrier*, when all threads terminate only the *master thread* continues.

In order to provide the desired functionality to *parallel constructs*, OpenMP provides a set of *work-sharing constructs*. These constructs are in charge of distributing computation among threads in OpenMP programs.

These *work-sharing constructs* can be complemented and/or modified with a set of clauses to control the parallel execution. Of particular importance in the context of this work is the `numthreads(n)` clause, which is used to specify the number of thread in which to split a *parallel region*. The most relevant *work-sharing* constructs follow.

The amount of parallelism achieved by a *sections construct* is a function of the number of threads and number of individual parallel section clauses associated to it. Each section is defined by the `#pragma omp section` directive.

The *single construct* (`#pragma omp single`) specifies that the associated structured code block is executed by only one of the threads in the team (not necessarily the *master thread*). The other threads in the team, wait at an implicit *barrier* at the end of the *single construct* (line 5, Figure 1). In a similar way the `#pragma omp master` directive guarantees that only the *master thread* will execute a specific code block. Figure 1 depicts a fragment of code related to the use of `parallel section` and `single` constructs.

The *loop construct* is signalled to the compiler through the `#pragma omp for` directive and is used to divide the `for` cycle iteration through several threads. This directive has different behaviours depending on the scheduler type selected, which is determined by internal OpenMP variables or by the `schedule(…)` clause.

This directive might also be associated with a `reduction` clause. Figure 2, depicts a fragment of code exemplifying the `parallel for` construct in OpenMP. In this example the for loop iterations are divided by three threads, each one executing two iterations. OpenMP also provides functionality for nested parallelism. Whenever a thread encounters another parallel construct, it creates a new team of threads and it becomes the new master thread of that team. This allows exploiting extra parallelism in OpenMP programs.

```
1.   #pragma omp parallel for
       num_threads(3) reduction(+:sum){
2.         for (i = 0; i < 6; i++)
3.            loopCode();
4.      }
5.   }
```

Figure 2. Parallel for directive example.

B. MPI Programming model

The MPI specification has become a de facto standard for developing parallel distributed programs using the message passing paradigm [8], which, among other, can implement the fork-join parallel programming model. Common fork-join programs implemented in MPI have i) a serial execution code segment (e.g., variables declaration and initialization); ii) the MPI environment initialization (e.g., a call to `MPI_Init(…)`); iii) an explicit work splitting algorithm, for the distribution of the workload among the processing elements (the splitting algorithm is implemented by the programmer); iv) transfer of data, using message passing calls (e.g., calls to `MPI_Send(…)`, `MPI_Recv(…)`); v) an execution of the computations; vi) a reduce of the partial results from remote nodes to obtain the final one (e.g., a call to `MPI_Reduce(…)`); and vii) the finalization of the execution (e.g., a call to `MPI_Finalize(…)`). A code fragment example is presented in Figure 3.

A normal MPI program starts with a call to `MPI_Init(…)` routine to initialize the MPI environment (line 3). This creates a *communicator*, which groups a set of MPI processes in a local or in different nodes. All MPI messages must specify a *communicator* for interchange of messages between the processes belonging to the same *communicator*. `MPI_Comm_rank(…)`, returns the “rank” (the ID) of a process within the associated *communicator*.

The real potential of MPI programs is realized by the *communication routines*. These communication routines are the ones that realize the real distribution of workload to processes for realizing the parallel computations. MPI communications can be Point-to-Point or Collective. The most used MPI communication functions are `MPI_Send(…)` and `MPI_Recv(…)`, where the first is a non-blocking call and the second blocks until a message is received.

Also, MPI implements reduce operations, in an analogous way to OpenMP through the `MPI_Reduce(…)` function.

One important difference between OpenMP and MPI is that the last leaves all the burden of parallel coding on the programmers hands, while the first supports increasing

parallelism at the cost allowing less flexibility. This is also the reason why we choose a hybrid approach where the programmers' front-end is OpenMP .

```
1.    /* Variables declaration*/
2.    MPI_Init(…);
3.    MPI_Comm_size(…);
4.    MPI_Comm_rank(…);
5.    if (rank == 0) {
6.       MPI_Send(…);
7.       MPI_Recv(…);
8.    }
9.    if (rank !=0){
10.       MPI_Recv(…);
11.      /*Execution*/
12.      MPI_Send(…);
13.      MPI_Finalize();
14.   }
```

Figure 3. MPI two-sided send/receive example.

IV. SUPPORTING PARALLEL/DISTRIBUTED EXECUTION WITH OPENMP/MPI

Based on the OpenMP/MPI constructs we can now propose a model for supporting parallel/distributed execution.

To reduce the complexity of writing parallel distributed programs, in our framework a programmer only writes code in OpenMP API with minor changes to the OpenMP specification. These include the extension of existing OpenMP constructs for enabling them to support workload distribution (supported by MPI). Therefore, the MPI code is not seen by the programmer (MPI code is implicitly called by the OpenMP library), and the programmer only needs to specify which OpenMP code blocks can be distributed using the `#pragma omp distributedParallel` pragma, specifying its deadline to execute that parallel code block. This is illustrated in Figure 4, where we specify that the `for` loop can be distributed among 3 threads, and the computation must be completed within a deadline of 200 milliseconds.

In this case, the `distributedParallel` directive, signals the compiler to enable the parallelization of some iterations of the parallel for loop on distributed nodes. How the decision of which chunks to distribute is taken, is out of the scope of this paper, but we are currently working on such algorithm for *work-sharing* distribution. It is also the responsibility of the compiler to generate code that can be dynamically or statically parallelized on the destination node, again using OpenMP.

```
1. #pragma omp distributedParallel for
      deadline (200) num_threads(3){
2.          for (i = 0; i < 4; i++)
3.             loopCode();
4.       }
5.    }
```

Figure 4. Distributed parallel for example.

By dynamic we mean that our run-time decides the number of threads to split the computation on the neighbour node(s), according to the availability of resources. By static we mean that it is the programmer who specifies the splitting procedure. Consequently, for software written based on this model, it presents an execution timeline similar to the one in Figure 5, which is related to the code in Figure 4.

The horizontal lines are the threads and the vertical lines represent forks and joins. In this case the `parallel for` clause splits in three threads, two are executed on the local node and another is, mainly, executed on a cooperative 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.

Observing the timeline in Figure 5, we assume 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. Thread $\sigma_{1,1,3}$ is hosted in a neighbour 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. Also, in Figure 5 it is considered the inherent delay of transmitting and receiving code or data, $\mathcal{TD}_{1,3}$ and $\mathcal{TD}_{3,1}$, respectively. We are also assuming that it is the responsibility of the master thread to marshal and send the data required for *remote execution*, and for receiving data and unmarshalling it.

A full notation is introduced in Section V.

The generic operation of the local thread which controls the *remote execution* (the master thread) is as follows: i) The local thread must issue a `MPI_Init()` to initialize the MPI environment; ii) it determines the data to be sent and sends it using `MPI_Send()`; iii) the data gets transmitted through the network with a certain delay; iv) the data is received on the other node and executed as an OpenMP program; v) when the execution is finished, the results are sent back to the local thread which should already be waiting for the results, by calling `MPI_Recv()`.

We are also assuming that the neighbour node already has the code to be executed. Therefore, we do not need to take into account the costs of transmitting and installing the code. Such operation can be executed during the system setup phase.

In order to combine the functionalities of OpenMP and MPI in a single program, both APIs need to reach certain commitments to guarantee for the correct execution of hybrid programs.

On the OpenMP side, it requires to guarantee, that if a single thread is blocked by an operating systems call, all the other threads can still be runnable. This is already supported by the most recent OpenMP implementations [7].On the MPI side, from the release of MPI-2 standard [8], the concept of *level of thread support* has been defined. There are four levels of support: 1) `MPI_THREAD_SINGLE`, only one thread exists in the application, 2) `MPI_THREAD_FUNNELED`, multiple threads can exist but only the *master* thread can make MPI calls, 3) `MPI_THREAD_SERIALIZED`, multiple threads can exists, each thread can make MPI calls as long as there is no other

thread realizing a call, and 4) `MPI_THREAD_MULTIPLE`, multiple threads can exist and they can make MPI calls at any time.

Using `MPI_THREAD_SINGLE`, would make it impossible to split into several threads on the neighbor nodes, any one of the other options allows the execution of such programs according to what has been described. Also, MPI has certain limitations for guaranteeing real-time communications. But for overcoming such limitations the authors of [16] introduced an extension for MPI Real-Time (MPI/RT), the MPI/RT standard is based on channel reservation and fault tolerant mechanisms to guarantee time properties.

This section provided an overview of the envisaged programming and execution model, in the Section V we expand and formalize the execution model.

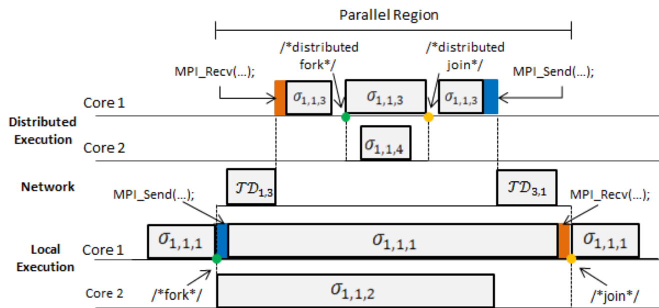

Figure 5. Timeline execution of tasks on a hybrid OpenMP/MPI program.

V. TIMING MODEL FOR HYBRID OPENMP/MPI PROGRAMS

In this section we propose a timing model for hybrid OpenMP/MPI programs. Some previous works have studied a generalization of the fork-join model for real-time systems in shared memory platforms [1-3]. But our work differentiates from others in the sense that we present a characterization of the timing execution for hybrid OpenMP/MPI programs. Furthermore, we propose a model that is able to map the behaviour of code blocks in hybrid programs into threads, which enables the timing analysability of OpenMP/MPI programs.

To model an hybrid OpenMP/MPI program we assume a set τ of n periodic tasks denoted by $\{\tau_1, \dots, \tau_n\}$. More precisely, each task τ_i , $i \in [1, n]$ is a potentially parallel task, composed of a set of n_i segments. A segment is a set of code blocks grouped inside a *parallel region* (i.e. an OpenMP *parallel region*).

Each parallel segment $\tau_{i,j}$, $i \in [1, n]$, $j \in [1, n_i]$ may further be composed of $l_{i,j}$ potentially parallel code blocks (e.g. an OpenMP section, the code inside a parallel for loop, etc.) denoted by $b_{i,j,k}$, $i \in [1, n], j \in [1, n_i], k \in [1, l_{i,j}]$, each code block has a Worst Case Execution Time (WCET) of $C_{i,j,k}$. As an example simply consider as a code block, line 4 in Figure 4, consequently, due to the 4 iterations of the for cycle, we get 4 different code blocks (assuming a chunk size of 1), which can be executed in parallel.

Each segment can be executed by a set of $nt_{i,j}$ threads denoted as $\sigma_{i,j,k}$, $i \in [1, n], j \in [1, n_i], k \in [1, nt_{i,j}]$.

The distributed nature of the programming model being proposed also assumes the existence of a set Ψ of nn distributed nodes Ψ_x, $x \in [1, nn]$.

A task τ_i can be executed in a single parallel node, if the execution requirements (e. g. deadline) of τ_i can be met, on the node (Ψ_x) where it has been released. If τ_i computational requirements exceed the capacity of a single node Ψ_x, then a subset $\mathcal{S}\Psi \subseteq \Psi$ of nodes can cooperate to cope with the requirements of τ_i.

In this Section, we model the execution of hybrid OpenMP/MPI programs, by reducing the code blocks DAG into the corresponding threads DAG. The code blocks DAG may include and model some inherent transmission delays $\mathcal{TD}_{i,j}$ when a task τ_i has been sent from node Ψ_i to a node Ψ_j for being remotely processed.

In Figure 5 we depict a possible execution of the code in Figure 4 on two nodes, a local node with two cores and a neighbour node, also with two cores. This figure depicts the interaction between the OpenMP code and the MPI code required for distribution.

The code blocks DAG can be mapped into a threads DAG, whose scheduling analysis can be performed with any of the currently available state-of-the-art scheduling tests for real-time parallel systems, e.g. [3].

A. Timing behaviour of OpenMP programs

To correctly characterize the OpenMP time behaviour, it is necessary to analyse the process involved in the transformation from high level `#pragma` directives to standard C/C++ code that is finally compiled by the C/C++ compiler. OpenMP is supported by several free and commercial compilers [7], and the implementations between them can vary. In here, we do not intent to address the possible details and differences between OpenMP compiler implementations. We rather prefer to present a general and abstract model of this transformation.

The process of converting OpenMP constructs to multithread code is known as *lowering* the code. This lowered code, is the one that makes the calls to the OpenMP run-time environment. OpenMP compilers make this lowering process in two phases: i) the *pre-lowering* and ii) *lowering.*

The *pre-lowering* phase is in charge of transforming (simplifying) some OpenMP *work-sharing constructs* in other equivalent ones, with the objective of facilitating later processing.

This is the case of the `sections` construct and the `single` construct. In particular, the `sections` construct is converted into an equivalent `for` loop and each `section` construct corresponds to one iteration in the loop. After this transformation, each iteration is scheduled according to the scheduler type in use, which can be defined as `static` or `dynamic`. Also, the `single` construct is

transformed to a `for` loop with just one iteration. When a *parallel region* uses the `single` construct then the `schedule` clause is always defined as `dynamic`.

The *lowering* phase takes the pre-lowered code and actually performs the transformation to C/C++ code. The *lowering* step realizes a transformation known as *outlining* the code. Outlining is the process of transforming lexically existing code into a new procedure and then this new procedure is passed as an argument to the runtime libraries of OpenMP. OpenMP does this outlining process to code encountered inside *parallel regions*. The outlining methods can vary between compilers. But, for our purposes, it is important to note that after this outlining phase the compiler calls the OpenMP run-time which is in charge of mapping code to threads. Therefore, the instructions that are scheduled and processed are the *lowered* ones.

Then, after *lowering* the code, the final mapping from code blocks to threads depends on the scheduler type that is used do assign (map) code blocks to threads.

Whenever the `schedule` clause is defined as `static`, the iterations are assigned in round-robin to threads. In our model, the *chunk_size* is the number of *lowered* iterations inside a `for` loop. When the parameter *chunk_size* is not specified, the *chunk_size* is approximately the same for all threads; equal to the number of iterations divided by the number of threads. However, regardless the *chunk_size*, equation (1) holds for mapping the number of lowered code blocks $b_{i,j,k}$ inside a *parallel region* that are ready for execution (without precedence constraint) into threads, when the `static` scheduler type is use to schedule *lowered* code:

$$nb_{i,j,k}^{max} = \left\lceil \frac{l_{i,j}}{nt_{i,j}} \right\rceil \qquad (1)$$

where $nb_{i,j,k}^{max}$ is the maximum number of code blocks per thread inside a *parallel region*. This is an upper bound on the number of code blocks to be assigned to each thread. Then, if we consider the maximum WCET execution time of a code block $b_{i,j,k}$ denoted as $C_{i,j,k}^{max}$, we can derive an upper bound for WCET $C_{\sigma_{i,j,k}}^{max}$ of a thread being executed in parallel as:

$$C_{\sigma_{i,j,k}}^{max} = nb_{i,j,k}^{max} \cdot C_{i,j,k}^{max} \qquad (2)$$

In case the `dynamic` scheduler type is chosen, *chunk* iterations are assigned to threads on request. In a similar way as a work sharing pool. Whenever a thread finishes processing a *chunk*, it requests another until no more chunks are available. Therefore, the `dynamic` scheduler type can potentially offer better performance, especially when the execution times of the respective code blocks are not uniformly distributed (irregular parallelism). However, the upper bound derived in (2) is also an upper bound whenever the `dynamic` scheduler type is used. For illustrating the reasoning of this, let us suppose two threads $\sigma_{1,1,1}$ and $\sigma_{1,1,2}$ to execute three code blocks $b_{1,1,1}$, $b_{1,1,2}$ and $b_{1,1,3}$, with execution times of $C_{1,1,1}$, $C_{1,1,2} + \varepsilon$ and $C_{1,1,3} + 2 \cdot \varepsilon$, where ε represents a very small execution time quantity; that is, ε approaches zero. Let us suppose that code blocks $b_{1,1,1}$, $b_{1,1,2}$ are being executed by threads $\sigma_{1,1,1}$ and $\sigma_{1,1,2}$, respectively. Then, $\sigma_{1,1,1}$ ends its execution and request the next code block $b_{1,1,3}$, which is the one having the maximum WCET $C_{i,j,k}^{max}$. From (1) we know that each thread has a maximum $nb_{i,j,k}^{max}$ of two. Hence, we can see that in this example $C_{\sigma_{i,j,k}}^{max}$ would not be bigger than $2 \cdot C_{i,j,k}^{max}$. Thus we show that equation (2) also holds as an upper bound for the `dynamic` scheduler.

OpenMP supports other scheduler types, such as `guided`, `runtime` and other variations. But they are implementation dependent and hence we do not consider them in this work.

After the initialization of an OpenMP program, a task τ_i is executed sequentially, and is only composed by the *master thread*. Whenever it encounters a *parallel region*, the *master thread* "forks" and creates a team of nt threads belonging to task τ_i. The number of threads to be created, is explicitly expressed by the `num_threads(nt)` clause.

An example of a typical OpenMP program is depicted in Figure 2. In line number 1, a `#pragma omp parallel for` directive is encountered, which also includes a `num_threads(3)` clause and a `reduction` clause. In this case, three threads are to share the iterations of a `for` loop. Iterations in a parallel `for` loop are divided in *chunks* that are assigned to threads in $\sigma_{i,j,k}$. In this specific example the number of iterations to share is six, and then assuming that the default scheduler kind is `static`, the threads $\sigma_{i,j,k}$ with $k = 1,2,3$ share two *chunks each* in a round robin manner. A possible timeline for the execution of Figure 2 code by three threads is shown in Figure 6. In this figure, we can find three different code segments: τ_1, τ_2, τ_3, with its code blocks. Code block $b_{1,1,1}$ corresponds to serial code being executed prior to the *parallel region*, then code blocks $b_{1,2,1-6}$ represent the execution of the code in line 3, the function `loopcode()`. Code block $b_{1,2,7}$, in segment $\tau_{1,2}$ corresponds to the execution of the reduction clause.

B. *Timing behaviour of MPI communications*

In contrast to the use of threads in OpenMP, MPI uses processes as execution units for implementing two-sided communication. But for modelling purposes, we do not distinguish between threads and processes, and therefore we use the same notation.

Figure 3 depicts an extract of code showing a two-sided communication example in MPI. The execution of the code starts with the initialization of MPI variables and the setting up the of MPI environment; this is done in lines 1-5. Then, the master process $\sigma_{1,1,1}$ initiates a transmission when calling to `MPI_Send` in line 7. A matching `MPI_Recv` is posted by process $\sigma_{1,1,2}$ in line 10. During the transmission

there is a transmission delay that depends on the size of the data to transfer and the network protocol.

In our proposed model, this communication is hidden from programmers as explained in Section IV, but the timing behaviour needs to be considered when designing hybrid OpenMP/MPI programs with real-time requirements. By observing the timeline shown in Figure 5, we can observe that during the transmission of data, to be used by thread $\sigma_{1,1,3}$ there is a transmission delay that depends on the size of the data to transfer and the network protocol. Then, we say that two processes $\sigma_{1,1,1}$ and $\sigma_{1,1,3}$ hosted in nodes Ψ_i and Ψ_j, respectively, incur in a transmission delay $\mathcal{TD}_{i,j}$ for $i \neq j$ and zero otherwise.

It is important to note that this transmission delay is a critical parameter on the ability of the proposed programming model to be able to fulfil the task's deadlines, due to its possible high duration in relation to the execution times on the other machines.

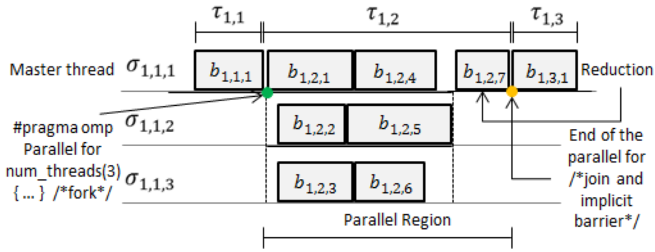

Figure 6. Timing threading execution of a parallel for in OpenMP.

C. Hybrid OpenMP/MPI model

In order to consider hybrid execution, we extend our OpenMP model into a Directed Acyclic Graph (DAG) that allows to fully model the workload. The goal is to provide a DAG that can be handled by a real-time schedulability test for global EDF or partitioned RM (after applying the transformations presented in [1-3]). Since these schedulability tests and their corresponding adaptations are designed for identical multi-core platforms, distribution needs to be integrated, handling transmission delays $\mathcal{TD}_{i,j}$ (with approaches such as [15]), which is outside of the scope of this paper.

The execution time of a task τ_i and its decomposition in threads due to parallelism can be represented by a DAG. A DAG $G(V, E)$ is able to capture the combination of sequential and parallel code blocks in parallel/distributed programs and the possible dependencies between them. Also, it is able to capture possible nested parallelism. Our hybrid model is based on two different graphs $GCB(V, E)$ and $GT(V', E')$, where the first represents the dependencies between code blocks in a program, and the second represents the mapping of such blocks to threads.

The graph $GCB(V, E)$ represents the structure of the program, with the code blocks that may be executed in parallel, as well the code blocks that may be executed serially.

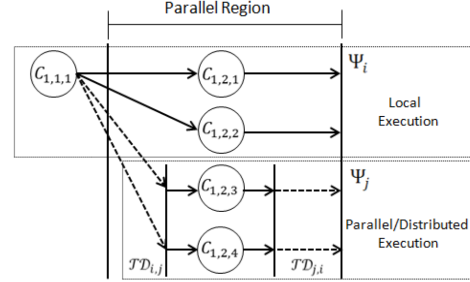

Figure 7. Code blocks DAG $GCB(V, E)$, including comm. delays $\mathcal{TD}_{i,j}$.

This graph of execution is commonly provided by a compiler, and it is known as the *execution tree*, in here we provide a more general approach by using a DAG. The set of vertices in $V = \{v_0, \dots, v_k\}$, represent the set of code blocks $b_{i,j,k}$, and the set of edges $E = \langle (v_0, v_1), \dots, (v_{k-1}, v_k) \rangle$ represent the dependencies between them. If a vertex v_1 precedes v_2, denoted by $v_1 \rightarrow v_2$, indicates that a vertex v_1 must complete execution, before v_2 can start execution. The relation $\rightarrow$ indicates a *predecessor-to-successor* relation.

The dependencies in an hybrid OpenMP/MPI program can be imposed by implicit synchronization points (e.g. `single` constructs, `master` constructs, etc.), explicit *barriers* or memory synchronization (e.g. critical sections, flush operations, etc.); just to mention some, which are related to OpenMP. If there is no precedence relation between nodes v_1 and v_2 we said that they are *logically parallel* and then they may be executed in parallel.

Whenever a graph $GCB(V, E)$, considers transmission delays due to *remote execution*, then these delays $\mathcal{TD}_{i,j}$ can be considered as a precedence constraint and added to the execution time of the predecessor node (Figure 7).

Then, given the graph $GCB(V, E)$, to be able to apply a scheduling algorithm we need to map this code blocks into the graph of threads $GT(V', E')$. Where the set of vertices $V' = \{v'_0, \dots, v'_k\}$ in $GT(V', E')$, represent the set of code blocks $b_{i,j,k,}$ and the set of edges $E' = \langle (v'_0, v'_1), \dots, (v'_{k-1}, v'_k) \rangle$ represent the order of execution of code blocks assigned to threads in $\sigma_{i,j,k}$.

To obtain $GT(V', E')$, we need to traverse $GCB(V, E)$ for obtaining a tree that contains *predecessor-to-successor* relations indicating which code blocks precedes another when assigned to threads. Each branch in the tree corresponds to the execution of successive code blocks in one thread, that is $GT(V', E')$ has exactly the same number of branches an threads (executing in the program).

The traverse mechanism is the well known Breadth-First Search (BSF) algorithm [17]. The BSF algorithm systematically discovers every vertex that is reachable from a source node s. The BSF algorithm has been designed to find the shortest path in a non-weighted graph. However, we are interested in a useful property of BSF, it expands the frontier between discovered and non-discovered nodes uniformly across the breadth. This means that all vertex at

distance k from s are discovered before discovering another vertex from distance $k+1$. This is particularly useful for our purposes, because all discovered vertex may be executed in parallel since they do not have precedence constraints between them. In the BSF algorithm, the discovered nodes that are reachable from s are maintained in a queue before deciding to discover another level in the DAG. This queue can be assigned to threads according to the defined schedule type (`static` or `dynamic`) and respecting the maximum blocks per thread specified in (1).

For example, let us assume the code blocks inside the *parallel region* in the DAG $GCB(V,E)$ as shown in Figure 7, also let us assume that we have four threads to assign the code blocks. After applying BSF algorithm to GCB according to a `static` schedule type we can obtain a DAG $GT(V',E')$ as shown in Figure 8 a). In Figure 8 b) we present the same example but in this case considering that we only have three threads.

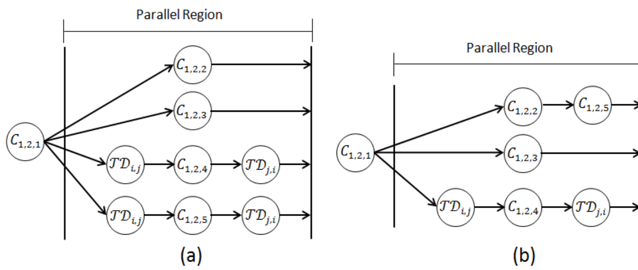

Figure 8. Thread Blocks DAG $GT(V',E')$.

With this approach, application designers are able to transform the applications structure into a model which can be analyzed in terms of timing behaviour.

VI. Conclusion and Future Work

This paper proposed a framework for the development of parallel and distributed embedded systems based on a hybrid programming model. The proposed framework considers a model where programs are written with OpenMP, and where MPI is transparently called for the underlying exchange of data. In this model some of the code executes locally in a mono or multi-core CPU, whilst some components execute distributed in neighbour nodes (potentially again parallelized using OpenMP). We then propose a technique which enables the timing characterization of these applications, transforming the program structure into the execution graph, such that schedulability analysis can be performed.

We are currently focusing our work on finding *work-sharing* algorithms that allow us to determine the most useful way in which to distribute the load between local and remote nodes, at the same time guaranteeing the timing constrains of the applications.

Acknowledgments

The authors thank the anonymous reviewers for the helpful comments and suggestions. 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 and SENODs projects, ref. FCOMP-01-0124-FEDER-015006 and FCOMP-01-0124-FEDER-012988, and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/71562/2010.

References

[1] K. Lakshmanan, S. Kato, and R. Rajkumar, “Scheduling parallel real-time tasks on multi-core processors,” in Proc. IEEE 31st Real-Time Systems Symposium (RTSS 2010), 2010, pp. 259–268.

[2] A. Saifullah, K. Agrawal, C. Lu, and C. Gill, “Multi-core real-time scheduling for generalized parallel task models,” in Proc. of IEEE 32nd Real-Time Systems Symposium (RTSS 2011), November 2011.

[3] A. Saifullah, D. Ferry, K. Agrawal, C. Lu, and C. Gill. “Real-time scheduling of parallel tasks under general DAG model,” online: http://www.cse.wustl.edu/~saifullaha/BIB_Files/dag_parallel.pdf

[4] Y. Nimmagadda, K. Kumar, L. Yung-Hsiang, and C. S. G. Lee, “Real-time moving object recognition and tracking using computation offloading,” in Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010), 2010, pp. 2449-2455.

[5] A, Nuechter, “Parallel and cached scan matching for robotic 3D mapping,” in Journal of Computing and Information Technology - Volume 17, Number 1, 2009, pp. 51-65.

[6] P. T. M. Saito, D. F. Wolf, B. A. Mendonça, K. R. L. J. C. Branco, and R. J. Sabatine, “A parallel approach for mobile robotic self-location,” in Proc. of Fourth International Conference on Computer Sciences and Convergence Information Technology (ICCIT 2009), 2009, pp. 762-767.

[7] L. Nogueira and L. M. Pinho, "Time-bounded distributed QoS-aware service configuration in heterogeneous cooperative environments", in Journal of Parallel and Distributed Computing 69 (2009), pp. 491-507.

[8] OpenMP Architecture Review Board, “OpenMP application program interface V3.1 July 2011,” www.openmp.org/wp/openmp-specifications/, last accessed April 2012.

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

[10] L. L. Ferreira, G. Silva, L. M. Pinho, “Service offloading in adaptive real-time systems,” in Proc. of IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA 2011), 2011, pp. 1-6.

[11] E. Lusk and A. Chan, “Early experiments with the OpenMP/MPI hybrid programming model,” in Proc. of the 4th international conference on OpenMP in a new era of parallelism (IWOMP'08), Rudolf Eigenmann and Bronis R. De Supinski (Eds.). Springer-Verlag, Berlin, Heidelberg, 2008, pp. 36-47.

[12] R. I. Davis and A. Burns, “A survey of hard real-time scheduling for multiprocessor systems,” ACM Comp. Surv., vol. 43, pp. 35:1–44.

[13] P. Serrano, A. Banchs, P. Patras, A. Azcorra, “Optimal configuration of 802.11e EDCA for real-time and data traffic,” IEEE Trans. on Vehicular Technology, vol.59, no.5, 2010, pp.2511-2528.

[14] J. Loeser and H. Haertig, “Low-latency hard real-time communication over switched Ethernet,” in Proc. 16th Euromicro Conference Real-Time Systems (ECRTS’04), 2004, pp. 13- 22, 30.

[15] J.C. Palencia and M. González Harbour, “Offset-based response time analysis of distributed systems scheduled under EDF”, Proc. of the 15th Euromicro Conference on Real-Time Systems, ECRTS, Porto, Portugal, July, 2003, pp. 3-12.

[16] A. Kanevsky, A. Skjellum , A. Rounbehler, “MPI/RT - An emerging standard for high-performance real-time systems”, in Proc. 31th Hawaii International Conference on System Science, 1998, vol.3, pp. 157-166.

[17] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson.. *Introduction to Algorithms* (2nd ed.), 2001, McGraw-Hill.