Enhancing Capability of Gang Scheduling by Integration of Multi Core Processors & Cache

Authors
Sawati1, Nikita2

1M.tech-Department of Computer Engineering Indus Institute of Engineering & Technology Jind kurukshetra University
2Assistant Professor & HOD-- Computer Engineering Indus Institute of Engineering & Technology Jind kurukshetra University

Email- jain008swati@gmail.com, nikitasagar@gmail.com

ABSTRACT:
Advantage of space sharing is elimination of multiprogramming, which eliminates context switching overhead. However, an equally clear disadvantage is time wasted when a CPU blocks & has nothing at all to do until it becomes ready again. Consequently, people had looked for algorithms that attempt to schedule within in both time & space together, specially for processes that is create multiple threads, which usually need to communicate within one another. To see kind of problem that could occur when threads of a process (or processes of a job) are independently scheduled, In our research we had enhanced capability of Gang Scheduling by integration of multi core processor & Cache & make simulation of performance in MATLAB. The trick that makes gang scheduling work is that all CPUs are scheduled synchronously. This means that time is divided into discrete quanta. At start of each new quantum, all CPUs are rescheduled, within a new thread being started on each one.

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

1. INTRODUCTION
On a uniprocessor, scheduling is one dimensional. On a multiprocessor, scheduling is two dimensional. The scheduler had to decide which process to run & which CPU to run it on. This extra edges greatly complicates scheduling on multiprocessors.

Another complicating factor is that within in some systems, all processes are unrelated whereas within in others they come within in groups. An example of former situation is a timesharing system which includes independent users start up free processes. The processes are unrelated & each one could be scheduled without regard to other ones.

An example of latter situation occurs regularly within in program development environments. Large systems seldom consist of few number of header files containing macros, type definitions, & variable declarations that are used by actual code files. When a header file is changed, all code files that include it must be recompiled. The program make is frequently used to manage development. When make is invoked, it starts compilation of only those code files that must be recompiled on basis of changes to header or code files. Object files that are still valid are not revived.

The original version of make did its work continuely, but newer versions designed for multiprocessors could start up all compilations at once. If 10 compilations are want, it does not make sense to schedule 9 of them quickly & leave last one until much later since user will not perceive work as completed until last one finishes. In it case it makes sense to regard processes as a group & to take that into account when scheduling them.
2. TIMESHARING

Let us foremost address case of scheduling free processes; later we would consider how to schedule related processes. Simplest scheduling algorithm for dealing within unrelated processes is to had a single system wide data structure for active processes, possibly just a list. Here 16 CPUs are all currently busy, & a prioritized set of fourteen processes are waiting to run. The first CPU to finish its ongoing work (or had its process block) is CPU 4, which then locks scheduling queues & selects highest priority process, A. Next, CPU 12 goes idle & chooses process B. As long as processes are completely unrelated, doing scheduling it way is a reasonable choice.

3. GANG SCHEDULING

A clear advantage of space sharing is elimination of multiprogramming, which eliminates context switching overhead. However, an equally clear disadvantage is time wasted when a CPU blocks & had nothing at all to do until it becomes ready again. Consequently, people had looked for algorithms that attempt to schedule within in both time & space together, important for processes that create multiple threads, which usually need to communicate within one another.

To see kind of problem that could occur when threads of a process (or processes of a job) are free scheduled, consider a system within threads $A_0$ & $A_1$ belonging to process A & threads $B_0$ & $B_1$ belonging to process B. threads $A_0$ & $B_0$ are timeshared on CPU 0; threads $A_1$ & $B_1$ are timeshared on CPU 1. threads $A_0$ & $A_1$ need to communicate often. The communication pattern is that $A_0$ sends $A_1$ a message, within $A_1$ then sending back a reply to $A_0$, followed by other such sequence. Suppose that luck has it that $A_0$ & $B_1$ start first.

Communication between two threads to process A that are running out of phase. In time slice 0, $A_0$ sends $A_1$ a request, but $A_1$ does not get it until it runs within in time slice 1 starting at 100 msec. It sends reply immediately, but $A_0$ does not get reply until it runs again at 200 msec. The net result is one request-reply sequence each 200 msec. Not very good.

The solution to it problem is gang scheduling, which is an outgrowth of co-scheduling (Ousterhout, 1982). Gang scheduling has three parts:

- Groups of related threads are scheduled as a unit, a gang.
- All members of a gang run simultaneously, on different timeshared CPUs.
- All gang members start & end their time slices together.

The trick that makes gang scheduling work is that all CPUs are scheduled synchronously. This means that time is divided into discrete quanta. At start of each new quantum, all CPUs are rescheduled, within a new thread being started on each one. At start of following quantum, another scheduling event happens. In between, no scheduling is done. If a thread blocks, its CPU stays idle until end of quantum.

An example of how gang scheduling works where we had a multiprocessor within six CPUs being used by five processes, A through E, within a total of 24 ready threads. During time slot 0, threads $A_0$ through $A_6$ are scheduled & run. During time slot 1, Threads $B_0$, $B_1$, $B_2$, $C_0$, $C_1$, & $C_2$ are scheduled & run. During time slot 2, $D$'s five threads & $E_0$ get to run. The remaining six threads belonging to process $E$ run within in time slot 3. Then cycle repeats, within slot 4 being same as slot 0 & so on.

Gang scheduling is a scheduling algorithm for parallel systems that schedules related threads or processes to run simultaneously on different processors. Usually these will be threads all belonging to same process, but they might also be from different processes, for example when processes had a producer-consumer relationship, or when they all come from same MPI program.

Gang scheduling is used so that if two or more threads or processes communicate within each other, they will all be ready to communicate at same time. If they were not gang-scheduled, then one could wait to send or receive a message to another while it is sleeping, & vice versa. When processors are over-subscribed &
gang scheduling is not used within a group of processes or threads which communicate within each other, it could lead to situations where each communication event suffers overhead of a context switch. Technically, gang scheduling is based on a data structure called Ousterhout matrix. In it matrix each row represents a time slice, & each column a processor. The threads or processes of each job are packed into a single row of matrix. During execution, coordinated context switching is performed across all nodes to switch from processes within in one row to those within in next row.

Gang scheduling is stricter than coscheduling. It requires all threads of same process to run concurrently, while coscheduling allows for fragments, which are sets of threads that do not run concurrently within rest of gang.

4. CHALLENGES WITHIN RESEARCH

Multiple threads could interfere within 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 within 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.

5. 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 inform of multi-core processors. Parallel computing is closely related to concurrent computing—they are frequently used together, & often conflated, thought two are distinct: it is possible to had 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 might had a varied nature & often require some inter-process communication during execution.
6. GANG TASK SCHEDULING

```matlab
input:
- n jobs: j(i1, jn), 1 <= j <= n;
- m: number of processors.
output:
- slice length: S(i), 1 <= i <= n;
- scheduled jobs: j(i) in slice: S(i), 1 <= i <= n;

L=sort(L); /* Jobs are sorted in non increasing order of vi */
v=0; /* Number of slices */
t=0; /* Job remaining execution times */
while v > 0 do // WHILE v > 0 DO
    t = u(i); /* Remaining processors */
    K = n; /* Slice length upper bound */
    S(i) = []; /* Empty slice */
    for j = 1 to n do // FOR j = 1 TO n DO
        /* Update remaining execution times */
        if S(i) == [] then
            S(i) = t;
        else
            S(i) = [S(i), t];
        end
    end
    /* For each job j in priority list */
    if v <= K then /* if V <= K then */
        if j is schedulable in current slice // if j is schedulable in current slice
            S(j) = [S(j), ]; /* Update slice length */
        end
        K = K - v; /* remaining processors */
    end
end
```

7. RESULT AND DISCUSSION

Single processor output

disp('problem within 7 tasks and 7 processors');
u=[6 8 3 2 1 7 9];
p=[2 1 2 1 1 6 5];
m=7;
disp(strcat('platform within ',num2str(m),',processors'));
disp('task execution requirements:');
[Slice X]=ghp12(u,p,m);
disp('schedule slices(rows:tasks 1...n;column:schedule slices):');
disp(X);
disp('Slice is');
disp(Slice);
disp(strcat('schedule length:',num2str((X))));
ganghueistic function
function [Slice X]= ghp(u,p,m)
disp('requirement');
disp(u);
disp('availability');
disp(p);
disp('no of processor');
disp(m);
c=u;
n=length(u);
disp('length is');
disp(n);
r=c;
disp('remaining processing times');
disp(r);
Slice=[];
X=[];
SchedCompleted=false;
tc=0;
etm=0;
walltime=zeros(1:4);
k=1;
maxslice=max(r);

matlabpool open
while(~SchedCompleted)
    used=0;
sched=zeros(1,n);
X(k)=maxslice;
for i=1:n
tic
    for j=1:4
        tc=tc+maxNumCompThreads(2^(j-1));
        if(r(i)>0 && used+p(i)<=m)
            sched(i)=1;
            used=used+p(i);
            X(k)=min(X(k),r(i));
        end
    end
    walltime(i)=toc;
etm=elm+walltime(i);
end
Slice(:,end+1)=sched;
r=r-(X(k).*sched);
if(r==0)
    SchedCompleted=true;
else
    k=k+1;
end
end
matlabpool close
disp('cpu cycle');
disp(tic);
disp('time');
disp(elm);
end

Code for Multithreaded based processor
disp('problem within 7 tasks and 7 processors');
u=[6 8 3 2 1 7 9];
p=[2 1 2 1 6 5];
m=7;
disp(strcat('platform within ',num2str(m),'processors'));
disp('task execution requirements:');
[Slice X]=ghp(u,p,m);
disp('schedule slices(rows:tasks 1...n;column:schedule slices):');
disp(X);
disp('Slice is');
disp(Slice);
disp(strcat('schedule length:',num2str((X))));
modified ganghueristic
function [Slice X]= ghp(u,p,m)
disp('reuirement');
disp(u);
disp('availability');
disp(p);
disp('no of processor');
disp(m);
c=u;
n=length(u);
disp('length is');
disp(n);
r=c;
disp('remaining processing times');
disp(r);
Slice=[];
X=[];
SchedCompleted=false;
tc=0;
etm=0;
walltime=zeros(1:4);
k=1;
maxslice=max(r);

matlabpool open
while(~SchedCompleted)
    used=0;
sched=zeros(1,n);
X(k)=maxslice;
for i=1:n
    tic
    for j=1:4
        tc=tc+maxNumCompThreads(2^(j-1));
        if(r(i)>0 && used+p(i)<=m)
            sched(i)=1;
            used=used+p(i);
        end
    end
end
SchedCompleted=true;
end

disp('schedule slices(rows:tasks 1...n;column:schedule slices):');
disp(X);
disp('Slice is');
disp(Slice);
disp(strcat('schedule length:',num2str((X))));
X(k)=min(X(k),r(i));
end
walltime(i)=toc;
eltm=eltm+walltime(i);
end
end
Slice(:,end+1)=sched;
r=r-(X(k).*sched);
if(r==0)
    SchedCompleted=true;
else
    k=k+1;
end
end
matlabpool close
disp('cpu cycle');
disp(tic);
disp('time');
disp(eltm);
end
Step 1

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example2}
\caption{Task Execution requirement}
\end{figure}
Step 2

<table>
<thead>
<tr>
<th>no of processor</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>length is</td>
<td>7</td>
</tr>
<tr>
<td>remaining processing times</td>
<td>6 8 3 1 7 9</td>
</tr>
<tr>
<td>cpu cycle</td>
<td>205577165538</td>
</tr>
<tr>
<td>time</td>
<td>2.4375e-005</td>
</tr>
<tr>
<td>schedule slices</td>
<td>1 1 1 1 3 2 5 9</td>
</tr>
<tr>
<td>Slice is</td>
<td>1 1 1 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 1 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 1 1 0</td>
</tr>
<tr>
<td></td>
<td>0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>schedule length</td>
<td>1 1 3 2 5 9</td>
</tr>
</tbody>
</table>

Fig 2. Finding schedule length

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

Submitted parallel job to the scheduler, waiting for it to start.
Connected to a markRNpool session with 4 tasks.
Sending a stop signal to all the tasks...
Waiting for parallel job to finish...
Performing parallel job cleanup...

| cpu cycle       | 10056213428 |
| time            | 0.086 |
| schedule slices | 4 2 1 7 9 |
| Slice is        | 1 0 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 | 2 1 7 9 |

7. SCOPE OF RESEARCH

If a thread gets a lot of cache misses, other threads could continue taking advantage of unused computing resources, that might lead to faster overall execution as these resources would had 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 might 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.

REFERENCE

4. Here is C-code for FCFS
5. Early Windows at Wayback Machine
7. Inside Windows Vista Kernel: Part 1, Microsoft Technet
8. "Vista Kernel Improvements".
10. "Mach Scheduling & Thread Interfaces".