PARALLEL MULTIPROCESSOR COMPUTING MODEL USING PARALLEL JAVASCRIPT MACHINE

Vladas Saulis

PRODATA PE, Debreceno St. 35-8, LT-94166 Klaipeda, Lithuania

Site: http://parallel-js.net:81, e-mail: saulis.vladas@gmail.com

March 14, 2024

Abstract

This paper presents a new programming model that can utilize multi-core CPU systems in a simple and auto-balanced way. This model also proposes an easier programming paradigm for developing parallel tasks and systems in most massively parallel computation areas, such as weather prediction, nuclear physics, search engines, etc.

In 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 technology in the near future.

What can we do in order to take advantage of this? This paper is about one of the possible solutions we can have.

The proposed computing model (which is named “Object Flow Model”) also provides some answers to questions raised in the well-known 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 when object processing units (OPUs) can be added or removed on the fly
- Automatic load-balancing
- No need for synchronization between task parts
- Little or no mutual locking on the system level

All these characteristics are implemented in the Parallel JavaScript Machine (PJM), which is described below in this document. PJM may be perceived as a mini operation system which controls multiple JavaScript tasks, multiple users and multiple front-end consoles.

1. Introduction

The main goal of PJM (OS) is to simplify parallel programming by introducing special instructions (hints) that are represented by specifically crafted comments in form of *//#pragma <name>*.

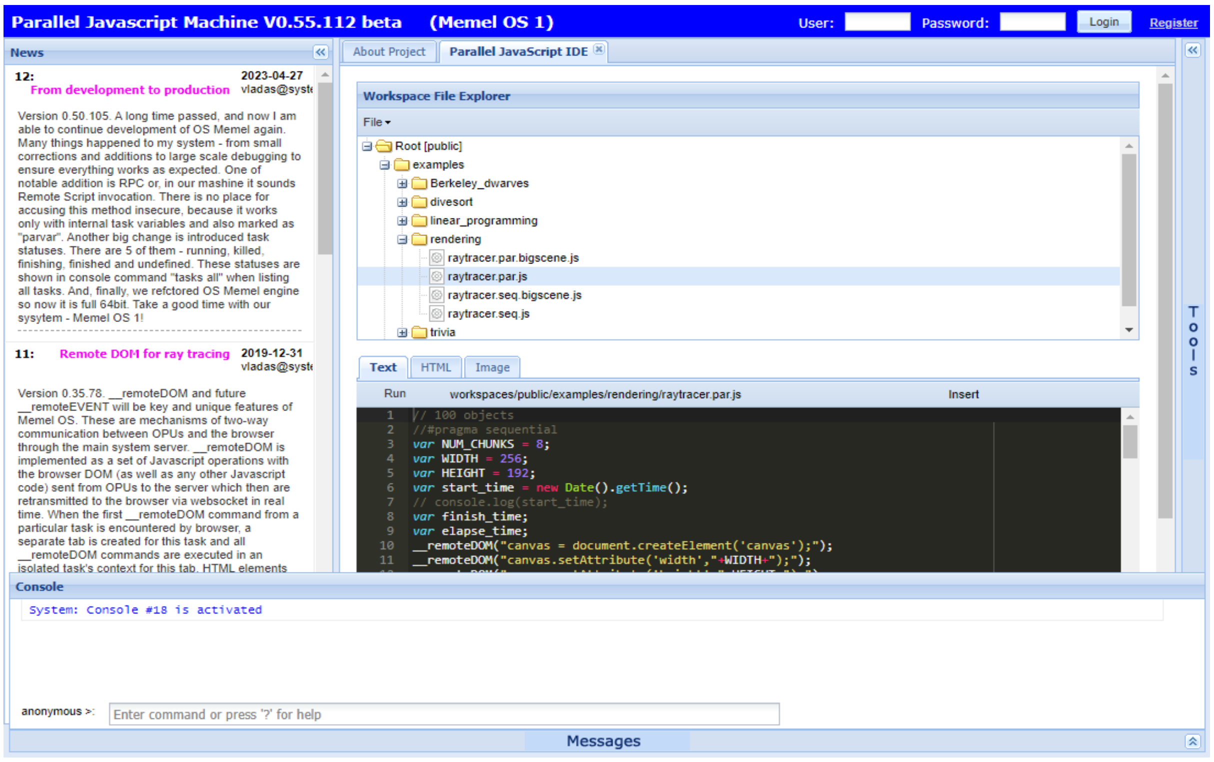
The Parallel JavaScript Machine uses NodeJS, and is implemented as a Web server for the frontend, as well as a server for the OPUs that really do the parallel execution of code. OPUs are small JavaScript network clients implemented in NodeJS too. There may be as many of them as necessary, connected to the main parallel machine server either locally or remotely. The overall performance of parallel processing strongly depends on the number of connected OPUs.

All system parts, working together, may be understood as a mini-OS which launches and parses the running JavaScript tasks, puts their chunks into system execution queue and provides some kind of cooperative multitasking between selected chunks. The calculation results are printed to the Web client's console by pipelining `console.log` output from OPUs through the main server. Every OPU is assigned to its own CPU (local or remote) and works through a simple round-robin scheduler (will be explained later). This is achieved by use of PM2 process manager Node module.

"Although compatibility with old binaries and C programs is valuable to industry, and some researchers are trying to help multicore product plans succeed, we have been thinking bolder thoughts. Our aim is to realize thousands of processors…"[1] not necessary on the single physical computer, but throughout the network, all orchestrated by the central server unit. From the system point of view all CPUs/OPUs are operating via internal network socket protocol which doesn't make distinction between local and remote CPUs. The server (PJM) and clients (OPUs) – all written in JavaScript, so it's not compatible with C binaries [yet].

All parallel programs, which are running in PJM, must not be using ES6+ JavaScript extensions (must be using ES5). It's important to state here that such extensions as *classes, arrow functions, lets and consts, and, especially, async/await* are hardly parallelizable, requiring more thorough JavaScript internals research. This is why PJM is running on NodeJS V8.2.1. All node modules are locked to this version for better performance and clarity of concepts.

PJM is controlled and tasks are running using Web Console which resides at http://parallel-js.net:8888. Here is how it usually look like:



12: **From development to production** 2023-04-27 vladas@syste

Version 0.50.105. A long time passed, and now I am able to continue development of OS Memel again. Many things happened to my system - from small corrections and additions to large scale debugging to ensure everything works as expected. One of notable addition is RPC or, in our mashine it sounds Remote Script invocation. There is no place for accusing this method insecure, because it works only with internal task variables and also marked as "parvar". Another big change is introduced task statuses. There are 5 of them - running, killed, finishing, finished and undefined. These statuses are shown in console command "tasks all" when listing all tasks. And, finally, we refctored OS Memel engine so now it is full 64bit. Take a good time with our sysytem - Memel OS 1!

11: **Remote DOM for ray tracing** 2019-12-31 vladas@syste

Version 0.35.78. __remoteDOM and future __remoteEVENT will be key and unique features of Memel OS. These are mechanisms of two-way communication between OPUs and the browser through the main system server. __remoteDOM is implemented as a set of Javascript operations with the browser DOM (as well as any other Javascript code) sent from OPUs to the server which then are retransmitted to the browser via websocket in real time. When the first __remoteDOM command from a particular task is encountered by browser, a separate tab is created for this task and all __remoteDOM commands are executed in an isolated task's context for this tab. HTML elements

```
// 100 objects
//#pragma sequential
var NUM_CHUNKS = 8;
var WIDTH = 256;
var HEIGHT = 192;
var start_time = new Date().getTime();
// console.log(start_time);
var finish_time;
var elapse_time;
__remoteDOM("canvas = document.createElement('canvas');");
__remoteDOM("canvas.setAttribute('width',"+WIDTH+");");
```

```
System: Console #18 is activated
```

Fig. 1 Parallel JavaScript Machine Web Console

The Web Console is written using the ExtJS framework and connected to the PJM server with the use of *express.js HTTP* NodeJS module. Another part of Web Console (called Console and Messages) is connected using WebSockets. This is where all programs' output and system messages are coming in real time. It is important to know that no real calculations are performed in the Web Console – all calculations are done on the PJM server and OPUs.

2. System design and implementation

Consider following trivial code snippet:

```
for (i = 0;  i < N;  i++) {
   a = a + b[i] * c[i];
}                                       example 1.
```

Now, after inserting *//#pragmas* for parallel execution, the same code will look as follows:

```
//#pragma parallel
//#pragma parvar a
for (i = 0;  i < N;  i++) {
   a = a + b[i] * c[i];
}
//#pragma wait                          example 1a.
```

2.1. #pragmas – an instrument for parallelization

All of *#pragmas* are interpreted during task (the running parallel JS program) launch and initialization. First, recursive parallelizer, then – analyzer, both of them find and apply all *#pragma* hints.

Here is a complete list of *#pragmas* with their meanings (omitting the *pragma* keyword):

- ***sequential*** – runs task in legacy sequential mode;
- ***parallel*** – running next part of task in parallel mode;
- ***wait*** – it is an assembly point of a program; system waits for all program chunks to be finished, and all parvars are in unlocked state; system enters idle state (if no other tasks are running) and calls parallelizer;
- ***parvar <varname>*** - automatically handles var's locking and unlocking between program chunks;
- ***noautoparvar*** – disable automatic handling of all parvars; manually use get and set functions (explained below);
- ***beginblock*** – mark the beginning of parallel block, creating new task chunk for execution on OPU;
- ***endblock*** – set the end of parallel block chunk and send chunk to execution queue;
- ***cache <varname>*** – cache task variable to current OPU;
- ***cachefuncs*** – cache all functions to current OPU;
- ***dive*** – parse AST of current loop for parallelizing inner loop;
- ***remotedom/noremotedom*** – turn on/off remote DOM feature;
- ***dombuffersize <number>*** – number of DOM operations in the buffer;
- ***keepsetvars/resetsetvars*** – keep/reset static variables;
- ***starttime*** – report program start time;
- ***endtime*** – report program end time;

- ***numopus*** – overall active OPUs number in the system; creates internal variable ***__numOPU***;

There also are internal OPU functions which are handful for direct use in programs:

- ***__get_Par_Var_Value(<varname>, <force lock>, __job)*** – get task variable from the server and lock it, if it's possible. Force locking if the second parameter is ***true***;
- ***__set_Par_Var_Value(<varname>, <varvalue>, __job, <force>, <slice>)***- set task variable, unlocking it; force unlock if force is ***true***; slice array if applicable;
- ***__RPC_set(<varname>, <varvalue>, __job, <force>, <script>, <params>, <taskvars>)*** – remote script invocation on the server;
- ***__remoteDOM(<script>)*** – DOM functions, as well as other operations, which are passed through the server to Web client using WebSocket;
- ***__read_File_Sync(<path>, <blocking>, __job)*** - read any file from program's directory;
- ***__write_File_Sync(<path>, <data>, <options>, <append>, __job)*** – write to the file in program's directory;

2.2. Tasks, Task variables and Queue

Like any other OS, the Parallel JavaScript Machine Mini-OS has a kernel. The kernel operates with such internal structures as the Queue object, Task list and Task variables object.

When the task is launched, its source code is passed to parallelizing function, where it is separated into single statements (also can be grouped in blocks). Every statement or block then enqueued into the system's Queue for execution by OPUs, which are connected to the server and extract these statements from the Queue one by one in no predetermined OPU order. In parallel programming terms it is said that the tasks are parallelized on the ***statement level***, or with ***statement granularity***.

The parallelization process is controlled by special instructions (or hints), called #pragmas. Some of them sets the start of parallel/sequential code, other sets beginning/end of blocks, parallel variables (so-called *Parvars*), begin/end of the process timing. One of them, #pragma wait, is very important for the parallelization flow control and marks the *assembly point* in the task. At this point the parallelization of the task is suspended until all pending parallel chunks of code have finished their execution (for parallel loops it waits for all iterations to complete). After that the process of parallelization resumes from this point. The 'wait' process in the system is really non-blocking, letting other parts of the task continue their execution, which is exactly opposite to the ordinary one-threaded JavaScript interpreter.

During the process of parallelization, when some statements contain an internal block of statements (such as loops, marked blocks), the analyzer function is called. It provides more specialized lexical interpretation of code logic. For example, it interprets for loop's initial expression, condition and step. This information is then added to the parallel code chunks as an extra internal code.

Another important part of parallel tasks execution is the task variables handling. All task's variables are stored within the server Task object. Before the task chunk is put into the Queue, the parallelization function determines which variables are in use within this chunk. It puts these variables along with their values as a definition to the preamble of the code. After execution of the code chunk, the resulting values of variables are returned and set back to the server's task variables object.

There are some special *parallel variables* (Parvars), which are handled differently. These variables are *declared* by the #pragma parvar <var name> special instruction. This means that these variables can be accessed and set concurrently. All 4 basic arithmetic operations are transitive (if processed one by one), so they can be processed in any order, and the result will always be the same. The parallel for loop works exactly in this way. So, the accumulator variable for the loop must be set in the way that it could be accessed and modified concurrently. This is what Parvars are for. The kernel adds additional code fragments that synchronously calls the main server for parallel variable value and locks it in order to be modified after the code chunk had finished its execution. It is important that the code chunk needs to be fast and atomic, so it would lock the parvar for as short a time as possible.

Here is a bird's-eye view picture of OPU parallel processing model:

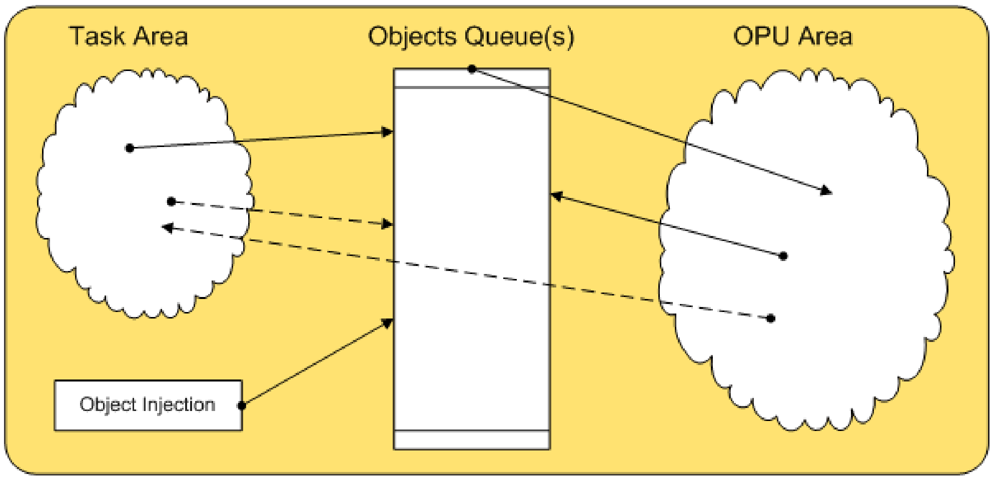


Figure 2. OPU parallel processing model – Parallel JavaScript Machine

2.3. Object processing unit

The OPU (Object Processing Unit) is a lightweight networking client (worker) which does the actual processing of parallelized chunks of code. OPU connects to the main server and scans the Queue for the next job in FIFO manner. When it finds a job, it executes it, returns the result (task variables) and scans again. If no jobs are found it enters an idle loop and waits. Therefore, OPUs are pro-active because they initiate job extracting and processing. There may be an unlimited number of OPUs, connected to the main server - locally or remotely - and working independently. Their number is only limited by the server's processing power and the network speed.

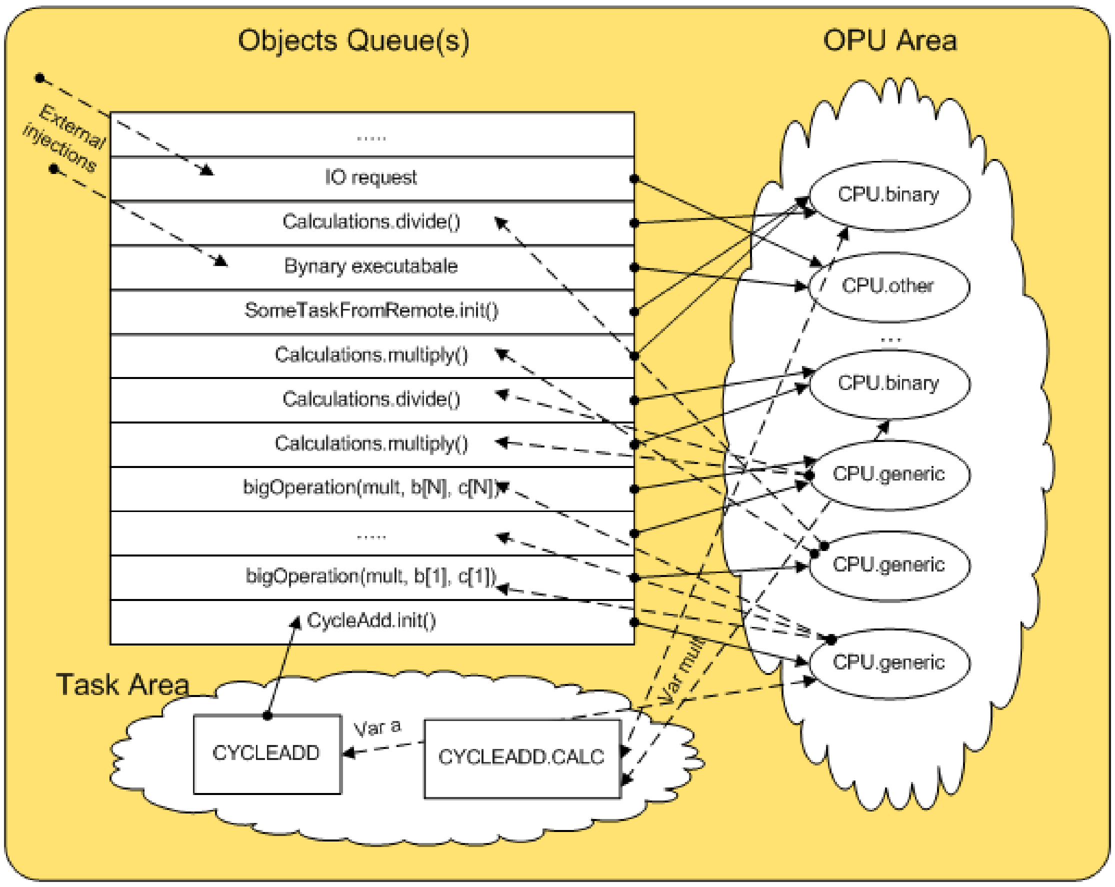


Figure 3 OPU dependence on Queue

The OPUs have another very important function. When they meet Parvars, they connect directly to the main server to get real time value of Parvar, and after Parvar has changed its value, they connect again and set its changed value to the corresponding task variable. When OPUs are processing Parvars in such a way, these are locked on the server to prevent race condition. Thus, it's said that the system has data bound control flow, which is opposite to the statement level control flow, as is usual for non-parallel systems.

The main principles of the OPU relations with the PJM are the following:

- 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). The result of any operation is always a direct manipulation with the task's persistent variables object.
- 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 that can regulate the flow of control.
- No processes as solid code chunks in RAM memory, which are executed on assigned CPU(s). There is a persistent object database in any memory instead.
- OPUs are pro-active

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

- OPUs can be added or removed dynamically

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

- Any OPU can pass to execute any operation remotely as well, by putting it to the queue.This is the most important thing to understand. This is the very basis of the OPU computing model. This is the gateway to true parallelism (now in development).

As is evident, everything in the system is spinning around the queue. And the only measure of overall system performance is - the current length of the queue.

To sum everything up, below is a list of characteristics and some unusual side effects of the 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 failure of some OPUs, the whole system still remains stable.
- Object reuse and method reloading on the system level
- Another similar object can be requested in case the 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 (in future versions)

2.4. System scheduler and Idle cycle

All OPUs are connected to the main system server via single network socket in non-blocking mode. When no new task chunks are available, the OPU enters an 'idle' state and waits for a 'wake-up' command, which is sent when a new chunk has appeared in the system queue. Every new chunk, when posted to the queue, is followed with the 'wake-up' command. When OPU gets the 'wake-up', then it sends 'getjob' back to the server to get new job.

As soon as the system server receives the 'getjob' command, it calls parallelizer (when all current chunks are finished), with possibly new chunks added to the system queue, and then it enters system scheduler. System scheduler works around system queue and has two modes:

1 – chunks are processed in queue order (FIFO);

2 – chunks are processed in tasks order (cooperative multitasking). This is the default mode.

All these operations are processed by the ‘extractor’ kernel function. When the next chunk is determined, it is sent to the ‘getjob’ issued OPU. Chunk processing will continue until all free OPUs are charged by job or queue becomes empty.

Scheduler mode can be changed from Web Console prompt, entering a command ***schedule [1|2]***.

2.5 Multi-user environment

Multi-user access is very important for this parallel machine. It lets users launch tasks independently and asynchronously. Every task in the system belongs to a particular user - and only that user can see his task's output. When some user is registered in the system, they get their own *workspace* with examples. They can then modify and save any file in their workspace, as well as to create new ones. The user's launched tasks may, in fact, execute in parallel. It depends on how often tasks wait for parallelization in *assembly points* (look at the previous section for this term). Such tasks behavior is often called *cooperative multitasking*. All tasks in the system should give a 'breath' for all other tasks in the system, and the #pragma wait instruction is a way to do this.

2.6. Web console and system commands

The PJM has a Web interface, consisting of personalized file manager, instant file editor, news pane and real-time message console. A user’s filesystem is ‘jailed’ and everyone’s root folder is different. At the initial start of the system default anonymous user filesystem is read-only. When the real user logs in, it enters to its root filesystem, which is fully read-write. To get real user access, a user must be created by *registering* in the system. Every new filesystem is provided with a folder called *‘examples*, where all the tests and benchmarks reside.

When some file is selected, its content is shown in the editor below the file manager. Files can be modified, saved (except for anonymous) and ran. All runnable files in this system have a *.js* extension, because all parallel programs are written in JavaScript.

The Web Console works as is usual for WebApps – using the HTTP protocol. But console messages (at the bottom) work using Web Sockets. Messages come from various sources and immediately both ways – from and to the system server, as well as from OPUs, tunneling them through the system server. All parallel programs’ output is directed to console, as well as error reporting and other system messages. Also, the system has several command line commands for various purposes, most of them for system monitoring.

Here is a full list of internal commands:

- ***`help or ?`*** – show commands list;
- ***`status`*** – shows queue length, number of active tasks and number of OPUs;
- ***`queue [n]`*** – lists queue entries; n – list last [n] entries;
- ***`schedule [n]`*** – set/get scheduler mode:
  1 – in queue order;
  2 – tasks round-robin;
  no parameter – shows current mode;
- ***`log […]`*** – logs variable like in console.log;
- ***`clear`*** – clears console;
- ***`tasks | ts [all]`*** – lists active tasks;
  all – lists all tasks;

- ***task | t[n]*** – views last task;
  n – view task number n;
- ***kill <n> -*** kill task number n;
- ***disable <n>*** - disable OPU number n;
- ***enable <n>*** - enable OPU number n;
- ***mload | ml*** – machine load (1.0 – optimal)

For now, all console commands are accessible to all users (for debugging purposes).

3. Examples and benchmarks

Examples are created for every new user registered in the system and reside in the 'examples' directory right next to the user's virtual 'root'. When entering 'examples' the user can see many directories, where the most notable one is the 'trivia' directory. This directory contains several simple examples that are made to easily compare the running time between sequential and parallel versions of tasks.

Here are the benchmarks of the programs in 'trivia' directory:

Program	seq.js	par.js – 8 OPUs	Performance gain
blocks	13.3 sec	1.9 sec	7x
double_loop	3 min 3 sec	56.9 sec	3.2x
for_loop	5.7 sec	1.6 sec	3.6x
long_funcs	13.3 sec	1.9 sec	7x
while_loop	23.3 sec	6.6 sec	3.5x

Table 1. 'trivia' benchmarks

Next notable examples are in the 'rendering' directory. It contains two ray-tracing examples with 300 objects (small scene) and with 3000 objects (big scene). These examples are from the class of 'embarrassingly' parallel tasks, where the rendering of each pixel is absolutely independent from rendering other pixels. Here are the benchmarks:

Program	seq.js	par.js – 8 OPUs	par.js – 16 OPUs	par.js – 20 OPUs
raytracer	7.8 sec	2.2 sec		3.5x
raytracer-bigscene	747 sec	199 sec	76.1 sec	67.6 sec 11x

Table2. raytracer benchmarks

Other examples are created to prove some of the "Berkeley dwarves" [1] problems. So far, only two dwarves are created and their level of parallelization examined. Here are the results:

Program	seq.js	par.js – 8 OPUs	Performance gain
embarrassingly-parallel (Monte Carlo)	22.3 sec	6.3 sec	3.5x
barnes_hut (N-Body-problem)	7.87 sec	7.67 sec	1.03x

Table 3. "Berkeley dwarves" benchmarks

The reason why the ‘barnes_hut’ parallel algorithm doesn’t show better results is yet to be investigated. This algorithm was ported from NESL language literally and it should be possible to refactor this algorithm in the future.

And finally, there is a class of examples under the directory ‘linear_programming’. Linear Programming (LP) is a linear optimization method to solve linear equations with NxM constraints matrix and M-dimension vector. The examples algorithm uses quite a rare combination of coordinate descent and conjugate gradients methods. It’s taken from [6], p.437. There are three examples with different matrix size – 150x500, 200x500 and 300x500. Each example was calculated using 5 OPUs, which had been considered optimal for these kind of tasks. All parallel versions performed 1.5 – 4 times faster than sequential counterparts.

It must be noted that this improvement is not because of native parallel behavior of the algorithm, but because of the increased convergention speed. The parallel version creates 5 variated independent streams of optimization paths. After each iteration, the variator variable ‘k’ is assigned to the most optimal path, decided by the value of minimal matrix norm ||x||g.

These examples are not to show the best solution for LP. Simplex method is still better for classical LP setup. These examples only show the possibility to improve iterative methods to solve them much faster. And even for LP there are super-large setups that can be solved faster than Simplex method.

Below are the benchmarks for all three LP setups:

Program	seq.js	par.js – 5 OPUs	Performance gain
linit-150x500	137 sec	32 sec	4.3x
linit-200x500	260 sec	172 sec	1.51x
linit-300x500	2905 sec	2055 sec	1.41x

Table 4. “Linear Programming” benchmarks

All benchmarks are performed on 8 x 11th Gen Intel Core i5-1135G7 @ 2.40 GHz.

4. Programming in JPM

Parallel programs for JPM must respect several rules:

- Parallelization is possible only in global scope, - not in functions;
- Functions work as helpers (like in procedural style of programming) and mostly are for code structuring, thus function programming as well as parallelizing inside functions are not possible;
- No classes OOP, instead convert them to the prototypal OOP;
- All variables must be declared using keyword ‘var’;
- JavaScript features should comply to the ES5 standard;
- No inline (‘arrow’) functions;
- No ‘async/await’;
- Modules loaded by ‘require’ keyword are not allowed (for security reasons);

5. The Timeout Problem

One of the main possible problems in the proposed Object Flow Model (another name for the proposed parallel computing model) that stands in front of all others - is a Timeout problem. This problem is similar to the well-known Halting problem [4], the paraphrased version of which could be the 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 should any variable wait for the result? What time is to be set to finally ensure that the variable cannot be unlocked, because of possible fault, looping, connection lost or whatever else issues? This is also important for choosing alternative paths in case of errors.

One possible solution is to make parallel operations so smooth that occasionally occurring long timeouts could mean an inevitable fault. However, this solution stops working when a high level of parallelization hierarchy is reached.

Another possible solution is a prediction by fact mechanism. We can set predicted and/or predefined timeout values based on previous execution times of similar or same operations.

In any case, the Timeout problem is still an open question and may be qualified as the most important problem.

6. Source code

The source code is available here for free and unlimited use. Extract this tarball from the root filesystem in Linux.

It's open-sourced under MIT license. © 2024, Prodata.

7. 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 programs may completely eliminate any sequential parts out of a program. Even seemingly strict sequential parts of a program may 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 [5] in some near future.

References

[1] **K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick**. The Landscape of Parallel Computing Research: A View from Berkeley. *Technical Report No. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences University of California at Berkeley,* December 18, 2006.

[2] **V. Saulis**. Infinite Power Computing Theory. *Posts series at Medium, https://medium.com/@saulis.vladas,* 2007.

[3] **C. Saulnier**. Software Development by Vorlath. *Blog posts series, http://my.opera.com/vorlath/blog,* 2006-2008

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

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

[6] **R.P. Fedorenko**. Approximate solution of optimal control problems, *Moscow, Science*, 1978