EFDA–JET–CP(12)03/02

D.F. Valcárcel, D. Alves, B.B. Carvalho, R. Felton, P.J. Lomas, A. Neto,
C. Reux, J. Sousa, L Zabeo and JET EFDA contributors

Parallel Task Management Library for MARTe

Parallel Task Management Library for MARTe

D.F. Valcárcel[1], D. Alves[1], B.B. Carvalho[1], R. Felton[3], P.J. Lomas[3], A. Neto[1], C. Reux[2], J. Sousa[1], L Zabeo[4] and JET EFDA contributors*

JET-EFDA, Culham Science Centre, OX14 3DB, Abingdon, UK

[1]Associação EURATOM/IST, Instituto de Plasmas e Fusão Nuclear, Instituto Superior Técnico, Universidade Técnica de Lisboa, P-1049-001, Lisboa, Portugal
[2]Ecole Polytechnique, LPP, CNRS UMR 7648, 91128 Palaiseau, France
[3]EURATOM-CCFE Fusion Association, Culham Science Centre, OX14 3DB, Abingdon, OXON, UK
[4]ITER Organisation, Cadarache, France
** See annex of F. Romanelli et al, "Overview of JET Results", (23rd IAEA Fusion Energy Conference, Daejon, Republic of Korea (2010)).*

Preprint of Paper to be submitted for publication in Proceedings of the
18th IEEE-NPSS Real-Time Conference, Berkeley, California, USA
11th June 2012 - 15th June 2012

ABSTRACT

The Multithreaded Application Real-Time executor (MARTe) is a real-time framework with increasing popularity and support in the thermonuclear fusion community. It allows to run modular code in a multi-threaded environment leveraging on the current multi-core processor (CPU) technology. Oneapplication that relies on the MARTe framework is the JET tokamak WALL Load Limiter System (WALLS). It calculates and monitors the temperature on metal tiles, Plasma Facing Components (PFCs) that can melt or flake if their temperature gets too high when exposed to power loads. One of the main time consuming tasks inWALLS is the calculation of thermal diffusion models in real-time. These models tend to be described by very large state-space models thus making them perfect candidates for parallelisation.

MARTe's traditional approach for task parallelisation is to split the problem into several Real-Time Threads, each responsible for a self-contained sequential execution of an input-tooutput chain. This is usually possible, but it might not always be practical for algorithmic or technical reasons. Also, it might not be easily scalable with an increase of the available number of CPU cores. The WorkLibrary introduces a "GPU-like approach" of splitting work among the available cores of modern CPUs that is (i) straightforward to use in an application, (ii) scalable with more available cores and all of this (iii) without code rewrite or recompilation.

The first part of this article explains the motivation behind the library, its architecture and implementation. The second part presents a real application for WALLS, a parallel version of a large state-space model describing the 2D thermal diffusion on a JET tile.

1. INTRODUCTION

THE present trend in software development is to take advantage of current multi-core Central Processing Unit (CPU) technology to increase performance [1]. This is accomplished by designing the software in such a way that several tasks are executed in parallel by threads scheduled by the operating system. Real-time software needs, in addition, to guarantee timing deadlines. The task of implementing realtime software that also takes full advantage of multi-core technology can be aided by novel multi-threaded libraries.

The Multithreaded Application Real-Time executor (MARTe) [2], [3] is a real-time framework with increasing popularity and support in the thermonuclear fusion community [4], [5], [6], [7], [8], [9]. It is also one of the software frameworks being considered for the International Thermonuclear Experimental Reactor (ITER) [10]. In MARTe the notion of real-time thread is central to the concept of parallelism. In its development cycle the problem at hand is split into smaller processing units called Generic Application Modules (GAMs) and these run in a sequential fashion in real-time threads, usually implementing an input-to-output chain. Under the Linux operating system the real-time behaviour is achieved by setting the CPU affinity of real-time threads to isolated CPU cores [11], allowing them to run without interference from other user processes.

To take advantage of multi-core processors a MARTe application has to be split into several real-

time threads running in parallel [12]. This is not always achievable and being possible there is the issue of load balancing between threads. One example of this is the case where only some GAMs are meant to be run in parallel and are preceded and/or followed by code that must run sequentially. Even if it is possible to split the problem into several real-time threads, this means that upgrading the hardware to a higher core-count does not bring any advantage unless the number of real-time threads is (manually) increased as well.

Software performance improvement can be achieved in a first instance by instruction level parallelism, but this can only go so far. Applications that explore thread level parallelism have been developed for several years with the aid of standards such as POSIX Threads and OpenMP [13]. With the advent of multi-core CPUs it has become important to take advantage of the extra processing power and these libraries played a pivotal role in this. Moving to a higher-level, task level parallelism has been exploited by recent versions of operating systems to increase software performance in an easy way for developers. Examples are Apple's Grand Central Dispatch [14] and Microsoft's Parallel Extensions [15], the former already ported for the Linux operating system as well. In another front, manycore devices such as Graphics Processing Units (GPUs) can also boost the performance of applications, exploiting task level parallelism, and scientific applications that use them have rapidly emerged [16]. Applications such as data mining [17] and graphical ray tracing [18] can make extensive use of these software and hardware technologies. However these are not well suited for scalable parallel real-time applications, as they were not primarily designed to guarantee strict timing goals.

This created an opportunity and this paper presents a shared memory multi-threaded task parallelisation scheme suitable for real-time applications. This scheme was implemented in C++ in a shared library powered by BaseLib2 [2], [19], named WorkLibrary, that can be used to develop parallel real-time applications. The applications described and demonstrated in this paper run on top of the MARTe framework but development of MARTeless applications is also possible. During the design process the following functional requirements were set:

1) straightforward to use in an application, meaning that it should contain ready to use objects that aid in the implementation of parallel applications;
2) scalable with more available cores without code or configuration rewrite and recompilation;
3) maintain the real-time performance of applications.

In Section 2 the design and implementations details of the WorkLibrary are described. In Section 3 the library is profiled in an ideal situation. Section 4 follows showing a real-world application and Section 5 concludes with the main points to be retained from this paper.

2. DESIGN AND IMPLEMENTATION

Figure 1 shows a diagram for the typical usage in a MARTe application. The design of the WorkLibrary is based on the concept of work items. The idea is that for a given application, part of it, such as an algorithm implemented in a GAM, can be split into several smaller and independent

pieces of code, called work items. This independence can stem from them acting on different data (data parallelism) or executing different tasks (task parallelism). A GAM and its work items can share data via a shared memory mechanism. Work items are organised in a doubly linked list, named a work list, and there is one work list per application.

A real-time thread may contain several GAMs that generate work items. During the initialisation phase of each GAM its work items are created and placed in the application's work list. GAMs are created in the order they are declared in the real-time thread and this means work items belonging to different GAMs are placed in the work list in the same order.

The third fundamental object in the library is the worker thread. Each one cycles over the work list executing work items as represented in the diagram of Fig.2. To preserve GAM execution order it is not enough that they are added to the list in the same order as GAMs are declared in the real-time thread. It is also necessary that the worker threads know when to start executing items that belong to a specific GAM. This is implemented by a lock mechanism in which GAMs unlock their work items for execution ("ready to run") when it is their turn to execute and lock them back again at the end. In this way worker threads halt at the first items of each GAM and wait until they are unlocked. Because each item must only be executed once in each cycle, there is also the notion of ownership and completeness. A semaphore is used to allow only one thread to take possession of a work item and execute it, and a flag is used to mark a work item as completed.

This scheme allows to avoid a priori assumptions on the resources available. Worker threads do not know how many work items are there in the list, they follow the linked list and execute as they go along. GAMs do not know how many worker threads are available, so no assumption can be made on how many work items to create. Worker threads do not know how many they are so they cannot coordinate work execution between themselves and must do it via the ownership/completeness mechanism.

To make the most out of this design worker threads should execute in isolated CPU cores (best performance) and work items that are meant to execute in parallel should run in approximately the same amount of time (best load balance).

It is arguable that idle worker threads could execute work items that are ready to run in GAMs that are to be executed later in the control chain. The problem of such an approach is that the next GAM requiring service would possibly have to wait in the real-time chain for the worker threads to be free, which is far from optimal and could jeopardize the control cycle timing.

3. PERFORMANCE MEASUREMENT

The most convenient way to assess the library's performance is to profile an application that generates known work items. For this purpose a GAM was used to generate a specified number of work items, each one executing a sleep, with no voluntary preemption, of an also specified duration, and the total execution time of this GAM was measured. Tables I and II describe the hardware and the CPU core allocation used for this profile. Three worker threads are allocated and constrained

to individual cores for best performance.

The profile consisted in running this MARTe application performing a scan of 3 parameters: number of worker threads (1, 2 and 3), number of work items (6, 12, 24, 48 and 96) and work item duration (2, 4, 6, 8, 10, 15, 20, 30, 40, 50, 60, 80, 100, 150 and 200μs). For each combination the time it takes to execute all the GAM's work items (named GAM execution time) was measured for 6000 consecutive cycles and the mean and standard deviation calculated. The resulting set of 225 runs provides an insight on the behaviour of the library when applied to the ideal case where:

- the duration of each work item is perfectly known (to the accuracy of the CPU high-resolution timer measurements)
- the work load is perfectly balanced (all work items have
- the same execution duration) there are no idle worker threads (the number of work items is a multiple of the number of worker threads available)

A real situation most likely will not achieve the same results, so profiling each application is advised.

The first analysis that can be done pertains the overhead introduced by the library during the work items execution. This overhead is determined by taking the difference between the measured GAM execution time and the work items' expected total execution time, which is approximately the product of the number of items by the single item duration. Figure 3 shows the overheads as a function of the work item duration. Results show the overhead has no dependency on the work item duration and for this reason averaging can be performed along the work item duration. Figure 4 shows the overhead as a function of the number of worker threads. It can be seen that the overhead is lower as the number of worker threads increases and also that, for a given number of worker threads, the overhead increases as the number of work items increases. These two results are related with the effort each thread has to do in order to execute a work item, more work items contribute to the overhead but are executed in parallel by the worker threads.

Even if the absolute overhead ($\Delta t_{OH} \pm \varepsilon \Delta t_{OH}$) increases with the number of work items its relative impact in the GAM execution time ($\Delta t_{GAM} \pm \varepsilon \Delta t_{GAM}$) gets reduced. This is depicted in figure 5 and can be expressed by the overhead fraction ($f_{OH} \pm \varepsilon f_{OH}$) ratio given by Eq.1.

$$f_{OH} = \frac{t_{OH}}{t_{GAM}} \pm \left[\frac{1}{t_{GAM}} \varepsilon \ t_{OH} + \frac{t_{OH}}{t_{GAM}^2} \varepsilon \ t_{GAM} \right] \tag{1}$$

Figure 5 shows that, for a given number of worker threads, the overhead fraction is reduced when the number of work items increases. This means that the fact that its absolute value increases becomes less significant. When the number of worker threads increases the overhead fraction increases as well for the same number of work items, but it is still a low fraction.

The scale factor, which determines how the results scale with the number of worker threads used, and consequently with the number of cores available, between the execution time for 1 worker thread and for 2 and 3 threads is also an important measure of the library's performance. Measuring the execution time over a period of time it is possible to calculate the mean and standard deviation of

the time measurement $t_X \pm \Delta t_X$ with X = 1; 2; 3 threads. The scale factor S is given by Eq. 2:

$$S = \left(\frac{t_{2,3}}{t_1} \times 100 \right) \pm \left[\left(\frac{1}{t_1} \varepsilon_{t2.3} + \frac{t_{2,3}}{t_1^2} \varepsilon_{t1} \right) \times 100 \right] \quad (\%) \tag{2}$$

Figure's 6 and 7 show the evolution of the scale factor with the work item duration and the number of work items, respectively. Regarding the evolution with the work item duration the scale factor converges to the expected values ($^1/_2$ and $^1/_3$) as the work item duration increases. For this case the convergence appears exponential and this might suggest that for a given application it might not compensate to increase the work item duration beyond a certain point. It can also be said that beyond that point the scale factor is almost the same for any number of work items. Figure 7 also shows that the scale factor always converges to a given value as the number of work items increases, reinforcing the idea set in figure 5 that the overhead is diluted in the total execution time.

Given that 3 CPU cores were available to run the worker threads the question has arisen of whether or not the Linux scheduler could be capable of deciding where to schedule the worker threads execution whilst maintaining similar performance. To clarify this, another test was performed where the 3 worker threads were allowed to run in any of the 3 isolated CPU cores, and thus the decision of where to schedule them was up to the Linux scheduler. Moreover, all the scheduling policies (Round-Robin, FIFO and OTHER) available in pthreads were tested. 96 work units of 20μs each were set to run, the GAM execution time measured for 6000 consecutive cycles and the results are depicted in histogram form in figure 8 and Table III where it can be seen that if only 1 worker thread is used all time measurements match, with only the isolated core run with slightly longer execution time. However using 2 or 3 worker threads there is a performance degradation, growing execution times with higher number of threads. It is likely that the scheduler is not aware that there are less worker threads than isolated cores and ends up scheduling more than 1 thread to the same core, resulting in a worst performance.

4. EXAMPLE APPLICATION

To demonstrate a real application the library was used to implement a parallel calculation of a state-space model. These models are used on a variety of areas and so this example is generic enough so that it can be used as a template for other applications.

The WALL load limiter System (WALLS) [20] is a protection system for the Joint European Torus (JET) [21] that calculates and monitors the temperature on metal tiles keeping them within the allowed temperature range during experiments, as these are subjected to high power loads. The temperature is modelled with state-space models and recent experiments indicate that a 2D model might be required for certain tiles. These 2D thermal diffusion state-space models are large enough so that parallelisation is required to compute several models within the 10 ms cycle time. The idea is not to execute several models in parallel, but to execute a single model in parallel.

A general state-space model in a discrete representation assumes the form of Eq.3:

$$
\begin{aligned}
x^{n+1} &= A\mathbf{x}^n + B\mathbf{u}^n \\
y^n &= C\mathbf{x}^n + D\mathbf{u}^n
\end{aligned}
\tag{3}
$$

where n is the current iteration number, A, B, C and D are matrices, x is the state vector of the system, u its input vector and y its output vector. If n_S designates the number of states, n_I the number of inputs and n_O the number of outputs (which are the dimensions of x, u and y respectively), A has dimensions n_S n_S, B n_S n_I , C $n_O \times n_S$ and D n_O n_I. Lets also define ai as A's i-th line vector, similarly for B, C and D (b_i, c_j and d_j with i = 1, ... , n_S and j = 1, ... , n_O), and the elements xi and y_j of x and y respectively, as in Eq.4:

$$
\begin{bmatrix} x_1^{n+1} \\ x_2^{n+1} \\ \vdots \\ x_{nS}^{n+1} \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{nS} \end{bmatrix} \begin{bmatrix} x_1^n \\ x_2^n \\ \vdots \\ x_{nS}^n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{nS} \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_{nI}^n \end{bmatrix}
$$

$$
\begin{bmatrix} y_1^n \\ y_2^n \\ \vdots \\ y_{nO}^n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \\ c_{nO} \end{bmatrix} \begin{bmatrix} x_1^n \\ x_2^n \\ \vdots \\ x_{nS}^n \end{bmatrix} + \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{nO} \end{bmatrix} \begin{bmatrix} u_1^n \\ u_2^n \\ \vdots \\ u_{nI}^n \end{bmatrix}
\tag{4}
$$

Using this notation a natural parallelisation for the state equation assumes the form of Eq.5:

$$
x_i^{n+1} = \mathbf{a}_i \cdot x^n + \mathbf{b}_i \cdot \mathbf{u}^n \tag{5}
$$

where this expression can be evaluated in parallel for all i = 1, ... , n_S and is the dot product between vectors. Similarly the output Eq.6 can be evaluated in parallel for all j = 1, ... , n_O:

$$
y_j^n = \mathbf{c}_j \cdot x^n + \mathbf{d}_j \cdot \mathbf{u}^n \tag{6}
$$

For this thermal diffusion model $n_S = 70$ and $n_O = n_I = 1000$, meaning there are 1070 independent calculations. Moreover, each of these expressions has 70 + 1000 = 1070 multiplications and 69 + 1 + 999 = 1069 sums resulting in the same number of operations for each evaluation. This information can be used to determine the number of work items in the parallel calculations. In order to balance the workload and at the same time to have work items with a reasonable execution time it was decided that each work item would process 10 of these expressions, resulting in 7 work items for the state computation and 100 for the output computation.

The work items were implemented in a way that minimizes main memory writes, which are expensive operations. This was accomplished by accumulating intermediate results in an internal variable, most likely stored in fast cache, and writing the result to main memory at the end of the work item execution, resulting in an improved performance. This simulation ran applying a constant power

density of $10MW \cdot m^{-2}$ on the tile surface, using a 10ms cycle time and recording the temperature and timing results. Three independent runs were made with a variable number of worker threads (1, 2 and 3) to assess the algorithm scaling. Table IV shows the performance measurements and figure 9 shows it graphically. To notice that the scaling is almost what we expect it to be, $^1/_2$ for 2 threads and $^1/_3$ for 3 threads. Figure 10 shows the calculated temperature distribution in the tile for a given time instant.

CONCLUSIONS

This paper presents the ideas behind and the results of the WorkLibrary, demonstrating that it meets the functional requirements set at the beginning of the project. This project is original in the sense that there is not much work done in the area of task level parallelism for real-time systems, in particular there was no such scheme for MARTe prior to this work.

The tests performed show that the library's performance scales well up to 3 worker threads, but there is no experimental data for more threads. In particular, the example presented for the state-space model scaled very well with an increase in the number of worker threads. The results also show that on one hand the work item duration must not be too short as to get the best scaling; on the other hand the overheads introduced by the library get diluted with the increase of the number of work items. This means that in each application there is a compromise between the number and duration of the work items.

It was also shown by running the same type of tests that the Linux scheduler is not suitable to schedule worker threads in isolated cores, even if the number of threads to schedule is less than the number of free isolated cores. Using the scheduler seems to slightly improve the execution time when there is only 1 worker thread with affinity to several isolated cores.

Although this library helps the real-time application get the most out of a multi-core CPU, it remains very important that each work item is tuned for the best sequential performance. This includes being careful with memory (main and cache) accesses and making the best use of the processing units the CPU provides [22], [23].

During the profiling of the WorkLibrary it was found that better profiling support is required at the library level and this represents one of the next possibilities of improvement. Moreover, it would be interesting to integrate this scheme directly into the MARTe framework, scheduling GAM execution based on the signal dependencies between them.

ACKNOWLEDGMENT

This work, supported by the European Communities under the contract of Association between EURATOM, IST and CCFE, was carried out within the framework of the European Fusion Development Agreement. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

REFERENCES

[1]. Advanced Micro Devices, "Physical cores v. enhanced threading software: performance evaluation whitepaper", 2010, Online: http://www.amd.com/us/Documents/Cores_vs_Threads_Whitepaper.pdf.

[2]. A. Neto, F. Sartori, F. Piccolo, et al., "MARTe: A Multiplatform Real- Time Framework", IEEE Transactions on Nuclear Science, Volume 57, Issue 2, April 2010, pp. 479–486.

[3]. EFDA-MARTe repository, Online: http://efdamarte. ipfn.ist.utl.pt/websvn/listing.php?repname=EFDA-MARTe

[4]. A. Neto, D. Alves, L. Boncagni, et al., "A Survey of Recent MARTe Based Systems", IEEE Transactions on Nuclear Science, Volume **58**, Issue 4, August 2011, pp. 1482–1489.

[5]. D.F. Valcárcel, A. Neto, I.S. Carvalho, et al., "The COMPASS tokamak plasma control software performance", IEEE Transactions on Nuclear Science, Volume **58**, Issue 4, August 2011, pp. 1490–1496.

[6]. L. Boncagni, Y. Sadeghi, D. Carnevale, et al., "First Steps in the FTU Migration Towards a Modular and Distributed Real-Time Control Architecture Based on MARTe", IEEE Transactions on Nuclear Science, Volume **58**, Issue 4, August 2011, pp. 1778–1783.

[7]. P.J. Carvalho, P. Duarte, T. Pereira, et al., "Real-Time Tomography System at ISTTOK", IEEE Transactions on Nuclear Science, Volume **58**, Issue 4, August 2011, pp. 1427–1432.

[8]. L. Zabeo, F. Sartori, A. Neto, et al., "Continuous data recording on fast real-time systems", Fusion Engineering and Design, Volume **85**, Issues 3–4, July 2010, Pages 374–377.

[9]. D.F. Valcárcel, A. Barbalace, A. Neto, et al., "EPICS as a MARTe Configuration Environment", IEEE Transactions on Nuclear Science, Volume **58**, Issue 4, August 2011, pp. 1472–1476.

[10]. B. Gonçalves, J. Sousa, B.B. Carvalho, et al., "ITER fast plant system controller prototype based on ATCA platform", In Press Corrected Proof, Fusion Engineering and Design, http://dx.doi.org/10.1016/j.fusengdes.2012.04.005

[11]. D. Alves, A. Neto, D.F. Valcárcel, et al., "A new generation of real-time systems in the JET tokamak", this conference.

[12]. A. Barbalace, "Emerging Hardware Architectures and Advanced Open- Source Software Technologies for Real-Time Control and Data Acquisition in Quasi-Continuous Nuclear Fusion Experiments", PhD Thesis, January 2011, Università degli Studi di Padova, Padua, Italy.

[13]. I.M.B. Nielsen, C.L. Janssen, "Multi-threading: A new dimension to massively parallel scientific computation", Computer Physics Communications, Volume 128, Issues 1–2, 9 June 2000, pp. 238–244 (http://dx.doi.org/10.1016/S0010-4655(00)00062-X).

[14]. Apple Inc., "Grand Central Dispatch (GCD) Reference", 18 November 2011, Online: http://developer.apple.com/library/ios/documentation/Performance/ Reference/GCD_libdispatch_Ref/GCD_libdispatch_Ref.pdf.

[15]. Microsoft, "Parallel Programming in the .NET Framework", 2012, Online: http://msdn.microsoft.com/en-us/library/dd460693.aspx.

[16]. D. Luebke, "CUDA: Scalable parallel programming for highperformance scientific computing", 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, 14-17 May 2008, Paris, France (http://dx.doi.org/10.1109/ISBI.2008.4541126).

[17]. R. Jin, G. Yang, G. Agrawal, "Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance", IEEE Transactions on Knowledge and Data Engineering, Volume **17**, Issue 1, January 2005, pp. 71–89.

[18]. J. Bigler, A. Stephens, S.G. Parker, "Design for Parallel Interactive Ray Tracing Systems", Proceedings of the IEEE Symposium on Interactive Ray Tracing 2006, 18-20 September 2006, Salt Lake City, United States of America, pp. 187–196 (http://dx.doi.org/10.1109/RT.2006.280230).

[19]. G. De Tommasi, F. Piccolo, A. Pironti, F. Sartori, "A flexible software for real-time control in nuclear fusion experiments", Control Engineering Practice, Volume **14**, Issue 11, November 2006, pp. 1387–1393.

[20]. A. Cenedese, F. Sartori, V. Riccardo, P.J. Lomas, "JET first wall and divertor protection system", Fusion Engineering and Design, Volumes **56–68**, September 2003, pp. 785–790.

[21]. F. Sartori, G. de Tommasi, F. Piccolo, "The Joint European Torus", IEEE Control Systems, Volume **26**, Issue 2, April 2006, pp. 64-78.

[22]. Advanced Micro Devices, "Software Optimization Guide for AMD Family 10h and 12h Processors", February 2011, Rev. 3.13, Online: http://support.amd.com/us/Processor_TechDocs/40546.pdf

[23]. Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual", April 2012, Online: http://www.intel.com/content/www/us/en/architecture-andtechnology/ 64-ia-32-architectures-optimization-manual.html.

CPU	AMD Phenom(TM) II X6 1090T @ 3.2GHz
Memory	Corsair® Dominator® 8GB DDR3 @ 1333MHz
Motherboard	ASRock 890GX Extreme4
Operating system	Linux Fedora Core 14 32-bit with isolated CPU cores
Kernel	Vanilla 2.6.35.9 SMP PREEMPT

Table 1: Specification of the hardware setup used for the tests.

CPU core number	Allocated to
1	Linux Processes
2	Timer for the Real-Time Thread
3	Real-Time Thread
4	Worker Thread 1
5	Worker Thread 2
6	Worker Thread 3

Table 2: CPU core allocation for the threads involved in the tests.

Worker threads	Policy	Minimum (ms)	Average (ms)	Maximum (ms)
1	Isolated RR	1.9538	1.9559 ± 0.0007	1.9598
	Round Robin	1.953	1.9547 ± 0.0005	1.9586
	FIFO	1.9532	1.9549 ± 0.0007	1.9581
	OTHER	1.9536	1.9551 ± 0.0007	1.9583
2	Isolated RR	0.9801	0.9881 ± 0.0034	0.9962
	Round Robin	1.9567	6.4733 ± 2.6241	19.932
	FIFO	1.9567	6.4440 ± 2.6148	19.81
	OTHER	1.957	6.4130 ± 2.6197	19.5254
3	Isolated RR	0.6574	0.6666 ± 0.0024	0.6744
	Round Robin	1.9548	12.4848 ± 2.2569	30.2141
	FIFO	1.9557	12.4902 ± 2.2922	29.8081
	OTHER	1.956	12.4838 ± 2.2581	29.9098

Table 3: Execution times for different thread schedulimg polocies in the LINUX scheduler. RR - Round-robin, FIFO- First in first out.

Worker threads	Minimum (ms)	Mean (ms)	Maximum (ms)	Scaling (%)
1	8.762	8.857 ± 0.0382	8.941	100.0 ± 0.86
2	4.433	4.487 ± .0149	4.524	50.7 ± 0.39
3	2.983	3.014 ± 0.0117	3.055	34.0 ± 0.28

Table 4: Performance measurements for the 2D thermal diffusion model execution.

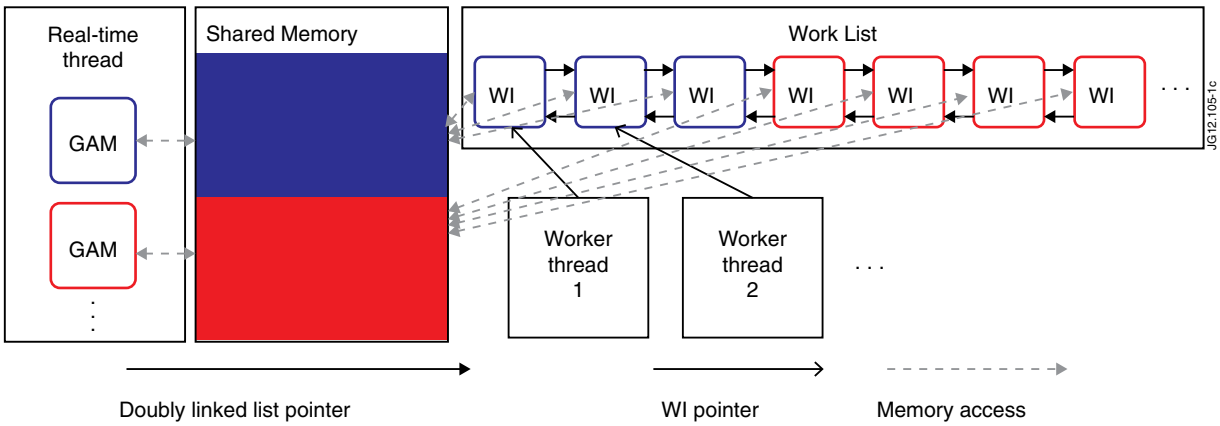

Figure 1: Diagram for a WorkLibrary typical usage within a MARTe application. The work item (WI), work list and the worker thread are the main components of the library. GAMs and their work items can share data by a shared memory mechanism. The work list is a doubly linked list of work items and the worker threads can access work items via pointers.

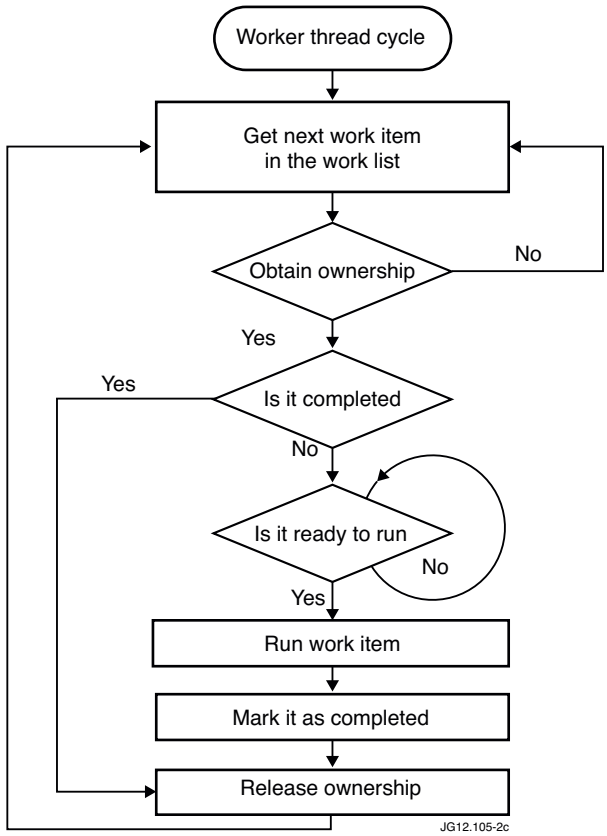

Figure 2: The worker thread execution cycle, the sequence of steps that a worker thread follows to execute the work items while traversing the work list. If a work item is already owned by another worker thread, it moves on to the next work item. If the work item is already completed it relinquishes ownership and moves on. If not, waits for it to be ready to run and continues.

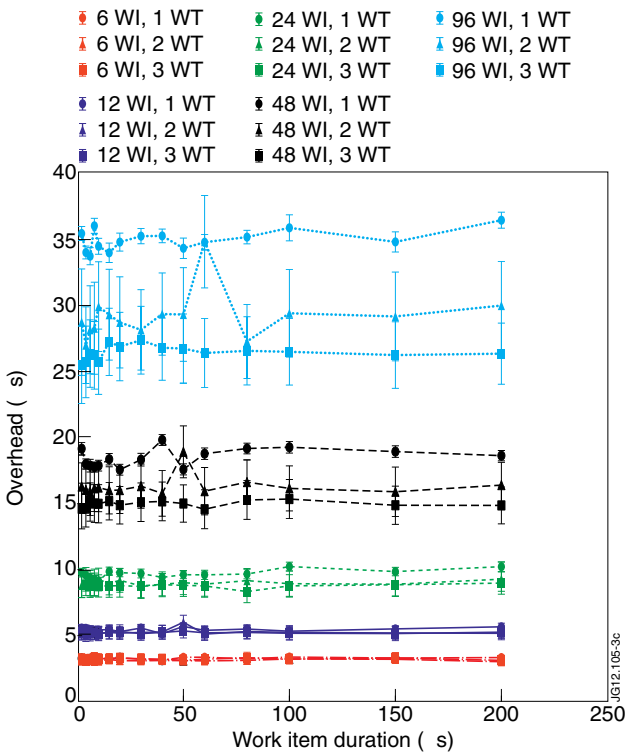

Figure 3: Overhead as a function of the work item duration (WI - Work Items, WT - Worker Threads). The overhead is independent of the work item duration, increases with the number of work items and decreases with the number of worker threads.

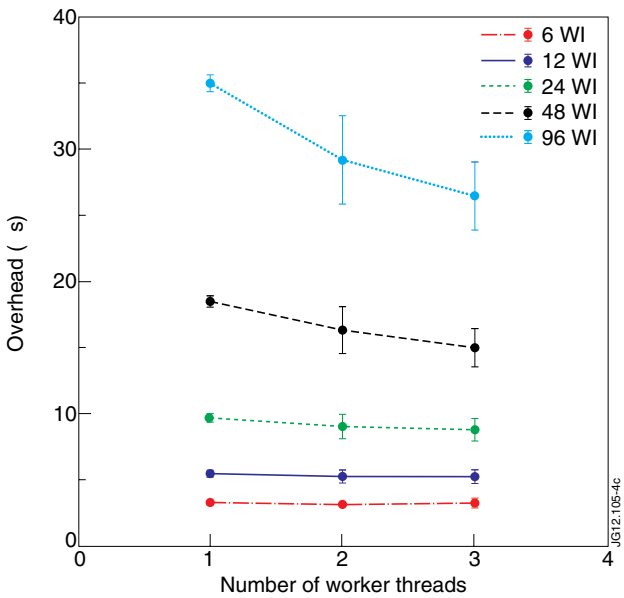

Figure 4: Overhead as a function of the number of worker threads, after data averaging along the work item duration (WI - Work Items). The overhead is lower with a higher number of worker threads and higher with more work items.

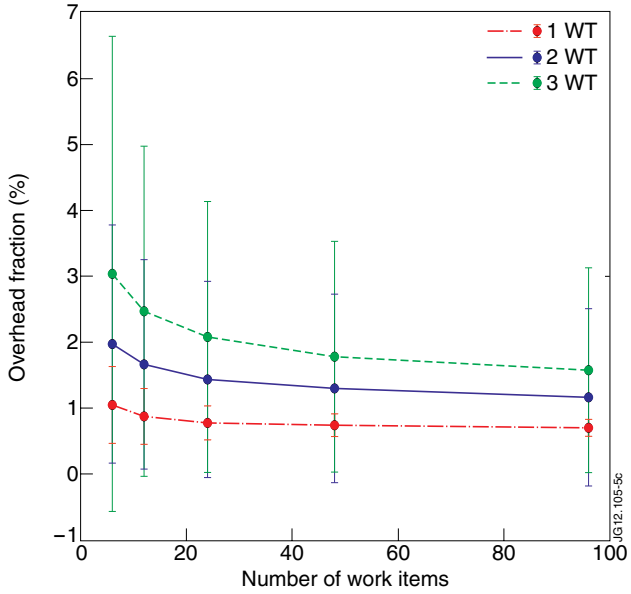

Figure 5: Overhead fraction as a function of the number of work items (WT - Worker Threads). The overhead fraction is lower with an increase of the number of work items and higher with the increase of the number of worker threads.

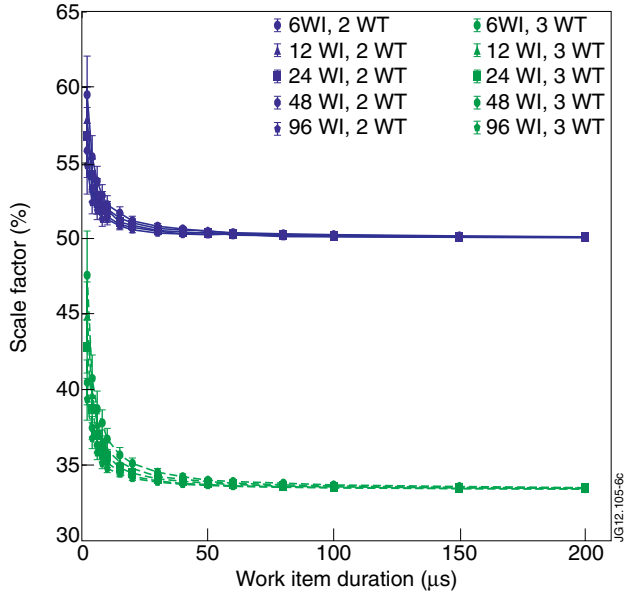

Figure 6: Scale factor as a function of the work item duration (WI - Work Items, WT - Worker Threads). The scale factor converges to the expected value of 1=2 and 1=3 as the work item duration increases.

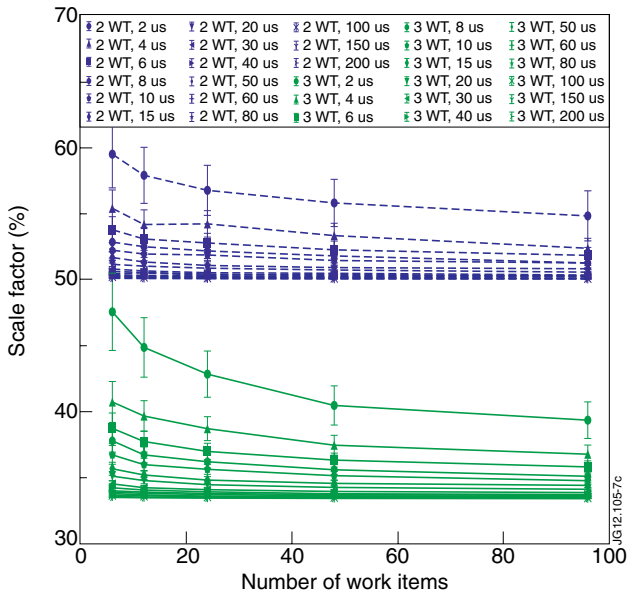

Figure 7: Scale factor as a function of the number of work items (WI - Work Items, WT - Worker Threads). The scale factor converges to a given value when the number of work items increases. However the convergence is closer to the expected values of 1=2 and 1=3 with longer work item duration.

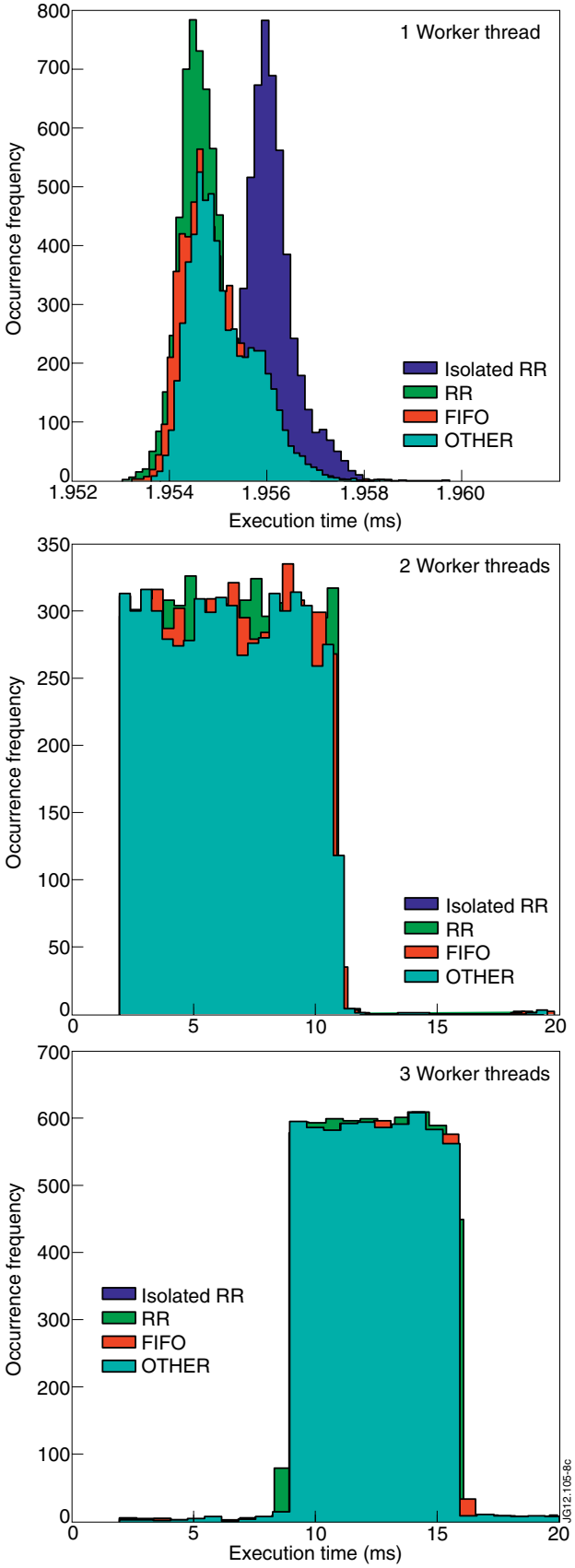

Figure 8: Execution times histogram for the scheduling policy tests. RR - Round-Robin, FIFO - First In First Out. With one worker thread using the scheduler without setting the thread affinity to a single core seems to result in slightly better execution times. When more than one worker thread is considered, the individual allocation of threads to isolated cores outperforms by far the operating system scheduler.

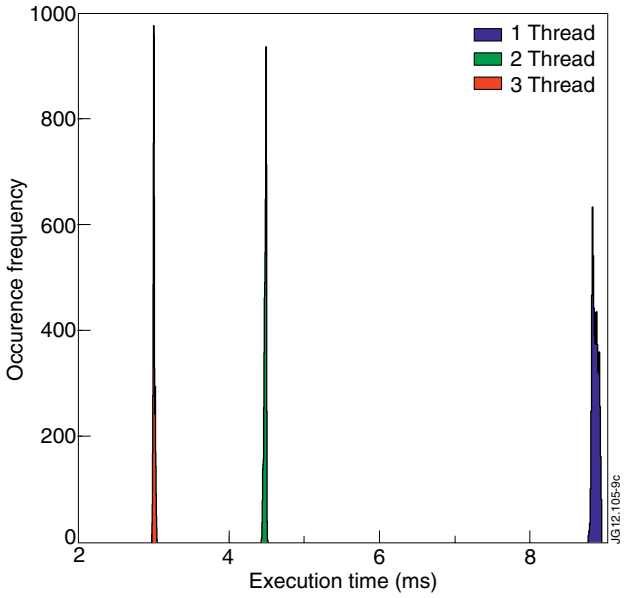

Figure 9: Histogram of the state-space model execution time. The observed scaling with the number of worker threads is close to what is expected, 1=2 and 1=3 for 2 and 3 worker threads respectively.

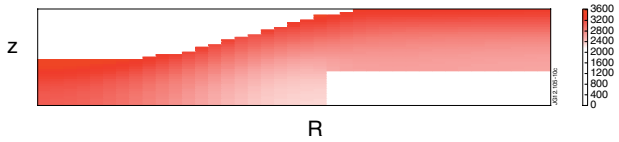

Figure 10: Visual representation of the tile temperature (ºC) in a particular time instant, corresponding to a poloidal cross section of the tile.