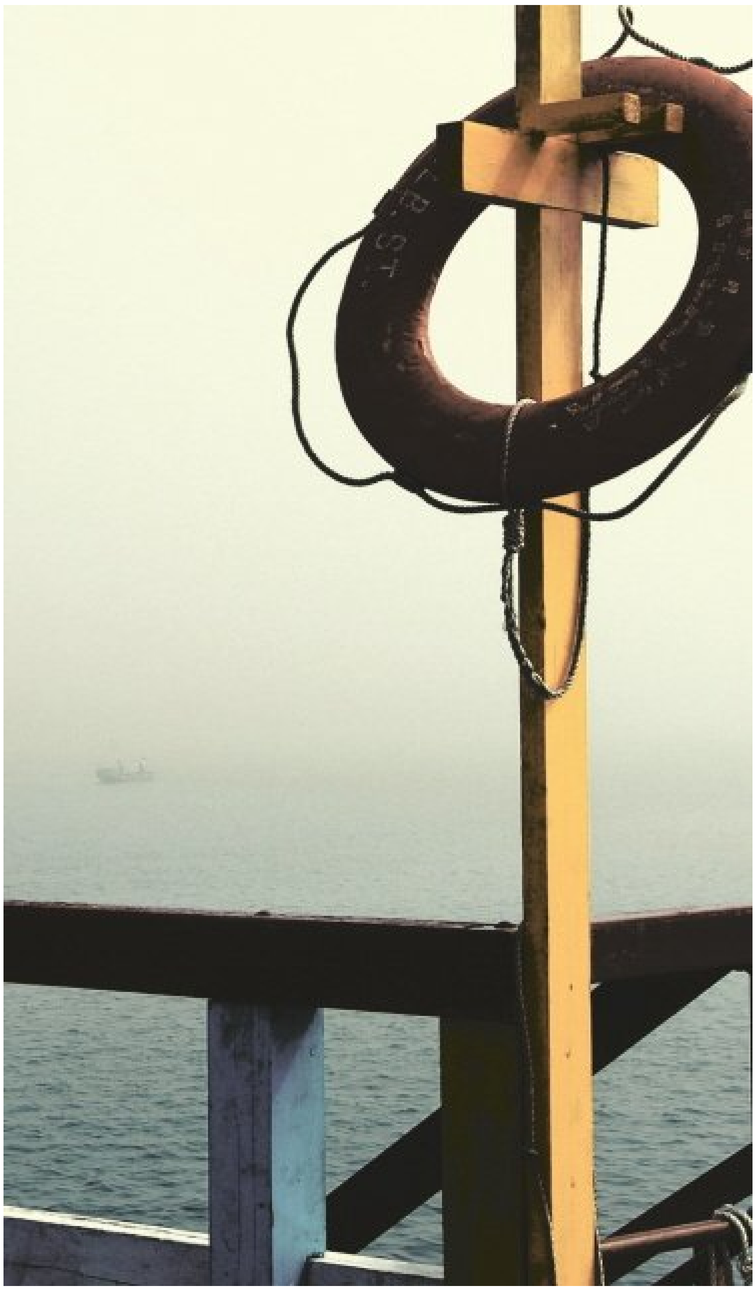
Parallel Multiprocessor Computing Model Using Independent Active OPUs

Snow-white paper

Mental research and proof of concept



Vladas Saulis

ProData PE
Klaipeda
Lithuania

February 17, 2007

Abstract

It is going to require quite a revolution
in software programming

(Dr Mark Bull,
Edinburgh Parallel
Computing Centre)

In the recent times we are facing a new shift in computing philosophy, caused by advance of new hardware architecture and even better performance. Multi-core architecture will become prevailing in near future. What we can do in order to take advantage of this? This paper is about the one of possible solutions we can have.

My proposed computing model also provides some answers to questions raised in recently published paper form Berkeley [1]. Here is a short list of advantages that can be reached using this model:

- Simple programming process and further maintenance
- Natural OPU (CPU) integration and migration (processing units can be added or removed on the fly)
- Automatic load-balancing
- No need of synchronization between task parts
- Little or no mutual locking on the system level

Lets see OPU programming model in more details.

Introduction

Lets imagine that every computing *operation or object* executes on different local or remote CPU. That CPU takes the object code and executes it indifferently. That means it doesn't care about the code it executes. So we call it OPU (Object Processing Unit). Every OPU is preloaded with simple round-robin multitasking (will be explained later) operating system with one or more language (object) interpreters. Lets see the simple example code:

```
for (i=0; i<N; i++) {
        a = a + b[i] * c[i];
}
```

Now lets introduce two new properties for every object of this task; for instance, variable `a` would have `a.complete` and `a.incomplete` properties. In my model there will be more additional properties, but we stop by these two for now. What does that change in calculations? It gives us the possibility to *split* execution! Now when the task in some stage will meet the statement like

```
        z = a + 1;
```

(which is also executed on separate OPU) it can now check variable `a` for *completeness.* In case this variable is still incomplete, the underlying logic can decide what to do in this case. It can wait for completeness, or it can be happy even with incomplete result, *intervening* and adding 1 to intermediate result. For the case the task logic must have a completed variable, it waits for it; in the same time other parts of the task still can be executed on other OPUs.

The OPU computing model also presumes the use of transactions. In case of some cyclic operations programmer must be sure that all variables changes in that cycle are made isolated and iterations had not been missed or lost. Obviously, there must be some support from compiler or operating system to take care of transaction commits and rollbacks.

Now we have some starting points for understanding the whole concept of OPU computing model. So, there are OPUs, tasks (set of objects) and *object queue.*

OPUs, Tasks and Queue

As we considered before – every single operation or object method can (but not necessary) be executed on separate OPU (remote or local). OPUs can be local processing units, like CPUs, FPUs, GPUs, specialized CPUs, IO processors, or any other kind of processing units. The remote OPU can be a whole remote computing unit, or any particular CPU within it. OPUs can be grouped by their characteristics and/or specialization.

OPUs are *pro-active!* This is a main difference from all existing computing models. This means that every single OPU decides what part of current task's operation to extract from the *queue.* Also, as I stated before, OPU doesn't care about the executed code or data (like normal CPU). It's a responsibility of the task (and programmer) to provide information about the required OPU type (or group) to the queue.

Now we came to the concept of *task*. First of all, - the task is not a process like in today's computing. It doesn't belong to any OPU. It is initiated by placing initial code (or method) into the queue too (by operating system or by external *injection*). And since then you cannot tell exactly which OPU will accept it for execution.

In OPU model the task is a *persistent object database* in system memory. I intentionally don't separate RAM and other storage (disks or whatever). It is a full responsibility of operating system to provide a transparent access to this object database.

And here I can present a bird-fly picture of OPU processing model:

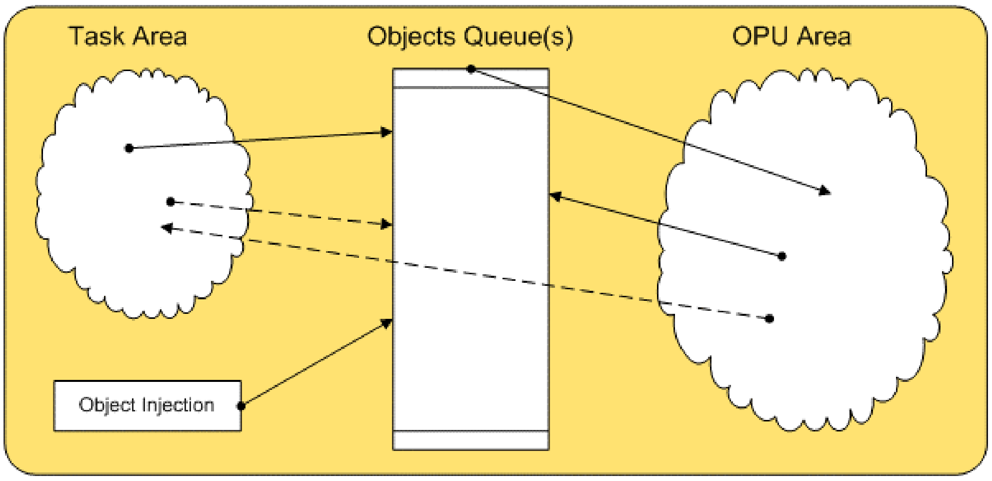


Fig. 1

The main principles of the proposed System are:

- Unidirectional flow of control

  All calculations in the system are done through the queue, with no acknowledgement or return of control from OPU, which will be in charge of executing the particular operation (or object method). The result of any operation is always a direct manipulation with the task's persistent object database.

- No return points in program logic flow

  This means that there is no place for legacy functions and procedures in OPU model, because they provide return values and so can regulate the flow of control.

- No processes as a solid code chunks in RAM memory, which are executed by assigned CPU(s).

  There is a persistent object database in any memory instead

- Isolated sub-context (subtasks)

  When some object method is executed on particular OPU, there can be local sub-context (or subtask) organized. This can be thought of as local *closures* as they exist in modern computing languages. It also can be treated as a local sub-transaction.

- OPUs are pro-active

  Each OPU is responsible for extracting the object methods for execution. No one OPU can be charged to do some task from outside.

- OPUs can be added or removed dynamically

  When some new OPUs appears in the system they just start to do their job – extracting and executing code from the queue. By analogy, any removed OPU just stops extracting the code from object queue. So, when system performance goes down, you can just add some amount of OPUs to the system (remember – they can be remote too).

- <u>Any OPU can pass to execute any operation remotely too, by putting it to the queue!</u>

  This is the most important thing to understand. This is a very basis of the OPU computing model. This is a gateway to true parallelism.

As you can see, everything in the system is spinning around the queue. So the only measure of overall system performance is - the current length of the queue.

Next important thing: Each OPU can execute several objects at the same time and not necessary from the same task. It is achieved by having a tiny multitasking operating system preloaded for each OPU. In this case each OPU can perceive its performance by itself. If it decides to be overloaded, it just stops extracting new objects form the queue, or *lowers the rate* of acquiring objects from it. By having very simple set of rules for polling the object queue, we can achieve natural self-balancing between OPUs in the system. On the other hand we can achieve an optimal load in case of many OPUs almost automatically.

For performance reasons not all objects in the system must be executed on different OPUs. For that purpose I can introduce some other built-in properties for objects. First of all there must be `.local` and `.remote` properties, as well as `.OPU` property. For those operations which must be done continuously on the same OPU we can set .local property. If we decide that the operation can be processed in parallel, or in other OPU environment – we set .remote and .OPU properties. .OPU property should be an object with the set of system setting which go to the queue. Other OPUs extract operations in accordance with these settings. Not every object can be extracted by particular OPU! The .OPU property can be set so, that this operation is dedicated to a "real-time" OPU group, for example, or it could be dedicated to OPU from particular IP address from outside system. It can be directed for specialized GPU or IO processor.

To sum everything up, I can present a list of characteristics and some unusual side effects of the proposing OPU computing model:

- Self load-balancing
- Seamless OPU migration
- Prioritizing using OPU groups (real-time, normal, batch)
- Simple programming at application level
- Programming logic becomes a state transition logic, and not sequential
- No threads, minimum locking and race conditions at the system level
- High system reliability; in case of some of OPUs failure, the whole system still can remain stable.
- Object reuse and method reloading on the system level
- Another similar object can be requested in case if current object fails
- Inter-task communication (former IPC) – simply by placing a foreign task's object to the queue.
- Direct object injection into the queue

An Example

This is another variation of a simple cycle, which is adapted for parallel calculation:

```
// Language Pseudo JavaScript

CycleAdd = function () {
     var a = 0;  // accumulator
     var b = [1,2,3,4,5,6,7,8,. . .];
     var c = [10,20,30,40,50, . . . ];

     task: "CYCLEADD"

     init: function () {
          a.transaction.Start();

          for (var i=0; i<b.length; i++) {
               a.incomplete +=
                   bigOperation(b[i],c[i]).complete;
          }

          a.transaction.End();

          if (a.transactionSuccess) {
               a.transaction.Commit();
          }
          else {
               a.transaction.Rollback();
               a.error = true;
          }
     }

     bigOperation: function (op1, op2) {
          var mult;
          Calculations.task = "CYCLEADD.CALC";
          mult = Calculations.multiply(op1,op2);
          mult.incomplete +=
                    Calculations.divide(op1,op2);
          return (mult);
     }

     this.add.OPU.local = true;

     this.bigOperation.OPU.local = false;
     this.bigOperation.OPU.remote.hostname = "localhost";
     this.bigOperation.OPU.language = "Javascript";
     this.bigOperation.OPU.group = "CPU.generic";

     Calculations.OPU.local = false;
     Calculations.OPU.remote.IP = "212.212.212.254";
     Calculations.OPU.language = "Machine";
     Calculations.OPU.group = "CPU.binary";
}
```

I'd like to state here that it's only a pseudo-code, although it looks like real Javascript code. Some internal mechanics of existing Javascript interpreters still are missing for make this code work for parallel computations. However, some emulation of parallelism is possible by use of AJAX-like technologies even now. But even in this case it would be only passive OPU calculations.

Now, we will look how this code could be executed in case of pro-active OPU model (fig 2).

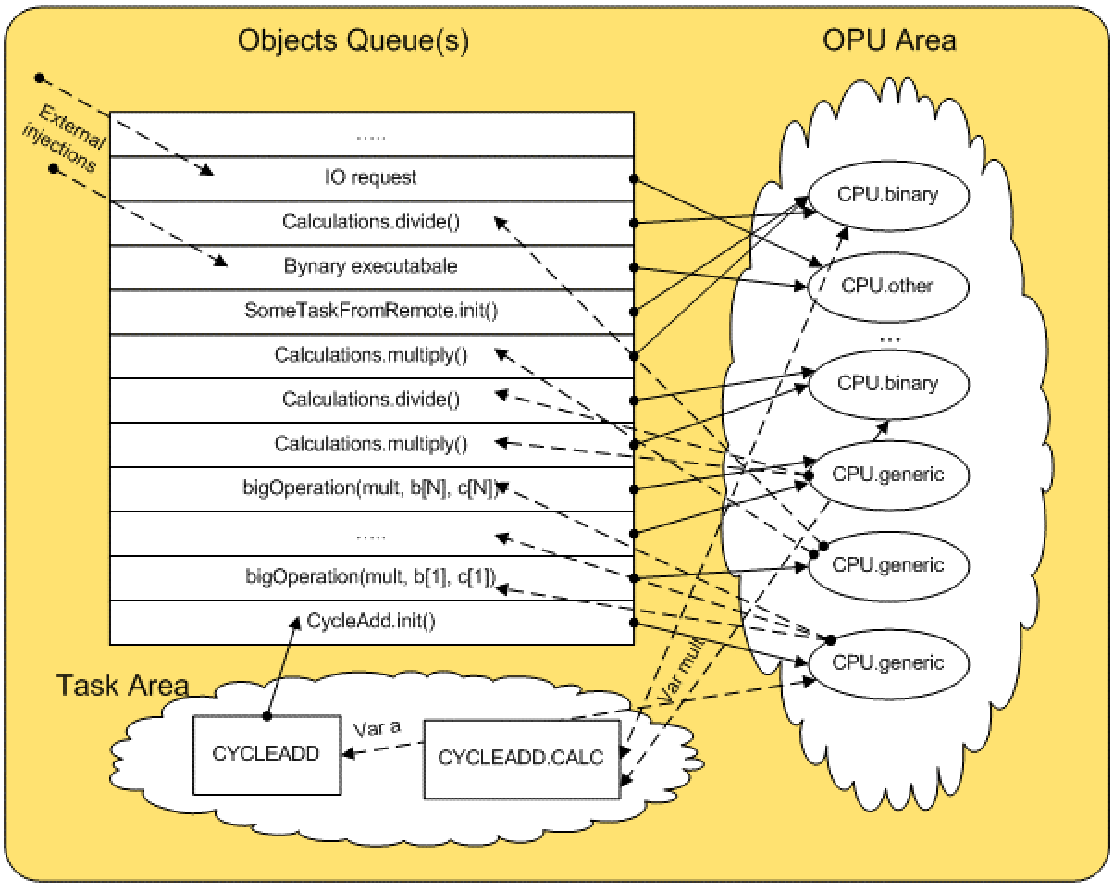


Fig. 2

As you can see, we don't even need any special "parallel or distributed" language. All we need is – to change internal semantics of the compiler or interpreter slightly, in order to work in parallel environment.

The above example shows how calculations are parallelized in OPU computing model. The initial object consists of local variables-properties `a`, `b` and `c`. Variable `a` is an accumulator for this cyclic addition program. To initiate the process, we put `init()` method into the queue. One of OPUs takes this method to execution, and then it sends all iterations of the `for()` cycle back to queue. Iterations (`bigOperation()` method calculation) then, in their turn, are taken by other OPUs, which belong strictly to corresponding OPU group, stated in the sending object's code (OPU.generic). All iterations are calculated in parallel and accumulated in variable `a`. Since an addition operation is transitive, additions can happen in any order, just as the results come. Thus, we can point interpreter to operate with incomplete intermediate values of our accumulator variable, and we set its property to `a.incomplete`. And this is the interpreter's responsibility to make sure that every addition would be performed *atomically*. Although, alternatively, we can establish a manual lock, having another internal property `a.lock`, for example.

Meanwhile, there is another task existing in the system (pre-launched, or it even can be a system task with other name), called `CYCLEADD.CALC`, which provides us with `Calculations` object and its methods. So, each `bigOperation()` method now *injects* two `Calculations` methods into the queue, and, again, are taken by any available OPU (busy or not) of corresponding kind (binary in this example). The binary type of OPU could mean that `Calculations` object probably is implemented as a legacy binary executable, written in C or C++, and is executed in non-parallel manner. Parallelization may end at this point.

The injection of both methods simultaneously becomes possible, because their addition is also transitive, so we can have `mult.incomplete` value here by design too. Note, that execution stops only when we have to wait for completed value.

As you can realize, `bigOperation()` method has a return statement. But this is only a syntactic sugar. But in reality, `mult` variable is sent indirectly to the context of `init()` method, which resides in the CYCLEADD task. When calling some method, a calling side must create a temporary variable in its context, so the called method could associate its internal variable(s) to it. After all that, the OPU which still executes an `init()` method, can add the *completed* value of `mult` to *incomplete* value of `a`. So, the return statement only tells interpreter what internal variable to set for result.

Transactions and Versioning

Parallel OPU computation principle may be simplistically seen as a multi-user database access. We may consider that every parallelized operation is a 'user', which is accessing local or remote data on some server. Each 'user' sends the data together with code to a particular remote database server and hopes it will be executed. However, it doesn't expect the calculated results to come back immediately, instead, it polls for the results in some time in future (when results are really needed by program logic). If results are not yet ready the program, depending on its internal logic, may either stop execution of the statement and continue polling results, or may set an error condition. And such behavior is a very important moment for all parallel OPU model! In case of error conditions we may have a chance to use alternative ways for performing operations. This is a possible way in creation of self-healing reflecting and redundant systems, which may adequately respond in timely changing or unstable systems and environments. More than that – the erroneous parts and paths of such systems are isolated from the execution path so this may lead to almost of no impact to the overall process.

You may already realize that we need some transactional mechanics and protocols defined for working in such highly parallel and cross-domain environments. This transactional protocol is also needed when using cycle-level parallelism. How does the parallel program would know that all iterations of the cycle are processed and processed correctly? In other words – how to know that all in-cycle data is acquisited correctly? This is a place where transactions come to light. Cycles may be small but may also be huge, spreading across multiply domains or performing huge internal subtasks. And we need to know that every part of resulting data came correctly (or even incorrectly, or didn't come at all). There must be mechanism to mark every cycle's iteration, sending the version number together with the request, and then to check against this number when results come back. When all iterations' results have come, the transaction may be committed; otherwise it could be rolled back or another appropriate actions may have been taken. You may, again, see the similarities with the multi-user database environments here.

Versioning is another important part in parallel multiprocessor computing using proposed model. Ones there are transactions - there should be versioning too. Furthermore, versions may be used even without their connection to transactions. Every single operation (or program statement) may operate on a particular variable version if needed. Suppose you have a real-time nuclear reactor process. You are performing some operations in a cycle, which increases some variable's value. The cycle may go forever, but when the number of iterations or the variable itself reaches some critical point, it must trigger an action. Every version of variable may be accessed along the way and used in upcoming program logic. This logic can even be generated or set on the fly. This is another very important impact of proposed parallel computing model. When there is no necessity for getting results immediately (waiting for execution path to return), the above program or system behavior becomes possible.

Levels of Parallelism. Locks and Stacks

Although cycle-level parallelism is the most obvious usage case, our proposed model is not limited only to it. In our model parallelisation may occur at any abstraction level. It may also be statement-level, object-level, and in some future, even instruction-level. Everything depends on parallel program design. The design itself now is not limited to an old sequential psychology. The important note here is that our model is not only for making existing program algorithms to run in parallel way, but also for creating brand new set of programs and systems, which becomes possible with our new paradigm. Statement-level parallelism means that the program is executing its statements all at once, while some of statements will wait for upcoming data and/or events. Thus, the overall program task becomes self-executing and existing until it either receives an explicit halting instruction (by injection), or naturally completes all incomplete variables within statements.

Execution paths of all statements are disconnected. This fact completely eliminates the current practice of using stacks for execution control. Stacks are of no use in such parallel environment. If there is no need of legacy subroutines, - no need of stacks! All execution becomes data-driven instead of process-driven. All execution synchronization is separately done by each data object and in isolation with other objects. Locking becomes data-bound and is more simple, isolated, smooth and manageable, which is opposite to process-bound locking. Data-bound locking is more self-contained (because it can be completely described by data container) and does not interfere much with other parts of the task.

The Timeout Problem

One of the main possible problems in our proposed Object Flow Model (this is another name for the proposing parallel computing model), which stands in front of all other - is a Timeout problem. This problem plays intact with well-known Halting problem [5], the paraphrased version of which could be following:

> "Given a program and an input to the program, determine *how long the program will be executed* when it is given that input".

For our parallel model this question becomes very important. How long any variable should wait for the result? What time is to be set to finally ensure that the variable cannot be completed because of possible fault, cycling, connection lost or whatever else? This also is important for choosing alternative paths in case of errors.

One possible solution is to make parallel operations so smooth that ever happening long timeouts could mean an inevitable fault. However, this solution stops working when a high level of parallelisation hierarchy is reached. Another possible solution is a prediction by fact mechanism. We can set predicted and/or predefined timeout values basing on previous execution times of similar or same operations. Anyway, the Timeout problem is still an open question and may be qualified as the most important problem.

How to Implement?

As we can see from example, we need not invent any new specialized programming languages in order to implement the proposed parallel computing model. All we need is - to slightly modify the behavior of existing ones. Also, in order to take the full advantage of the model, it would be good to write a special Operating System, based on these concepts.

These goals may be achieved by following three steps:

- First step - emulator. Modify existing languages behavior by adding special objects and classes, which can imitate parallel behavior. Parallelisation then may be done using classic client-server approach. The only requirement would be to have the same language at both sides (at every executing peer). One of possible real-life implementations could be – to use JavaScript environment at Web browser, as well as at the Web server, using Ajax for making asynchronous calls for object propagation and acquisition. This first step is to be used only for proving the concept, emulating parallel environment. The complete solution can't be achieved at this step.
- Second step – interpreter. Modify existing language interpreter to handle parallel behavior. At this step specialized objects and classes must be incorporated into the language. The modified interpreter must implicitly provide us with calls to other local or remote peers for placing objects and the data into local/remote queues, as well as provide an in-built interface for further polling of results.
- Third step – complete parallel OS. There can be two phases during OS implementation. First and most simple implementation can be based on an existing OS, with preference to message-passing micro-kernel-based ones, such like Mach or Hurd. A message-passing OS architecture could be naturally exploited for making parallel OS prototype. Next phase should be a complete specialized parallel OS.

Works on first step implementation have already been started. You can find our Proof of Concept project page at: http:// 195.178.176.98/Memel.

Conclusion

The proposed Object Flow Model shows how it can be possible to design self-parallelized programming languages and systems and, at the same time, to fully utilize any dynamic range of available processor units (or OPUs) in a self-balanced way. Object Flow Model also shows that parallel program may completely eliminate any sequential parts out of program. Even seemingly strict sequential parts of program become just a set of occasional mutual wait operations, which are, in their turn, executed in parallel. There is no place for sequential programming in this model anymore! And this is the most important result of proposed model, which may revise or even break the famous Amdahl's – Gustafson's Law [4] in some near future.

References

[1] *The Landscape of Parallel Computing Research: A View from Berkeley, Dec 18, 2006,* *http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html*

[2] **Cleo Saulnier***, "Software Development by Vorlath", blogposts series,* http://my.opera.com/Vorlath

[3] **Vladas Saulis***, "Infinite Power Computing Theory", blogposts series,* http://my.opera.com/vladas

[4] **Gene Amdahl**. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. *AFIPS Conference Proceedings, (30), pp. 483-485*, 1967.

[5] **Alan Turing**. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2, 42 (1936), pp 230-265,* 1936.