Parallel Processing of Real-time Tasks on Multi-core Systems

Saleh Alrashed
Imam Abdulrahman Bin Faisal University
Nasro Allah
nabdullatief@iau.edu.sa

Imam Abdulrahman Bin Faisal University

Research Article

Keywords: Operating Systems, Parallel Programming, Real-time Systems, Fixed-priority Scheduling, IoT, Feasibility Analysis

Posted Date: January 29th, 2024

DOI: https://doi.org/10.21203/rs.3.rs-3891715/v1

Additional Declarations: No competing interests reported.

Parallel Processing of Real-time Tasks on Multi-core Systems

Saleh Alrashed[1*] and Nasro Min-Allah[2]

[1]Management Information Systems Department, College of Applied Studies and Community Service, Imam Abdulrahman Bin Faisal University, P.O. Box 1982, Dammam 31441, Saudi Arabia.
[2]Department of Computer Science, College of Computer Science and Information Technology, Imam Abdulrahman Bin Faisal University, P.O. Box 1982, Dammam 31441, Saudi Arabia.

Abstract

Real-time systems are becoming more and more compute-intensive establishing the need for novel techniques to distribute the workload on multi-core systems adequately. In this work, we divide directed acyclic graph (DAG) of a task into sub-tasks such that each sub-tasks is represented as a thread and hence becomes a candidate to run in parallel on multi-core systems. We begin by adjusting tasks deadlines based on fixed priority scheduling algorithm in such a way that required computation is done much faster on multi-core system by exploiting potential parallelism that exists in periodic tasks. We assign modified deadlines to tasks based on the concept of span and prove that such task sets remain schedulable with modified deadlines by utilizing adequate number of cores. This approach allows parallel processing of periodic tasks employing a minimum number of required cores. Our experimental results show that for embarrassingly parallel code, deadlines of unit lengths provide linear speedup while there is no improvement when a task can not be divided into sub-tasks and hence one processor suffice to deal with serial execution of the task. We run our experiments on synthetic task set of various system utilization. Our results show that significant speedup is achieved when the task system exhibits high degree of parallelism.

Keywords: Operating Systems, Parallel Programming, Real-time Systems, Fixed-priority Scheduling, IoT, Feasibility Analysis

1 Introduction

Real-time systems impose strict timing constraints on tasks and hence can not rely on statistical models. Instead, verified mathematical models are needed to ensure that deadlines of tasks are never missed, even under worst case scenario. This worst-case scenario puts limitations and requires ample resources to handle such scenarios, though while running the actual resource usages might be much lower than the anticipated worst case. Unlike non-real-time system where applications can exploit multi-cores system, care is needed for real-time system to guarantee that deadlines are always respected [1]. Especially in latest applications such medical or other IoT based applications, more and more data is generated in real-time that needs efficient processing on latest multi-core systems.

Feasibility analysis of real-time system are divided into two main types of priority driven algorithms: fixed priority and dynamic priority. With an operating systems using fixed-priority scheduling algorithm priority, a priority once assigned to a task remains fixed. On contrary, priorities do change at run time under dynamic priority algorithms [2]. Fixed priority scheduling is superior when it comes to system predictability. A real time system must provide predictable behavior. Due to its reliability and simplicity, Rate Monotonic (RM) scheduling [2] has become the de-fecto standard for real time systems and adopted by Boeing, General Dynamics, General Electric, Honeywell, IMB and NASA etc [3][4].

With multi-core systems, performance gains are possible sometimes but need considerable work is needed to analyze the code if it can run in parallel. Performance gains are high for embarrassingly parallel systems where performance is linked with number of cores, however, in the programs where majority of the code run in serial fashion, performance gains are negligible and even worse, programs degrades due to communication and synchronisations overheads involved. Almost, all computing machines even cellphones are multi-cores today and like other applications, real-time systems also run successfully on such multi-core systems. However, it is complicated to take full advantage of cores when tasks have hard deadlines. With hard deadlines, missing an associated deadline is not permitted under any circumstances.

In this work, we propose a technique that successfully executes task with strict relative deadlines to run on multi-core systems. Each task generate a number of jobs which becomes a run-able entity with its own CPU utilization demand and corresponding absolute deadline. We divide a job into sub-jobs and execute these sub-jobs in parallel as long as timing restrictions are intact. We also assign absolute deadlines based on the longest threads among the running sub-jobs. It is worth noting that when number of processors approaches infinity, system performance is dominated by the span–the longest time needed to execute a parallel path of computation by a thread [5].

Initially, we assume a periodic task set with strict deadlines and then distribute the task computation on multiple threads which can run in parallel.

This strategy helps in completing tasks execution much early than the originals deadline and hence provides a time window to adjust deadlines on the fly. A limiting factor in our work is the long threads which takes maximum time as such threads run serially and irrespective of number of available processors, a task can not be completed before this its corresponding deadline. We create Directed Acyclic Graph (DAG) of a periodic task, calculate the work, and then identify the span. Based on these factors we run the task in parallel by utilizing an adequate number of processors.

The rest of the work is organized in the following order. Section 2 discusses background work and related to real-time system followed by feasibility tests. In Section 2, we also discuss the problem of tasks computation is shown through DAG to highlight the impact of span on task execution on multi-core system. The main results are shown in Section 3 and 4 where theoretical results are established to run real-time system on multi-core system under a fixed priority scheduling algorithm. Experimental results are given in Section 5 and finally, the paper is concluded in Section 6.

2 Background and Related Work

Let $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ represent a non-concrete periodic task system having periodic tasks. A non-concrete periodic task τ_i recurs and is represented by a tuple (c_i, d_i, p_i), where c_i, d_i, p_i represent the computation time, relative deadline, and task period, respectively . In our model of a hard real-time task set, each task τ_i generates a job j_i at each integer multiple of p_i and each such job has an execution requirement of c_i time units that must be completed by the next integer multiple of p_i. Moreover, all tasks immediately get ready for execution on uniprocessor as soon as they are released and all tasks overheads, such as task swapping times etc. are subsumed into task computation times. Furthermore, we assume that initially, all of the tasks arrive simultaneously at $t = 0$.

An important issue in the design of real-time systems is of timing correctness. To handle the timing constraints, optimal scheduling of tasks on the given platform is needed. Tasks in real-time systems are scheduled using a priority based algorithms. These algorithms assign priorities to task based on some predefined criteria such as activation rate or deadline etc. Priority based scheduling algorithms for real-time systems fall into two major types: fixed priority and dynamic priority [6, 7]. Under a fixed-priority algorithm, fixed priorities are assigned to all jobs in each task. In contrast, dynamic-priority scheduling algorithms assign dynamic priorities to individual jobs and hence can vary at run time. In theory dynamic algorithms are considered superior due to its higher system utilization capability, such algorithms become unpredictable when transient overload occurs [8–10]. Fixed-priority scheduling algorithms offer reliability and simplicity, and hence we only consider fixed-priority scheduling in this work.

For validating timing constrains under fixed priority systems, feasibility analyses — given a real-time application and processing resource, determining whether it is possible to meet all the deadlines under RM scheduling algorithm — are performed to achieve system predictability. In their seminal paper [2] in 1973, C. L. Liu and J. W. Layland formalized real-time scheduling theory and ended up with a classic Rate Monotonic (RM) scheduling policy. This policy is optimal for case when task deadlines and periods are the equal. Authors in [2], showed that the task set is always feasible under rate monotonic scheduling scheme if the system utilization is below a threshold value of 69%. Consequently, in the last 40 years that have followed, assumptions were relaxed. With such adjustments, the impact of rate monotonic scheduling policy and feasibility analysis has been reported in literature. Accordingly many feasibility conditions were presented [6–8, 11–15].

RM assigns static priorities on task activation rates (periods) such that for any two tasks τ_i and τ_j, priority $(\tau_i) >$ priority $(\tau_j) \Rightarrow$ period $(\tau_i) <$ period (τ_j), while ties are broken arbitrarily. RM is optimal in the sense that if any task set can be scheduled with a fixed-priority assignment scheme, then the given task set also can be scheduled with a rate-monotonic scheme. Due to its implicit characteristics such as simplicity and reliability, the RM has become the *de facto* standard supported by the USA Department of Defense and many other organization/manufacturers, such as IBM and General Motors [16].

In the rest of the paper, the task model refers to the above task model (implicit deadline model), which is well studied in literature. Rate Monotonic Analysis (RMA) is made to determine RM task feasibility. A brief discussion on these tests is provided in the following. The workload constituted by τ_i at time t, consists of its execution demand c_i as well as the interference it encounters due to higher priority tasks from τ_{i-1} to τ_1, and can be expressed mathematically as

$$w_i(t) = c_i + \sum_{j=1}^{i-1} \lceil \frac{t}{p_j} \rceil c_j \tag{1}$$

A periodic task τ_i is feasible if we find some $t \in [0, t]$ satisfying

$$L_i = \min_{0<t\leq p_i} \left(w_i(t) \leq t \right) \tag{2}$$

The time complexity of the above condition depends on both the number of tasks and maximum task period i.e., $O(np_n/p_1)$.

3 Parallel Execution of Real-time Systems

Quality of a parallel algorithm can be defined by two metrics called work and span. Both metrics are important because they give limits to parallel computing and introduce the notion of work. Parallel algorithms have the challenge of being fast, but also to generate the minimum amount of extra

work. By doing less extra work, they become more efficient. In our model, work is the total time needed to execute a parallel algorithm using one processor; denoted as T(n, 1), while span (also called depth) is defined as the longest time needed to execute a parallel path of computation by one thread; denoted as T(n, p). Span is the equivalent of measuring time when using an infinite number of processors.

In a parallel program, one of the limiting factor is the longest thread which will occupy one of the processors and hence the all other cores will need to wait for this thread to complete and hence this longest thread dominate the execution time of the overall program. It is worth noting that a thread in our model can not be broken into small parts and hence considered the smallest portion of the program that can run on a core. Using fork-join approach, the remain threads will complete early but cant reduce the run-time of the program due to this bottleneck of longest thread. Irrespective, of the number of cores, the program run-time depends on this thread. In embarrassingly parallel code, all threads of equal size and run-time reduces with adding more cores. To highlight the role, of longest thread, we introduce the work and span law in the following.

According to work law [5], the running time of a parallel algorithm must be at least 1/p of its work. The work law equation states the first lower bound: $T(n,p)/geT(n,1)/p$, where $T(n,1)$ is the time needed to execute a parallel algorithm using one processor, $T(n,p)$ is time needed to execute a parallel algorithm using p processors, while p represents the number of processors. With the work law, one can realize that parallel algorithms run faster when the work per processor is balanced. That is, the running time of a parallel algorithm must be at least 1/p of its work. With the work law, one can realize that parallel algorithms run faster when the work per processor is balanced. The span law defines the second lower bound for T(n, p) as $T(n,p) >= T(n,\infty)$, where $T(n,1)$ is the time needed to execute a parallel algorithm using infinite number of processors i.e., span. This means that the time of a parallel algorithm cannot be lower than the span or the minimal amount of time needed by a processor in an infinite processor machine. In graph theory, DAG is used to represent an acyclic graph with no directed cycles. In general scheduling DAG is an NP-hard problem. To construct a program that has DAG, it is useful to add labels to the strands to indicate the number of milliseconds it takes to execute each strand. In data analysis, normally jobs consist of many stages and can be expressed as DAG. The precedence relationships between stages cause scheduling a very challenging task.

In Fig. 1, we represent DAG of a parallel structures of a general parallel task and assuming the numbers associated with edges represent time in milliseconds. In Fig. 1, the total amount of processor time required to complete the program is the sum of all the numbers. This value is defined as the work. So for the given DAG, the work is 128 milliseconds for all 12 strands shown in Fig. 1. In other words, if the program runs on a single processor, the program

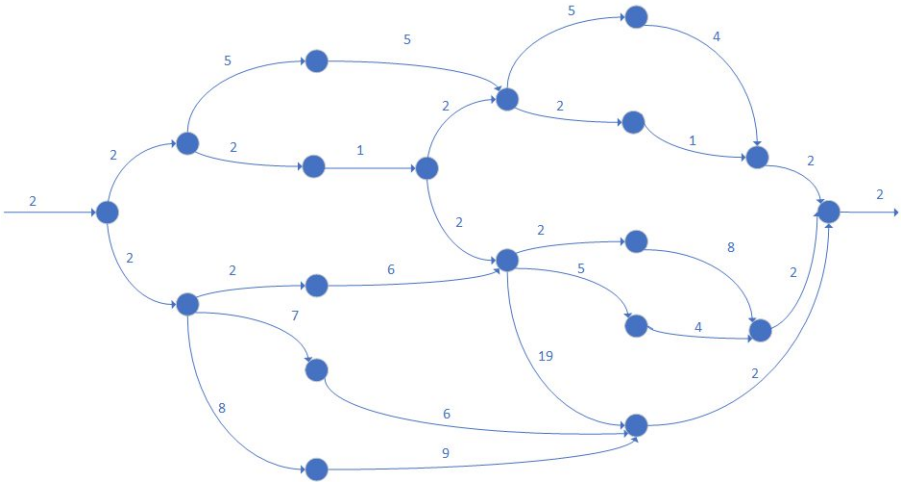

Fig. 1 Parallel Flow of Threads of a Task τ_i

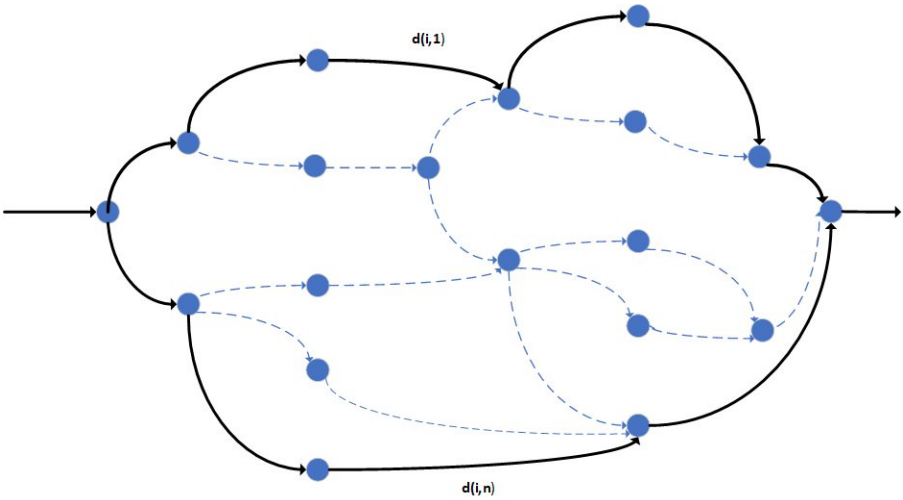

Fig. 2 Dividing Deadline into Multiple Sub-deadlines

should run for 128 milliseconds. Fig. 2 shows the breakdown of a job into sub-jobs where each job as its own deadline and these are handled by two threads responsible for the running parts of the code following different paths. We now extract, the most expensive route from the beginning to the end of the code which is the span. For this DAG, the span is 29 milliseconds, as shown in the following Fig. 3.

We can now express parallelism for Fig. 1 as Parallelism = Work/Span= 2.69. This shows that the program achieve reasonable speedup on 3 cores and any further additional cores will be useless. Rather it can degrade performance in actual with synchronization and associated communication overheads.

After extracting parallelism, we calculate speedup for the given DAG. Speedup can be described as one of the most important actions in parallel computing as it actually measure how much faster a parallel algorithm runs with respect to the best sequential one. This measure is known as speedup and it is the gain in speed made by parallel execution compared to sequential execution. For a problem of size n, the expression for speedup is: $SP >= w/\psi_i$

This speedup becomes an upper bounded when n is fixed because of the work law. Let c be the fraction (in a/b form or as a real number) of a program that is parallel, $(1 - c)$ the fraction that runs sequential and p the number of processors. According to Amdahl's law [5] when parallel portion is dominant, the speedup is more , especially when $P = \infty$, then $SP = 1/(1 - par)$, where par is the parallel portion of a task. When $p \to \infty$, no matter how many processors we use, we would not expect to gain any more speedup than the reciprocal of the serial fraction.

When the speedup increases linearly as a function of p, then it means that the overhead of the algorithm is always in the same proportion with its running time, for all p. In the particular case, we consider ideal speedup or perfect linear speedup. We expect that speedup cannot be better (larger) than linear, and indeed should be smaller. If the entire work of the sequential program could be evenly divided among the p processors, they could all complete in $1/p$ time units. But it is unlikely that the work could be divided evenly; programs tend to have a sequential part, such as initialization or reading data from or writing results to sequential files. Even if the program could be evenly divided among p processors, the processors would probably have to coordinate their work with each other, which would require extra instruction executions beyond the sequential program.

When there is no dependence constraints for task, the main attraction is how to execute tasks in parallel and this steps is straightforward. We now study the impact of parallelism portion of code on the speed up. For a single core machine, whatever is the situation (parallel or no parallel code), the speed up is always 1. Definitely, for the maxim performance, we assume $P = \infty$ and thus the speedup becomes: $Sp = 1/(1 - c)$, where $1 - c$ is the parallel portion of the code. Lets assume $(1 - c) = 0.1$ i.e., parallel portion is=10 %, Speedup=1.11 (as compared to 1.0 on single core. So if you keep p=infinity, the speed is still almost the same. In other words, not much gains are achieved when we increase the processors but the code is only 10% parallelizable. For improved speed up, the motivation is to develop another algorithm that allows more parallelism in code. Again, lets put c=0.5, where 50% code can run in parallel, then $Sp = 2$ and instead of infinite number of processors the speedup remains 2 which suggest more more than 2 processors could hardly bring any improvement. Even when we add more processors to this computation, say p=100000, the speed will remain 2 for the above case as $Sp = 1/((1-c)+c/p) = /((1 - 0.5) + 0.5/100000) = 2.000005$. It is worth mentioning that with 2 processors number of the speed up is 2 while by adding 100000, the speed up

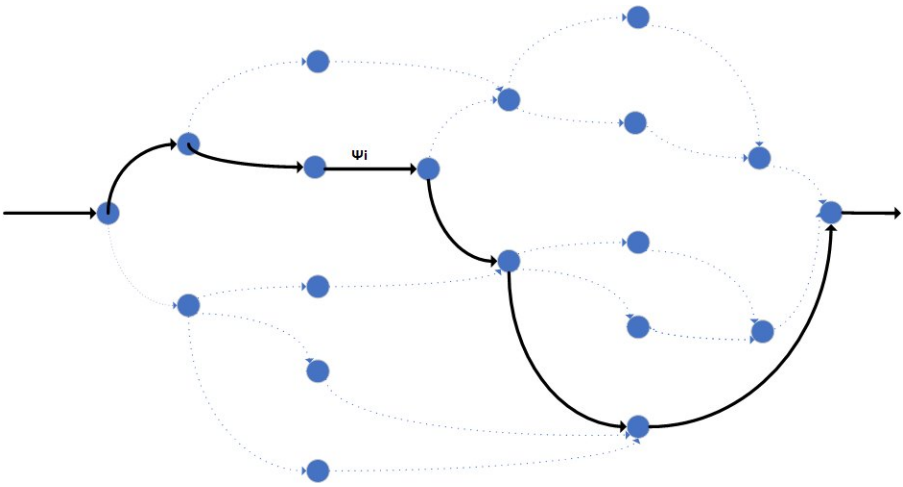

Fig. 3 Identification of Span in Parallel Structure of Program

improvement is only 0.000005 and this is also when coordination among the processors is subsumed to zero.

We subsumed all the serial code before and after the parallel region into ψ_i and it can be noted task τ_i can not be completed before time ψ_i whatever is the number of cores. We now introduce the modified task which has computation demand $c_i^{'}$ and deadline ψ_i i.e., $\tau_i^{'}(c_i^{'}, \psi_i)$, where $c_i^{'} = c_i(seri) + c_i(para)$. Here $c_i(serial)$ represents the serial execution while $c_i(para)$ denotes of the parallel execution of a task. When $c_i(seri) >> c_i(para)$ then single CPU might be sufficient otherwise multi-core is better alternative as it can lower the $c_i(paral)$ part with increased number of cores.

4 Feasibility Test for Parallel Systems

It can be seen from Fig. 1 that multiple threads have now separate deadlines and we should identify the longest thread. In a serial execution of a task τ_i, it is desirable to minimize d_i so that task can be completed as soon as possible, however d_i remains the same under single processor system. While in parallel execution, there is a possibility that the task tau_i can be divided into sub-tasks so that multiple processing units could execute a sub-tasks in parallel. Ideally the deadline d_i should be uniformly divided among the existing p processor as d_i/p. In practicals system such embarrassing parallelism (systems where workload can be perfectly divided among cores) is rare and for periodic task, it is extremely rare. If a task deadline $d_i = 10ms$ and it runs on single core machine so it takes $10ms$ to complete while ruining the same task (assuming embarrassingly parallel) on 10 core machines will take $1ms$ as $d_i/p = 10ms/10$. This observation suggests to distribute the workload c_i of a task τ_i among cores so that deadlines are respected and the task gets c_i time before d_i on p processors. In Fig. 2, we show two deadlines only d(i, 1) and d(i, n), where

d(i, 1) is the first thread that has to be completed by d(i, 1) while last thread has a deadline $d(i, n)$ for a task τ_i.

Among all threads, the longest thread is the one which will completed last even if executed in parallel and all other threads are expected to be complete before this point. Since this is the worst combination for any thread in task τ_i , we represent such thread by ψ_i in Fig. 3. Now, ψ_i is the critical path which can be visualized as serial execution of the thread and can be derived as : $\psi_i = max(d(i, j))$, where "i" is the i-th task and "j" represents Thread-j in all possible threads of task τ_i. The minimum number of threads needed to execute this workload can be written as: $p_{threads} = \lceil w/j \rceil$, and hence we run each thread on a separate processor where p processors are sufficient to run the task successfully. With such representation of span, a task can not be completed before ψ_i even on infinite number of processors.

With the new deadline of ψ_i, we need to ensure the task τ_i is schedulable by ψ_i. We need to ensure the task τ_i remains schedulable and hence the term $\sum_{i=1}^{n} c_i(para)$ which is multi-core dependent and plays critical role in feasibility analysis. When $\psi_i = d_i$, there is no parallel section in the task computation and when $\psi_i << d_i$, there are more opportunities to be exploited due to parallelism. Since ψ_i is the longest path, no other path can be greater than ψ_i and hence one a homogeneous system, the other core can not take more than ψ_i time in worst case.

For non-real time systems, especially for embarrassingly parallel problems, the increased number of processors has huge impact on workload. However, in real-time systems with hard deadlines, jobs can not be assigned without guaranteed timing constraints. This requirement posses a question of how to determine feasibility of task with known $c_i(w)$? To answer this question, we now modify feasibility condition provided in [17] for RM by integrating $c_i(w)$.

Theorem 1 *A task τ_i is schedulable iff $w_i(t) <= t;\ t \in S_i, S_i = lpj,;\ j = 1, ..., n, l = 1, ..\lfloor \psi_i/pj \rfloor$.*

Proof Let $d_i < \psi_i$ and task is schedulable. Thus, the inequality $[\psi_i/pj] < [di/pj]$ holds. Since it is not possible to happen a due to the fact that τ_i as d_i can not be satisfied. So, when $c_i(ser) + c_i(para) < t$ then $p \to \infty$. As $c_i(w)$ is not influenced by $p \to \infty$, while $c_i(para)$ becomes negligible in this case. Then we can say a task on $p \to \infty$ is schedulable if $c_i(ser) < t$. This concludes the proof. □

Since two task τ_i and τ_{i+1} are schedulable only if $c_i(\psi_i) + c_{i+1}(\psi_{i+1}) <= t$ and a task set of size n tasks is also schedulable.

Corollary 1.1 For a periodic task τ_i, d_i is always larger or equal to ψ_i.

Proof Lets assume $\psi_i > d_i$ and τ_i is schedulable. So,

$$c_i + \sum_{k=1}^{i-1} \lceil \psi_i / p_k \rceil c_k \leq w_i(\psi_i) \tag{3}$$

Hence, $\psi_i > d_i$, then $\lceil \psi_i / p_k \rceil c_k > \lceil d_i / p_k \rceil c_k$ which is incorrect as for $\forall k$, $(\psi_i / p_k) \leq (d_i / p_k)$ always holds. □

Theorem 2 *A task τ_i is always RM schedulable on a multi-core system iff:*

$$w_i(\psi_i) <= \psi_i \tag{4}$$

Proof We prove Theorem 2 by contradiction. Lets assume τ_i is RM-schedulable on multi-system while the expression does not hold. So,

$$w(\psi_i) > w_i(d_i)$$

$$c_i + \sum_{k=1}^{n-1} \lceil \psi_i / p_k \rceil c_k > w_i(d_i)$$

Since,

$$\sum_{k=1}^{n-1} \lceil \psi_i / p_k \rceil c_k = c_i + \sum_{k=1}^{n-1} \lceil \psi_i / p_k \rceil c_k \tag{5}$$

As $\psi_i \leq d_i$, therefore,

$$\sum_{k=1}^{n} \lceil \psi_i / p_k \rceil c_k > \sum_{k=1}^{n} \lceil d_i / p_k \rceil c_k \tag{6}$$

which contradicts our assumption that the task is RM-schedulable and hence concludes the proof. □

Corollary 2.1 An ordered set of periodic task is always RM-feasible on a *SP* multi-core system iff: $max_{(1 \leq i)}\{(w_i(\psi_i) <= \psi_i)\}$

Proof It follows directly from Theorem 2 for the entire task set. □

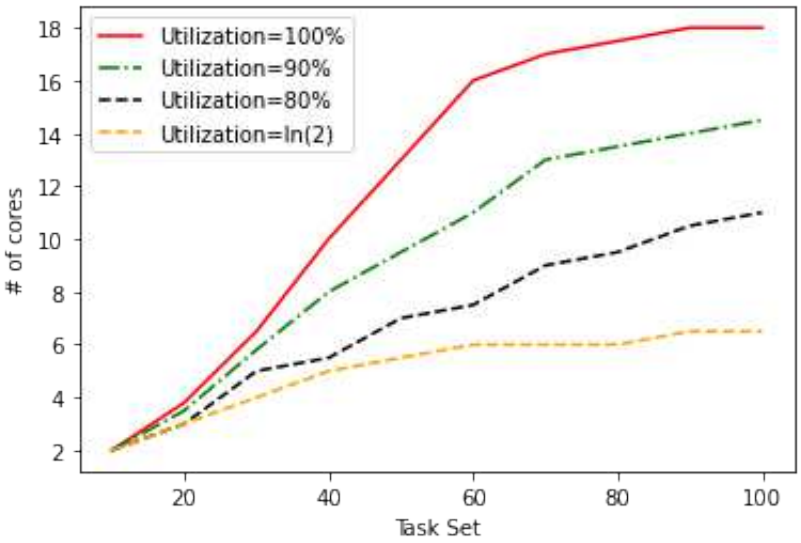

Fig. 4 Effect of System Utilization on Number of Cores

5 Experimental Results

We generated random task sets with varying task deadlines under uniform distribution. According to respective deadlines, task computation values were extracted again using uniform distribution. Based on task periods, RM priorities were assigned to tasks. For simplicity, we normalized task deadlines on a scale of 1 to 10. We run the experiment 200 times and average values are plotted in Fig. 4 and Fig. 5.

In Fig. 4, we study the effect of system utilization on the overall schedulability of the task set and the number of processors need to keep system feasibility intact. When utilization is low then it can be seen that all task are schedulable. In such cases the utilization of the task set is not more than $ln(2)$. By increasing the utilization up-to 80% , a few task can miss the deadlines, however the number is negligible. By increasing utilization further to 90%, a large number of tasks start missing deadlines and hence need more processors. Since RM scheduling theory is used in this work, system feasibility is questionable at higher utilization. By introducing ψ_i, we split a deadline into multiple threads and try to respect timing constraints by ruining these threads on available cores. As seen in Fig. 4, after a certain increase in utilization the number of cores become irrelevant as the task size is also increasing but if we could evenly divide a given task among available cores, there are speed gains and few cores are needed to run these tasks by respecting associated deadlines.

When $\psi_i = w_i$ then there is no parallelism in tasks and only one one thread is running and hence executing this thread on multi-core system does not bring any benefit. On the other hands, when ψ_i is large, threads takes more time to complete. In our case, the shorter is the psi_i, then better it is as the task

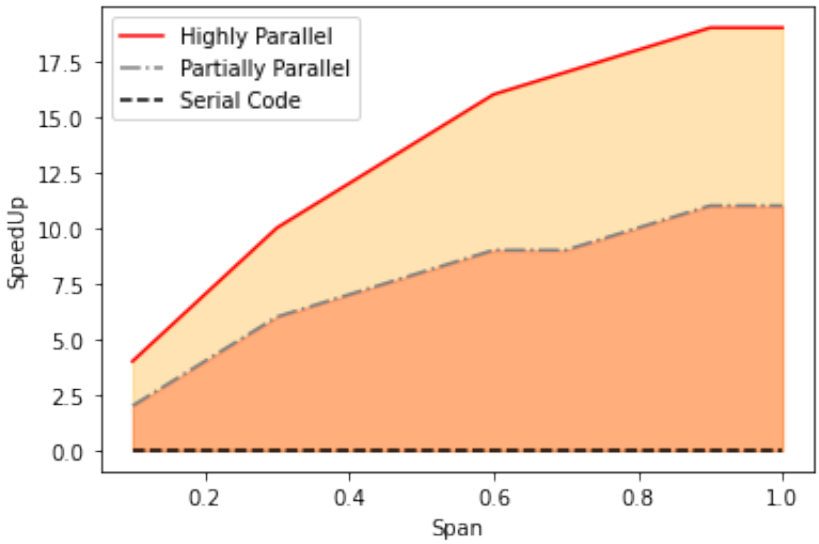

Fig. 5 SpeedUp vs Degree of Parallelism in Tasks

can now complete early but it will need more processors to complete. With linear speed up, $\psi_i = p_i$ where ψ_i is a fraction number. When we keep $\psi_i = 1$ then w_i processors will be needed and each processors will perform a unit of the workload, however making $\psi_i < 1$ will need more processors i.e. when $\psi_i \to \infty$, then $p \to \infty$ which is unreasonable assumption. It can be seen from Fig. 5 that when ψ_i is shorter the d_i then there exists more opportunities for parallelism. This trends goes on to unit value when ration between ψ_i and p is 1 which is an ideal case as the code is highly parallel. On the other side, when both ψ_i and d_i are the same and suggests lack of any parallel component and hence one thread is sufficient to run on a single core due to sequential nature of the task. For tasks when ψ_i is smaller, speedup is insignificant as little or no parallelism is available, however when ψ_i is increased, there are more opportunities for parallelism and hence speedup is more as shown in Figure 5. We drawn a boundary as partially parallel to divide tasks which posses less parallelism are shown between serial and partially parallel while tasks with more parallelism are group between partially parallel and highly parallel regions. When $pthread = 1$, this situation is shown as serial code because there is no parallelism as deadline can not be divided into smaller deadlines without jeopardising the task feasibility. By relaxing deadlines, more and more speedup is achieved and when restrictions is removed fully, the behaviour leads to perfectly linear performance. It can be seen that initially performance grows when deadlines are larger but irrespective of degree of parallelism, speedup becomes flat after achieving a certain degree of performance gain as more speedup becomes very hard which is aligned with our theoretical results.

6 Conclusion

We developed a mechanism to run real-time systems with strict deadlines efficiently on multi-core systems. The original deadlines were divided into shorter deadlines which identified parallelism in a task. Based on inherit parallelism, sub-tasks were then presented to a scheduler that run the code in parallel (where possible) and offered enhanced speedup. With such adjustments, we proved that deadlines were respected by running the code on adequate number of processing cores. As a future work, it will interesting to study parallel execution of periodic tasks by considering dynamic voltage scaling to identify system speed such that deadlines are respected while running the task set using appropriate system speed.

7 Declarations

Ethics approval and consent to participate -not applicable.

Consent for publication Authors hereby provide consent for the publication of the manuscript detailed above, including any accompanying images or data contained within the manuscript. All study participants, or their legal guardian, provided informed written consent prior to study enrollment.

Availability of data and materials –not applicable.

Competing interests The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding not applicable.

Authors' contributions N.M. Allah sketched the idea of parallel execution by considering span. S. Alrashed derived theorems and performed experimentation.

Acknowledgements We would like to express our sincere gratitude to all the individuals and organizations that have contributed to the publication of this research paper.

References

[1] Krishna, C.M., Shin, K.G.: Real Time Systems. McGrawHill, 1 (1997)

[2] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM **20**(1), 40–61 (1973)

[3] Fisher, N., Baruah, S., Baker, T.P.: The partitioned scheduling of sporadic tasks according to static-priorities. In: ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems, pp. 118–127. IEEE Computer Society, Washington, DC, USA (2006). https://doi.org/10.1109/ECRTS.2006.30

[4] Bini, E., Buttazzo, G.C., Buttazzo, G.: Rate monotonic analysis: the hyperbolic bound. IEEE Transactions on Computers **52**(7), 933–942 (2003)

[5] Amdahl, G.M.: Validity of the single processor approach to achieving large-scale computing capabilities. In: AFIPS Conference Proceedings (30), pp. 483–485 (1967)

[6] Liu, J.W.S.: Real Time Systems. Prentice Hall, 2000

[7] Buttazzo, G.C.: Rate monotonic vs. edf: Judgment day. Real-Time Systems **29**(1), 5–26 (2005)

[8] Bini, E., C.Buttazzo, G.: Schedulability analysis of periodic fixed priority systems. IEEE Transactions on Computers **53**(11), 1462–1473 (2004)

[9] Laplante, P.A.: REAL-TIME SYSTEMS DESIGN AND ANALYSIS, 2004

[10] Burns, A., Wellings, A.: Real-time Systems and Programming Languages. Ada 95, Real-Time Java and Real-time POSIX, 3rd edn. Addison Wesley Longmain, 2001

[11] Audsley, N.C., Burns, A., M.Richardson, Wellings, A.: Applyingnew scheduling theory to static priority preemptive scheduling. Software Engineering Journal **8**(5), 284–292 (1993)

[12] Joseph, M., Pandya, P.: Finding response times in a real-time system. The Computer Journal **29**(5), 390–395 (1986)

[13] Min-Allah, N., Ali, I., Xing, J., Wang, Y.: Utilization bound for periodic task set with composite deadline. Journal of Computers and Electrical Engineering, Accepted (2010)

[14] Manabe, Y., Aoyagi, S.: A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. Real-Time Systems **14**(2), 171–181 (1998)

[15] Sjodin, M., Hansson, H.: Improved response-time analysis calculations. Proceedings of the 19th IEEE Real-Time Systems Symposium, 399–409 (1998)

[16] Orozco, J., Cayssials, R., Santos, J., Santos, R.: On the Minimum Number of Priority Levels Required for the Rate Monotonic Scheduling of Real-time Systems. In: 10th Euromicro Workshop on Real Time System (1998)

[17] Lehoczky, J.P., Sha, L., Strosnider, J., Tokuda, H.: Fixed Priority Scheduling Theory for Hard Real-time Systems. In: van Tilborg, A.M.,

Koob, G.M. (eds.) Foundations of Real-Time Computing Scheduling and Resource Management, pp. 1–30. Kluwer. 1991