Distributed [*Computing*] Systems

Modeling Resource Management in Concurrent Computing Systems

Isaac D. Scherson (aka The Schark ☺)

Dept. of Computer Science (Systems)
Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, CA 92697-3425

isaac@ics.uci.edu
www.ics.uci.edu/~isaac www.ics.uci.edu/~schark

CompSci-230, Winter 2019

Modeling Resource Management in Concurrent Computing Systems

Acknowledgements

This work was started in 1994-95 and the following were the pioneers to whom we owe the ideas presented here.

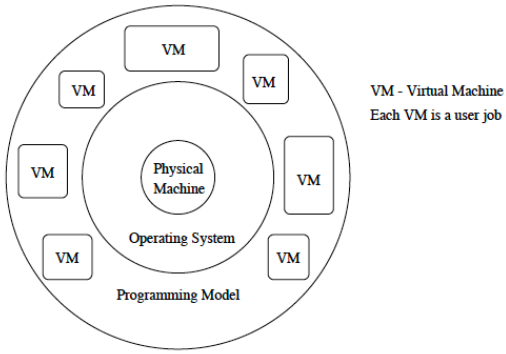
- Piotr Chrząstowski-Wachtel - Institute of Informatics, Warsaw University
- Dinesh Ramanathan - Somewhere in the Silicon Valley
- Raghu Subramanian - Somewhere in the Silicon Valley

This work was supported in part through various grants from NASA, the NSF and the AFOSR

Warning !

This is a thought-provoking presentation. The ideas are very simple and are presented to motivate a little formal thinking about the scheduling problem in Concurrent Computing Systems.

Concurrent Computing OS Model



General Purpose Parallel Operating System

The Programming Model

- There seems to be a convergence on the data parallel programming in the form of HPF and Fortran 90.
- In a survey of 120 parallel algorithms from three ACM Symposia on Theory of Computing (STOC), all were found to be data parallel.
- This preponderance of the data parallel programming model is perhaps because it allows one to express a large degree of parallelism, while retaining the single-thread-of-control philosophy of sequential programming.

The Machine Model

- A natural way of executing a data parallel program is on a SIMD machine.
- BUT . . . data parallelism does not equal SIMD.
- A data parallel program may be executed on an asynchronous MIMD machine:
  - Chief advantage: It is possible to run several jobs on a MIMD machine simultaneously.
  - A MIMD machine does not force unnecessary synchronization after every instruction, or unnecessary sequentialization of non-interfering branches, as a SIMD machine does.

SPMD Programs

- The combination of the data parallel programming model and a MIMD machine model is called a SPMD execution model
- SPMD stands for Single Program Multiple Data
  - All processors execute the same program, but may be at different stages of the program at a given time
- For the remainder of this presentation, our discussion is predicated on the SPMD execution model.

Virtual Processors

- A program is expressed in a language that describes a virtual machine.
- For a data parallel program, the virtual machine consists of a (typically large) number of identical *virtual processors* (VPs), communicating through an interconnection network.
  - For instance, the standard data parallel program to multiply two $N \times N$ matrices may be viewed as a virtual machine consisting of N^2 VPs communicating in a mesh.

Virtual Processors (Cont'd)

- For many years now, the concept of virtual processors has been relegated to the status of a mere logical aid to programmers
  - In our view, the notion of virtual processors should form the fundamental basis for the definition of a Concurrent Computer System and for the resource management strategies embedded in the OS.

Redefining Massive Parallelism

- A Programmer sees only a VM with as many Virtual Processors as s/he needs.
- Solutions are cast in the space of the problem and not on that of the physical machine.
- The measure of Massive Parallelism is through the number of VPs the physical machine is capable of emulating per unit of time.
- HENCE . . . It does not matter what the physical machine looks like, provided it can emulate Concurrent Computing Virtual Machines.

Resource Management Operations

- **Spatial scheduling:** A space sharing policy that defines which physical processor each VP is allocated to.
- **Temporal scheduling:** A time sharing policy that dictates how each physical processor switches between the execution of the VPs allocated to it.
- **Load balancing:** Static and/or Dynamic redistribution of VPs among physical processors responding to some predetermined objective function.

Resource Management Operations

- **Memory and I/O problems:** Memory limitations and I/O bottlenecks may also be phrased in terms of VPs. Memory limitations occur when a processor does not have enough local memory to hold all the VPs allocated to it. I/O bottlenecks are most pronounced while loading (roll-in) the VPs of a job from the disk into the local memories of various processors.

AND NOW ...

A game you can play in your free time . . . . . .

Model of the System

- Let $\Pi = \{\pi_1, \pi_2, \ldots, \pi_n\}$. denote the set of physical processors or processing elements (PEs).
- Interconnection network details are ignored, only assuming that it *guarantees* the completion of all communications within a reasonable time bound (not necessarily a constant, as in the PRAM model).
- Let $\mathcal{J} = \{J_1, J_2, \ldots, J_m\}$ denote the set of jobs in the system at some time.
- For notational convenience, job J_i is identified with its set of VPs $\{P_{i1}, P_{i2}, \ldots, P_{i|J_i|}\}$.

Handling Time

- Break time into **time slices**: discrete intervals of equal length.
- The time slice is assumed to be machine dependent and constant.
- Within each time slice, every π_i runs some VP chosen from the VPs allocated to it, or it idles.

Handling Time (Cont'd)

- A PE π_i runs a VP of a Job until:
  - The VP issues a communication request which cannot be completed (data to be read is not ready), or
  - The time slice expires.
- The communication instruction is considered an indivisible instruction and time slices are extended accordingly.

Execution of Programs

- *For simplicity of analysis,* it is assumed that jobs run indefinitely.
  - The OS schedules an arriving Job as if the job is going to last forever in the system
- This approximation reflects that the time slice is very small (in the order of milliseconds) compared to the execution times of the jobs (in the order of minutes).
- The assumption of infinite running time might be removed with negligible change to the final results, but with significant complications in the definitions and proofs.

The 2 Types of Scheduling

- Given a job, J, as a set of $|J|$ VPs, how is this job allocated over the n physical PEs of the concurrent computer so that the resources are efficiently utilized?
- More generally, how are VPs of the same and different jobs, say $J_1, J_2, \ldots, J_m$, allocated over the PEs to efficiently utilize all system resources?
  - This is called the problem of spatial scheduling or spatial allocation.
- Intuitively. Spatial schedule: "Where to run the VPs of job J?". Temporal schedule: "When to run the VPs of job J?".

. . . and more formally

Definition

A **spatial schedule** *is an n-tuple allocation function (The terms* **allocation** *and* **spatial scheduling** *will be used interchangeably)*

$$\mathcal{A} : \mathcal{J} \longrightarrow \boldsymbol{N}^{|\Pi|}$$

such that for any job J, $\mathcal{A}(J) = (a_1, a_2, \ldots, a_n)$*,* a_i *VPs of J are allocated to processor* $\pi_i \in \Pi$*,* $(\sum_{i=1}^{n} a_i = |J|)$*.*

. . . and more formally (Cont'd)

Definition

A **temporal schedule for a time slice** *t is defined as an n-tuple function, $S(t)$, such that $S(t)[i]$ is the job run by π_i in the time slice t. A* **temporal schedule**, *$\mathcal{S}$, is a time ordered sequence of $S(t)$ for all $t > 0$.*

Definition

Define a schedule, $\mathcal{S}$, to be **impartial** *if every job $J \in \mathcal{J}$ occurs infinitely many times in $\mathcal{S}$.*

An impartial schedule ensures that every job in it completes.

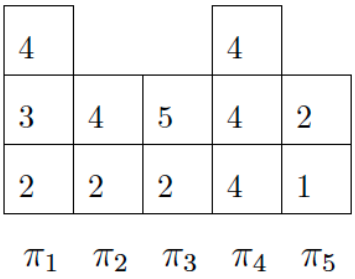
Example

Consider a machine with 5 PEs and 5 jobs:

- J_1 requires 1 VP,
- J_2 requires 4 VPs,
- J_3 requires 1 VP,
- J_4 requires 5 VPs,
- J_5 requires 1 VP.

. . . suppose we allocate the VPs of the Jobs as follows . . .

Example (Graphical)



With the following allocation functions:
$\mathcal{A}(J_1) = (0,0,0,0,1)$ $\mathcal{A}(J_2) = (1,1,1,0,1)$
$\mathcal{A}(J_3) = (1,0,0,0,0)$ $\mathcal{A}(J_4) = (1,1,0,3,0)$
$\mathcal{A}(J_5) = (0,0,1,0,0)$

Job Execution

- **Global communication:** all VPs of a given Job execute a communication instruction.
- **Legal execution of a Job:** no VP of the Job is ahead of any other VP of the same job by more than one global communication step at any time.
- **Legal scheduling strategy:** will always choose any trailing VP before any non-trailing VP.
- A **step** begins when all VPs of a job have executed the same number of global communications and it ends at the occurrence of the same condition at a future time slice (after at least one VP of the job has been executed).

A Job's Schedule Vector

Definition

*Define a job J's t-***schedule vector***, $\epsilon(t, J)$ as a 0-1 n-tuple such that,*

$$\epsilon(t, J)[k] = \begin{cases} 1 & \text{if } \pi_k \in \Pi \text{ runs job } J \text{ in time slice } t \\ 0 & \text{otherwise} \end{cases}$$

A Job's Progress Vector

Definition

Define **progress of a job** *(at the begining of a time slice t) as an n-tuple function*

$$\mathcal{P} : t \times \mathcal{J} \longrightarrow \boldsymbol{N}^{\Pi}$$

such that:

1. $\mathcal{P}(t, J) = \vec{0}$ *for* $t = 0$,
2. *If* $\mathcal{P}(t-1, J) + \epsilon(t, J) = \mathcal{A}(J)$ *then* $\mathcal{P}(t, J) = \vec{0}$,
   else $\mathcal{P}(t, J) = \mathcal{P}(t-1, J) + \epsilon(t, J)$ *(where the addition is componentwise)*
3. $\mathcal{P}(t, J) \leq \mathcal{A}(J)$ *where the* $\leq$ *order is applied componentwise.*

Progress Vector in English

The intuition behind the progress vector is as follows:

Start at the first step of a job J, let job J proceed on the processor's having 1's on the $\epsilon(t, J)$. If the progress vector reaches $\mathcal{A}(J)$ then another step of job J is complete and progress counters associated with each processor are reset. A new step cannot be started until the previous one is complete. Every job has a progress vector associated with it at the beginning of every time slice.

A more complete Example

Consider an allocation on 5 processors (same example):

4			4	
3	4	5	4	2
2	2	2	4	1
π_1	π_2	π_3	π_4	π_5

And consider the problem of scheduling J_4.

A more complete Example (Cont'd)

An arbitrary schedule for job J_4 is as follows:

$$\begin{array}{rcl} \mathcal{A}(J_4) & = & (1,1,0,3,0) \\ P(0,J_4) & = & (0,0,0,0,0) \\ \epsilon(1,J_4) & = & (0,1,0,1,0) \end{array}$$

$$\left[\begin{array}{rcl} P(0,J_4)+\epsilon(1,J_4) & = & P(1,J_4) \\ & & \epsilon(2,J_4) \end{array}\right. \begin{array}{l} = (0,1,0,1,0) \\ = (1,0,0,1,0) \end{array}$$

$\epsilon(1,J_4) = (0,1,0,1,0)$ indicates that processors π_2 and π_4 run job J_4.
$\epsilon(1,J_4)$ is added to $P(0,J_4)$ componentwise to give the new progress vector $P(1,J_4)$ for job J_4 and so on.

A more complete Example (Cont'd)

Continuing to schedule J_4 ...

$$\left[\begin{array}{rcl} P(1,J_4)+\epsilon(2,J_4) & = & P(2,J_4) \\ & & \epsilon(3,J_4) \end{array}\right. \begin{array}{l} = (1,1,0,2,0) \\ = (0,0,0,1,0) \end{array}$$

$$P(2,J_4)+\epsilon(3,J_4) = (1,1,0,3,0)$$

Now, $P(3,J_4) = \mathcal{A}(J_4)$, so reset, $P(3,J_4) = (0,0,0,0,0)$

System's Progress and Schedule

Definition

Define the **system progress matrix**, *Q, as a* $(n \times m)$*-array of progress vectors (columns) of all jobs (rows) in the system.*

Definition

Define the **system schedule matrix**, *S, as a* $n -$ *column matrix such that each column corresponds to a physical processor while each row corresponds to a time slice. Each entry is numbered with the job number whose VP is executed at that time slice.*

A Schedule for the Example

Time/PE	π_1	π_2	π_3	π_4	π_5
1	2	2	2	4	1
2	3	4	5	4	2
3	4			4	

The same three lines can be repeated until all jobs terminate.

The above will be dubbed the **Scheduling Matrix**

HINT: We'll be seeing periodic schedules.

Another Schedule for the Example

Time/PE	π_1	π_2	π_3	π_4	π_5
1	2	2	2	4	2
2	3	4	5	4	1
3	4		5	4	1
4	2	2	2	4	2
5	3	4	5	4	1
6	4		5	4	1

Same period as the previous schedule (3) but only one idle slice per period.

Schedules and Allocations

- A Schedule will strongly depend upon the initial Allocation.
- Challenge: Find another allocation/schedule for the example such that there are no idle time slices.

Metrics

There are two points of view when measuring the system's performance.

- One focuses on the functional quality from the system manager's point of view, who is concerned with the throughput and utilization of the machine.
- The second focuses on the functional quality from the users' point of view who expects the system to respond within a specific time.

Scheduling Metrics

For the case of Scheduling:

- Question: Which schedule is the "best"?
- The parallel OS should be able to satisfy two objectives:
  - Minimize processor idling time (important for the System).
  - Guarantee an upper limit on the system response time to every job (important to the single user).

Idling Ratio

Definition

For a n-processor system, for a schedule $\mathcal{S}$, and over and interval of Δt time slices, define an **idling ratio** *or* **throughput** $\imath_{\mathcal{S}} : \boldsymbol{N} \rightarrow \boldsymbol{R}$ *as:*

$$\imath_{\mathcal{S}}(\Delta t) = \frac{\sum\limits_{k \in \Delta t} i_k}{n \times \Delta t}$$

where i_k is the total number of idling processors during time slice k.

The idling ratio is a measure of the resource utilization.

Idling Ratio for the Example

Time/PE	π_1	π_2	π_3	π_4	π_5
1	2	2	2	4	1
2	3	4	5	4	2
3	4			4	

Idling Ratio for this schedule over the period of 3 time slices =
$3/15 = 1/5 = 20\%$

Idling Ratio for the Example

Time/PE	π_1	π_2	π_3	π_4	π_5
1	2	2	2	4	2
2	3	4	5	4	1
3	4		5	4	1

Idling Ratio for this schedule over the period of 3 time slices = $1/15$

An Interesting Case

Consider a system where only one job is running and utilizes all processors (J_1 has 5 VPs):

Time/PE	π_1	π_2	π_3	π_4	π_5
1	1	1	1	1	1

An Interesting Case

. . . SUDDENLY . . .

An Interesting Case

. . . SUDDENLY . . .

Another job J_2 arrives and requires 4 VPs

An Interesting Case

In a hurry, the system allocates and schedules J_2 in the apparently obvious manner:

Time/PE	π_1	π_2	π_3	π_4	π_5
1	1	1	1	1	1
2		2	2	2	2

With an Idling Ratio of $1/10 = 10\%$

An Interesting Case

BUT ... Consider the following 0% Idling Ratio Schedule:

Time/PE	π_1	π_2	π_3	π_4	π_5
1	1	2	2	2	2
2	1	2	2	2	2
3	1	2	2	2	2
4	1	2	2	2	2
5	1	2	2	2	2

The response time of J_1 was sacrificed in favor of full 100% Physical Processor utilization. AN UNHAPPY USER !!!

Happiness Function

Definition

Define the **happiness function**, $\hbar_{\Delta t}(J)$, *of a job* J *for a time length* Δt *as,*

$$\hbar_{\Delta t}(J) = \min_{t \geq 0} \left\{ \frac{\sum_{\tau=t}^{t+\Delta t-1} \sum_{\pi \in \Pi} \epsilon(\tau, J)}{|J| \cdot \Delta t} \right\}$$

The time interval Δt is the **Impatience Latency** of the user.

Happiness Function

- Intuitively, happiness represents the machine power that a job is given during a time slice. The term in the braces represents the fraction of the full parallel speedup that was achieved by job J in time interval $(t, t + \Delta t - 1)$.
- Δt represents the expected **impatience latency** of the user. It is the minimum time the user is required to wait before the system responds.

Virtual Happiness

- The proposed parallel OS paradigm allows the user to program in a virtual parallel model with no limit on the number of VPs in a job.
- To get a handle on the maximum parallel speedup that can be achieved for a job given n physical processors, define **virtual happiness**,
  $v(J) = \min\{1, \frac{n}{|J|}\}$.
- Virtual happiness of a job J represents the happiness of the job if the entire machine is used by J.
- Hence, J cannot be happier than $v(J)$.

Happiness is a Balancing Act

- A Job (user) will be happier if it acquires more parallel speedup over time.
- If a schedule is unable to provide the required happiness for a job, rescheduling with a different allocation may have to be considered.
- If the required happiness for a set of jobs cannot be achieved by any allocation, then some jobs must be queued for later allocation.
- A happy schedule is not necessarily optimal for throughput. It merely guarantees an upper limit on the response time.

Periodic Temporal Schedules

Definition

For an allocation $\mathcal{A}$, $\mathcal{S}_p = S(t), S(t+1), \ldots, S(t+p-1)$ is called a **periodic schedule**, *iff $\exists\, p > 0$ such that $S(t) = S(t+p)$ for all $t > t_s$, where t_s is called the startup time of the schedule. p is the period of the schedule.*

Also for each job J in $\mathcal{S}_p$,

$$\sum_{1 \le t \le p, 1 \le i \le |\Pi|} [S(t)[i] = J] = k \cdot |J|$$

must be true for some positive integer k to ensure that each job executes iat least k steps during each period (k need not be the same for every job).

How far did we go?

- This is *Work in Progress*. A lot to be done !
- We have proven a few theorems on Impartial Periodic Schedules.
  - Periodic Schedules are the BEST !!
  - We have conditions to minimize Idling Ratio and maximize Happiness.

Collaborations Welcome

Anybody interested ? . . . Let's work together !!!

THANKS !!