International Journal of Scientific Research in Computer Science, Engineering and Information Technology

ISSN : 2456-3307

Available Online at : www.ijsrcseit.com
doi : https://doi.org/10.32628/IJSRCSEIT

# Enhancing Capability of Gang scheduling by integration of Multi Core Processors and Cache

**Dr. Sangeeta**[1], **Kavita**[2]
[1]Assistant Professor, Department of CSE, Chaudhary Devilal University, Sirsa, Haryana, India
[2]M.Tech. Scholar, Department of CSE, Chaudhary Devilal University, Sirsa, Haryana, India

ARTICLEINFO

**Article History:**

Accepted: 01 July 2023
Published: 10 July 2023

**Publication Issue**
Volume 9, Issue 4
July-August-2023

**Page Number**
77-83

## ABSTRACT

In computer architecture, multithreading is ability of a central processing unit (CPU) or a single core within a multi-core processor to execute multiple processes or threads concurrently, appropriately supported by operating system. This approach differs from multiprocessing, as with multithreading processes & threads have to share resources of a single or multiple cores: computing units, CPU caches, & translation lookaside buffer (TLB). Multiprocessing systems include multiple complete processing units, multithreading aims to increase utilization of a single core by using thread-level as well as instruction-level parallelism. Objective of research is increase efficiency of scheduling dependent task using enhanced multithreading. gang scheduling of parallel implicit-deadline periodic task systems upon identical multiprocessor platforms is considered. In this scheduling problem, parallel tasks use several processors simultaneously. first algorithm is based on linear programming & is first one to be proved optimal for considered gang scheduling problem. Furthermore, it runs in polynomial time for a fixed number m of processors & an efficient implementation is fully detailed. Second algorithm is an approximation algorithm based on a fixed-priority rule that is competitive under resource augmentation analysis in order to compute an optimal schedule pattern. Precisely, its speedup factor is bounded by (2–1/m). Both algorithms are also evaluated through intensive numerical experiments. In our research we have enhanced capability of Gang Scheduling by integration of multi core processor & Cache & make simulation of performance in MATLAB.

**Keywords** : TLP, Response Time, Latency, throughput, multithreading, Scheduling

## I. INTRODUCTION

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since late 1990s. This allowed concept of throughput computing to re-emerge from more specialized field of transaction processing; even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve throughput of all tasks result within overall performance gains.

**Types of multithreading**

**Block multithreading**

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, that might take hundreds of CPU cycles for data to return. Instead of waiting for stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when data for previous thread had arrived, would previous thread be placed back on list of ready-to-run threads.

For example:

1. Cycle *i*: instruction *j* from thread *A* is issued.
2. Cycle $i + 1$: instruction $j + 1$ from thread *A* is issued.
3. Cycle $i + 2$: instruction $j + 2$ from thread *A* is issued, that is a load instruction that misses within all caches.
4. Cycle $i + 3$: thread scheduler invoked, switches to thread *B*.
5. Cycle $i + 4$: instruction *k* from thread *B* is issued.
6. Cycle $i + 5$: instruction $k + 1$ from thread *B* is issued.

Conceptually, it is similar to cooperative multi-tasking used within real-time operating systems, within which tasks voluntarily give up execution time when they need to wait upon some type of event. This type of multithreading is known as block, cooperative or coarse-grained multithreading. The goal of multithreading hardware support is to allow quick switching between a blocked thread & another thread ready to run. To achieve this goal, hardware cost is to replicate program visible registers, as well as some processor control registers. Switching from one thread to another thread means hardware switches from using one register set to another; to switch efficiently between active threads, each active thread needs to have its own register set. For example, to quickly switch between two threads, register hardware needs to be instantiated twice. Additional hardware support for multithreading allows thread switching to be done within one CPU cycle, bringing performance improvements. Also, additional hardware allows each thread to behave as if it were executing alone & not sharing any hardware resources with other threads, minimizing amount of software changes needed within application & operating system to support multithreading.

Many families of microcontrollers & embedded processors have multiple register banks to allow quick context switching for interrupts. Such schemes could be considered a type of block multithreading among user program thread & interrupt threads.

**Interleaved multithreading**

The purpose of interleaved multithreading is to remove all data dependency stalls from execution pipeline. Since one thread is relatively independent from other threads, there is less chance of one instruction within one pipelining stage needing an output from an older instruction within pipeline. Conceptually, it is similar to preemptive multitasking used within operating systems; an analogy would be that time slice given to each active thread is one CPU cycle.

For example:

1. Cycle $i + 1$: an instruction from thread *B* is issued.
2. Cycle $i + 2$: an instruction from thread *C* is issued.

This type of multithreading was first called barrel processing, within which staves of a barrel represent pipeline stages & their executing threads. Interleaved, preemptive, fine-grained or time-sliced multithreading are more modern terminology.

In addition to hardware costs discussed within block type of multithreading, interleaved multithreading has an additional cost of each pipeline stage tracking thread ID of instruction it is processing. Also, since there are more threads being executed concurrently within pipeline, shared resources such as caches & TLBs need to be larger to avoid thrashing between different threads.

**Simultaneous multithreading**

The most advanced type of multithreading applies to superscalar processors. Whereas a normal superscalar processor issues multiple instructions from a single thread every CPU cycle, within simultaneous multithreading (SMT) a superscalar processor could issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction-level parallelism, this type of multithreading tries to exploit parallelism available across multiple threads to decrease waste associated with unused issue slots.

For example:

1. Cycle *i*: instructions *j* & *j* + 1 from thread *A* & instruction *k* from thread *B* are simultaneously issued.
2. Cycle *i* + 1: instruction *j* + 2 from thread *A*, instruction *k* + 1 from thread *B*, & instruction *m* from thread *C* are all simultaneously issued.
3. Cycle *i* + 2: instruction *j* + 3 from thread *A* & instructions *m* + 1 & *m* + 2 from thread *C* are all simultaneously issued.

To distinguish other types of multithreading from SMT, term "temporal multithreading" is used to denote when instructions from only one thread could be issued at a time.

In addition to hardware costs discussed for interleaved multithreading, SMT has additional cost of each pipeline stage tracking thread ID of each instruction being processed. Again, shared resources such as caches & TLBs have to be sized for large number of active threads being processed.

Implementations include DEC (later Compaq) EV8 (not completed), Intel Hyper-Threading, IBM POWER5, Sun Microsystems UltraSPARC T2, MIPS MT, & CRAY XMT.

## II. LITERATURE REVIEW

***Yeh-Ching Chung wrote on* "Applications & Performance Analysis of A Compile-Time Optimization Approach for List Scheduling Algorithms on Distributed Memory Multiprocessors "**

They have proposedacompile-time optimization approach, *bottom-up top-down duplication heuristic* (BTDH), for static scheduling of *directed+cyclic graphs* (DAGS) on *distributed memory multiprocessors* (DMMs). In this paper, they discuss applications of BTDH for *list scheddhg algorithms* (LSAs). There are two ways to use BTDH for LSAs.BTDHcan be used with aLSAto form a new *scheduling* algorithm (LSA/BTDH). It could be usedas apure *opti*mization algorithm for a LSA (LSA-BTDH)..

**Ishfaq Ahmad1 & Yu-Kwong Kwok2 wrote on "On Parallelizing Multiprocessor Scheduling Problem"**

Existing heuristics for scheduling a node & edge weighted directed task graph to multiple processors could produce satisfactory solutions but incur high time complexities that tend to exacerbate within more realistic environments with relaxed assumptions. Consequently, these heuristics do not scale well & cannot handle problems of moderate sizes. The algorithm also exhibits an interesting trade-off between solution quality & speedup while scaling well with problem size.

**Maruf Ahmed, Sharif M. H. Chowdhury wrote on List Heuristic Scheduling Algorithms for Distributed Memory Systems with Improved Time Complexity**

They present a compile time list heuristic scheduling algorithm called *Low Cost Critical Path algorithm (LCCP)* for distributed memory systems. LCCP has low scheduling cost for both homogeneous & heterogeneous systems. In some recent papers list heuristic scheduling algorithms keep their scheduling cost low by using a fixed size heap & a FIFO, where heap always keeps fixed number of tasks & excess tasks are inserted within FIFO. When heap has empty spaces, tasks are inserted within it from FIFO. Best known list scheduling algorithm based on this strategy requires two heap restoration operations, one after extraction & another after insertion. Our LCCP algorithm improves on this by using only one such operation for both extraction & insertion, that within theory reduces scheduling cost without compromising scheduling performance. In our experiment they compare LCCP with other well known list scheduling algorithms & it shows that LCCP is fastest among all.

**Wayne F. Boyer wrote on "Non-evolutionary algorithm for scheduling dependent tasks within distributed heterogeneous computing environments"**
The Problem of obtaining an optimal matching & scheduling of interdependent tasks within distributed heterogeneous computing (DHC) environments is well known to be an NP-hard problem. In a DHC system, task execution time is dependent on machine to which it is assigned & task precedence constraints are represented by a directed acyclic graph. Recent research within evolutionary techniques has shown that genetic algorithms usually obtain more efficient schedules that other known algorithms.
We propose a non-evolutionary random scheduling (RS) algorithm for efficient matching & scheduling of inter-dependent tasks within a DHC system. RS is a succession of randomized task orderings & a heuristic mapping from task order to schedule. Randomized task ordering is effectively a topological sort where outcome may be any possible task order for which task precedent constraints are maintained. A detailed comparison to existing evolutionary techniques (GA & PSGA) shows proposed algorithm is less complex than evolutionary techniques, computes schedules within less time, requires less memory & fewer tuning parameters. Simulation results show that average schedules produced by RS are approximately as efficient as PSGA schedules for all cases studied & clearly more efficient than PSGA for certain cases.

## III. RESEARCH METHODOLOGY

In computing, **scheduling** is method by which work specified by some means is assigned to resources that complete work. The work may be virtual computation elements such as threads, processes or data flows, that are within turn scheduled onto hardware resources such as processors, network links or expansion cards.
A scheduler is what carries out scheduling activity. Schedulers are often implemented so they keep all computer resources busy (as within load balancing), allow multiple users to share system resources effectively, or to achieve a target quality of service. Scheduling is fundamental to computation itself, & an intrinsic part of execution model of a computer system; concept of scheduling makes it possible to have computer multitasking with a single central processing unit (CPU).
A scheduler may aim at one of several goals, for example, maximizing *throughput* (total amount of work completed per time unit), minimizing *response time* (time from work becoming enabled until first point it begins execution on resources), or minimizing *latency* (the time between work becoming enabled & its subsequent completion), maximizing *fairness* (equal CPU time to each process, or more generally appropriate times according to priority & workload of each process). In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler would implement a suitable compromise. Preference is given to any one of concerns mentioned above, depending upon user's needs & objectives.

| Operating System | Preemption | Algorithm |
|---|---|---|

| | | |
|---|---|---|
| **Amiga OS** | Yes | Prioritized round-robin scheduling |
| **FreeBSD** | Yes | Multilevel feedback queue |
| **Linux kernel before 2.6.0** | Yes | Multilevel feedback queue |
| **Linux kernel 2.6.0–2.6.23** | Yes | O(1) scheduler |
| **Linux kernel after 2.6.23** | Yes | Completely Fair Scheduler |
| **Mac OS pre-9** | None | Cooperative scheduler |
| **Mac OS 9** | Some | Preemptive scheduler for MP tasks, & cooperative for processes & threads |
| **Mac OS X** | Yes | Multilevel feedback queue |
| **NetBSD** | Yes | Multilevel feedback queue |
| **Solaris** | Yes | Multilevel feedback queue |
| **Windows 3.1x** | None | Cooperative scheduler |
| **Windows 95, 98, Me** | Half | Preemptive scheduler for 32-bit processes, & cooperative for 16-bit processes |
| **Windows NT (including 2000, XP, Vista, 7, & Server)** | Yes | Multilevel feedback queue |

Table 1 List of algorithms

## IV. CHALLENGES WITHIN RESEARCH

Multiple threads could interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved but could be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement with its HyperThreading technology,[1] while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run within parallel. On other hand, hand-tuned assembly language programs using MMX or Altivec extensions & performing data pre-fetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading & could indeed see degraded performance due to contention for shared resources.

From software standpoint, hardware support for multithreading is more visible to software, requiring more changes to both application programs & operating systems than multiprocessing. Hardware techniques used to support multithreading often parallel software techniques used for computer multitasking of computer programs. Thread scheduling is also a major problem within multithreading.

## V. PARALLEL COMPUTING

**Parallel computing** is a type of computation in which many calculations are carried out simultaneously, operating on principle that large problems could often be divided into smaller ones, which are then solved at same time. There are several different forms of parallel computing: bit-level, instruction-level, data, & task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years, parallel computing has become dominant paradigm in computer architecture, mainly in form of multi-core processors. Parallel computing is closely related to concurrent computing—they are frequently used together, & often conflated, though two are distinct: it is possible to have parallelism without concurrency & concurrency without parallelism. In parallel computing, a computational task is typically broken down in several, often many, very similar subtasks that could be processed independently & whose results are combined afterwards, upon completion. In contrast, in concurrent computing, various processes often do not address related tasks; when they do, as is typical in distributed computing, separate tasks may have a varied nature & often require some inter-process communication during execution.

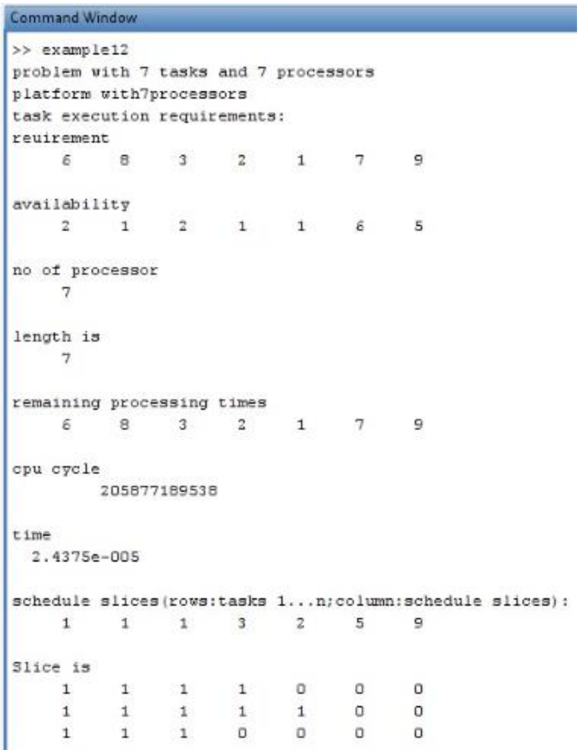
## VI. GANG TASK SCHEDULING

```
input :
        $n$ jobs $J_j(u_j, v_j), 1 \le j \le n$ ;
        $m$: number of processors;
output:
        Slice lengths $S(s), s = 1, 2, \ldots$;
        Scheduled jobs $j$ in slice $s$: Sched($s$,$j$) $\in \{0,1\}, 1 \le j \le n$ ;
List=Sort($J_1, \ldots, J_n$) ; /* Jobs are sorted in non increasing order of $v_j$ */
$s = 0$ ;                                                  /* Number of slices */
$r_j := u_j \quad \forall j = 1 \cdots n$ ;                /* Job remaining execution times $r_j$ */
while $\exists j, r_j > 0, 1 \le j \le n$ do
    /* create a new slice                                                    */
    $s = s + 1$ ;                                          /* Number of slices */
    $K = m$ ;                                          /* Remaining processors */
    $\ell = \infty$ ;                                  /* Slice length upper bound */
    Sched($s$,$j$)=0   $1 \le j \le n$ ;                       /* Empty Slice */
    foreach $j \in List$ do
        /* For each job $j$ in priority List                                 */
        if $v_j \le K$ then
            /* the job $j$ is schedulable in current slice                   */
            Sched($s$,$j$)=1;
            $\ell = \min(\ell, r_j)$;                      /* update slice length */
            $K = K - v_j$ ;                              /* remaining processors */
        end
    end
    S(s)=l;                                                    /* Slice length */
    for $j = 1 \ldots n$ do
        /* Update remaining execution times                                   */
        if $Sched(s, j)$ then
            $r_j = r_j - \ell$;
        end
    end
end
```

## VII. RESULT AND DISCUSSION

**Single processor output**

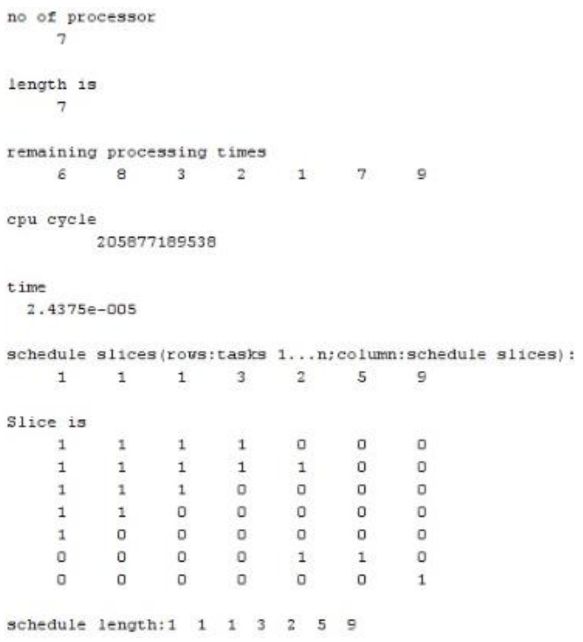
**Step 1**



```
Command Window
>> example12
problem with 7 tasks and 7 processors
platform with7processors
task execution requirements:
reuirement
     6     8     3     2     1     7     9

availability
     2     1     2     1     1     6     5

no of processor
     7

length is
     7

remaining processing times
     6     8     3     2     1     7     9

cpu cycle
        205877189538

time
  2.4375e-005

schedule slices(rows:tasks 1...n;column:schedule slices):
     1     1     1     3     2     5     9

Slice is
     1     1     1     1     0     0     0
     1     1     1     1     1     0     0
     1     1     1     0     0     0     0
```

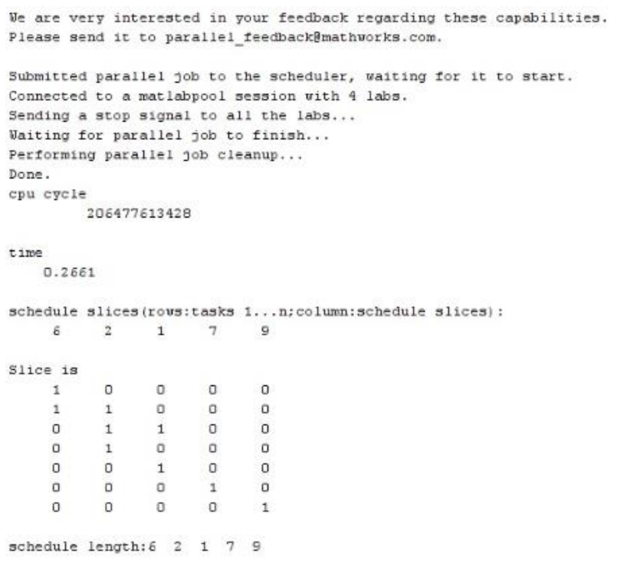
**Fig1 Task Execution requirement**

**Step 2**



```
no of processor
     7

length is
     7

remaining processing times
     6     8     3     2     1     7     9

cpu cycle
        205877189538

time
  2.4375e-005

schedule slices(rows:tasks 1...n;column:schedule slices):
     1     1     1     3     2     5     9

Slice is
     1     1     1     1     0     0     0
     1     1     1     1     1     0     0
     1     1     1     0     0     0     0
     1     1     0     0     0     0     0
     1     0     0     0     0     0     0
     0     0     0     0     1     1     0
     0     0     0     0     0     0     1


schedule length:1  1  1  3  2  5  9
```

**Fig 2. Finding schedule lenght**



```
We are very interested in your feedback regarding these capabilities.
Please send it to parallel_feedback@mathworks.com.

Submitted parallel job to the scheduler, waiting for it to start.
Connected to a matlabpool session with 4 labs.
Sending a stop signal to all the labs...
Waiting for parallel job to finish...
Performing parallel job cleanup...
Done.
cpu cycle
        206477613428

time
    0.2661

schedule slices(rows:tasks 1...n;column:schedule slices):
     6     2     1     7     9

Slice is
     1     0     0     0     0
     1     1     0     0     0
     0     1     1     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1

schedule length:6  2  1  7  9
```

## VIII. SCOPE OF RESEARCH

If a thread gets a lot of cache misses, other threads could continue taking advantage of unused computing resources, that may lead to faster overall execution as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all computing resources of CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle. If several threads work on same set of data, they could actually share their cache, leading to better cache usage or synchronization on its values.

## IX. REFERENCES

[1]. Remzi H. Arpaci-Dusseau; Andrea C. Arpaci-Dusseau (January 4, 2021). "Chapter 7: Scheduling: Introduction, Section 7.6: A New Metric: Response Time". Operating Systems: Three Easy Pieces (PDF). p. 6. Retrieved February 2, 2015.

[2]. Paul Krzyzanowski "Process Scheduling: Who gets to run next?". cs.rutgers.edu. Retrieved 2021

[3]. Abraham Silberschatz, Peter Baer Galvin & Greg Gagne (2021). Operating System Concepts 9. John Wiley & Sons,Inc. ISBN 978-1-118-06333-0.

[4]. Here is C-code for FCFS

[5]. Early Windows at Wayback Machine

[6]. Sriram Krishnan. "A Tale of Two Schedulers Windows NT & Windows CE".

[7]. Inside Windows Vista Kernel: Part 1, Microsoft Technet

[8]. "Vista Kernel Improvements".

[9]. "Technical Note TN2028 - Threading Architectures".

[10]. "Mach Scheduling & Thread Interfaces".

[11]. http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6

[12]. Molnár, Ingo (2020). "[patch] Modular Scheduler Core & Completely Fair Scheduler [CFS]". linux-kernel (Mailing list).

**Cite this article as :**

Dr. Sangeeta, Kavita, "Enhancing Capability of Gang scheduling by integration of Multi Core Processors and Cache", International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT), ISSN : 2456-3307, Volume 9, Issue 4, pp.77-83, July-August-2023.