Virtue - a Different Approach To Human/Computer Interaction

Zorislav Šojat, Tomislav Ćosić, Karolj Skala*
* RBI/CIC, Zagreb, Croatia
sojat@irb.hr, tcosic@irb.hr, skala@irb.hr

Throughout the development of computing tools usage and computer science, from the calculating machines up and into the present day computers, the major technical problem to be solved was the reduction of the amount of components and their bulk. This led inevitably to the use of serial processing. Only quite recently we started mass producing and using multi-processing. Though the nature itself is infinitely parallelised, the technical problems of computer development led also to the development of such software tools and programming languages which mirrored the serial nature of computers, so that the serialisation of parallel natural processes is performed by humans, the programmers. However, modern day developments of scientific and every-day needs for computing power have led to the introduction of multiprocessors, GPUs, clusters, grids and clouds of computers, as to ascertain enough processing speed, power and memory for very complex algorithms. However, the human/computer interaction that supports these developments is still heavily based on "classical" computer programming languages, serial programming, and the multi-computer environment is accessed only through programmed interfaces, where all the burden of parallelising the, now well known, serial algorithms is again the job of (human) programmers.

Virtue is a development primarily based on the idea - as there already is so much we know of mathematical, logical and other important basic algorithms used in many fields, and their computer implementation(s) - that we would be able to "raise" the level of computer "understanding" more towards the level of our own, human language communication. This means that, for example, mathematical operations in Virtue are performed not only on integers and reals (floats), but also on complex numbers, quaternions and octonions, and that they are freely intermixed. Or that all logical operations work also on multi-levelled and multi-dimensional logical values (and not just Boolean). These are mathematically well defined operations used very often in visualisations, as well as scientific modelling. Furthermore, another basic underlying idea of the development of Virtue is that the language developed may not destroy a possibly parallel structure of data, which then allows direct and user (human) independent automatic parallel execution. Within such an approach interactive sentences can be "decoded" (executed, performed, saved, used...) on a single processor, a multi-core, multiprocessor, GPU, cluster, grid or cloud system, or any combination of them.

Designed on these basic principles, Virtue is a language which proposes a different approach, by keeping the inherent parallel structure of natural algorithms, and doing the parallel processing by itself, if it is algorithmically possible. Virtue is a syntactically very simple, yet semantically extremely complex language, offering no "reserved words", synonyms, automation of memoisation, multiple word contexts, combined data types of anything Virtue supports (e.g. functions, symbol names, scalars, multidimensional sub-structured arrays etc.), stochastic processing, multivalued and multidimensional logic operations, multidimensional sub-structured file access structures, continuations etc.

Therefore, due to this semantic richness and grammatical simpleness, in Virtue, for example, the text of the algorithm for Conway's "Game of Life" necessitates only 12 language tokens (7 words, 14 numbers in 3 vectors and 2 delimiters) in one sentence.[1]

Further development of the idea allows for development of a more syntactically rich very high level human oriented machine interaction language, which, combined with additional artificial intelligence components, appropriate ergonomic human presentation/sensory interfaces and with the integration of user style association memory, we sincerely hope can help the future development of Computer Science and Usage Practice.

I. Introduction

The challenges of the modern day world, the development of sciences and the development of social networking change quite radically the way computers are used as compared to the time when most of the basic computer construction principles were innovated. By basic computer construction principles we primarily mean the basic computer architecture based on monolithic processors executing serially specific instructions on individual values. Though many of the challenges of multiprocessing, execution parallelism, big data, cluster, grid and cloud computing are more or less, on the architectural level, solved, and throughout the development of computing a lot of physical and technical problems of multi-processing cooperation were solved and are being solved, there is still a huge lack in the development of appropriate human oriented interfaces with these (primarily hardware/firmware/systemware) complex integrated and/or dispersed computing systems.

Modern age computing needs to solve several important problems. We have to deal with enormous amounts of data which shall be mined, we necessitate High Performance Computing focusing on tightly coupled

[1] `MONADIC (1 1 1 1 0.5 1 1 1 1) [3 3] MONADIC RAVEL SUM (2.5 3 3.5) IDENTICAL ANY; MASK;.` This programme will work for any size of the "Game of Life" board.

parallel jobs for complex scientific and modelling applications, High Throughput Computing focusing on the efficient execution of a large number of loosely-coupled tasks, we need the Internet of Things, where individual appliances interrelate, and we need Personal Assistants...

And finally we have also a stringent need for High Productivity Computing, which stresses the importance of global Human-computer interaction results, i.e. the productivity, and not just the pure performance of the (already programmed) computer. High Performance Computing means speeding up the process of defining a problem to be solved and obtaining the results from the computer, that is shortening the overall problem solving time. That means that High Productivity Computing is primarily oriented towards shortening the time necessary to solve a new problem, that is, the time which is necessary for a Human to coordinate efforts with the Computer in getting an answer to a freshly thought out idea (i.e. problem to be solved).

The Virtue project is actually a product of a wish to integrate the present day computer science knowledge, as much as possible and feasible, into a consistent linguistic framework, with features which enable it to be "levelled up" towards the Human language communication possibilities, as to provide a personalised communication experience.

II. Present State of the Art

Throughout centuries of the development of computing equipment serial processing was more or less the only option, therefore, after the event of modern day computing, from the first half of the 20th century, and the development of programming languages, most of the computer-linguistic effort was put into the development of conventional serial programming languages. And, to cite John Backus from his ACM paper of 1978(!): "Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs." "Programming languages appear to be in trouble. Each successive language incorporates, with a little cleaning up, all the features of its predecessors plus a few more." "Each new language claims new and fashionable features, such as strong typing or structured control statements, but the plain fact is that few languages make programming sufficiently cheaper or more reliable to justify the cost of producing and learning to use them." "For twenty years programming languages have been steadily progressing toward their present condition of obesity; as a result, the study and invention of programming languages has lost much of its excitement."

Now it is already 36 years after John Backus made this statement, so his statement would today cite "For almost 60 years...". Because, unfortunately, modern day developments continued down these lines with those very few extremely popular and very programmer unfriendly languages like C, C++, Java, and a whole bunch of their syntactic followers, then the popular "scripting" languages, and the vast overpopulation of all kinds of so called "Application Programming Interfaces" superimposed over the already overcrowded linguistic situation of those languages, including the fact that some of such "language extension" interfaces even claim to be "languages" (e.g. OpenGL claims to be a Graphical Language, but is actually a very complicated, even complex, API). We can actually see that what Backus said so many years ago is getting towards human limits of comprehension possibilities. And this basic problem is actually the main bottleneck of our present day human-computer productivity.

Imagine a mathematically simple and effective visualisation – a four-dimensional hypersphere to be rendered as simple asterisks showing all the dots inside the sphere's radius, and spaces outside, the third and fourth dimension to be shown as two-dimensional slices. By just looking at such a very elementary visualisation the basic structure of the hypersphere can be easily deduced. Now find a C, C++ or Java programmer and ask him to make a short program to show such a sphere to you... And now, after they have shown you this four-dimensional sphere, ask them to show to you how it would look like in 5 or 6 dimensions... Or, better, do not ask them to do any of that for you in such a language, as it would take quite a lengthy time even for the best. (A non-optimised solution which we prototyped in C for 4 dimensions has more then 130 lines of source code![2] An optimized version would take much longer to develop.) And it is a kind of "one-execution is enough" request and programme.

But with the all-penetrating spread of computer equipment, networking, and data collecting and serving there are vast and rapidly growing needs for new computer applications on all levels of the human civilisation.

We started tackling these problems systematically several years ago, and an investigation into basic principles of computer programming, as well as the principles by which we serialize the inherently extremely massively parallel universe around us into serial algorithms was performed - and a new approach taken.

One of the major points which started being obvious during the investigation of the programming principles is that most (or almost all commonly used) computer languages, being primarily serially oriented both in their

[2] The definition of a new Virtue verb "sphere" which solves the above problem for any number of dimensions up to 8 -- taking three scalar numerical objects, expressed as two two-dimensional numbers (complex - 2d, quaternion -3d or 4d, or octonion -5d, -6d, -7d or -8d) for the centre and the space-size, and a real number – the radius -- is this sentence: `TRIADIC SPACE CENTRE MAGNITUDE GREATER '*' MULTIPLY; OPERATOR @sphere SET.`

What it says is that you want to have asterisks wherever the radius is greater than the magnitude of the centered space elements. The verb SPACE (synonymous to INTERVAL) with a scalar numerical object makes a space of indices from 1 (or 1i1, 1i1j1, 1i1j1k1...). The word CENTRE is a simple synonym for the verb SUBTRACT.

semantics and in their syntax – i.e. in their whole grammar, enforce the de-contextualisation of their data structures mostly in their imperative parts. In other words this means that though a complex data structure may be declared in the language, the operations on that data structure are linguistically restricted on individual data elements from that data structure. As a consequence of this approach the context of the individual data elements in a data structure has been erased from the programme, leading to important information loss, and the language executor (compiler, interpreter, assembler…) has theoretically[3] no possibility to 'know' that the user is actually doing parallel processing in a serial way.

Interestingly enough, although obviously a consequence of the above-mentioned centuries of serial computing[4], there are not many computer languages at all which do not do this 'de-contextualisation' of their data structure operations – notably between these are APL and APL-children (APL2, J, K, A+…), and the 'array processing' linguistic branch. Unfortunately, of those few languages which keep the operations context available for the executing processor and therefore allow the processor (usually an interpreter, not the low level processing units) to make intelligent decisions on the automatic parallelisation of each particular operation on (possibly) different data structures[5] there is presently (2014) no publicly (particularly open-source) available parallel implementations we are aware of. Many so-called parallel programming languages actually necessitate explicit programmer description of the parallelistic execution, as well as (for example in Occam) explicit inter-serial-part-of-the-parallel-algorithm communication.

III. Approach

Based on the above, the approach was taken to define a new data processing language, or better to say a new data processing language family, which would allow reasonably simple algorithm description for complex data structures, and which would allow the lowest level "processor" to automatically distribute (parallelise) the operations execution on a set of processing units. Generally the idea is to have a hierarchy of languages, whereas the lowest level one, described on these pages, has to be as close to the computer hardware level as possible, given the imposed complexity, and should actually behave as a **Virtual Executor**. The job of the Virtual Executor is to do all the necessary execution and parallelisation during execution based on the data, i.e. data types and data structures. Therefore it could be said that the resulting Virtual Executor is parallely programme and data structure driven.

The idea of such a language system was born when clusters of computers and grids started being developed on top of single-processor, multiprocessor, multithreading and multicore computers, as well as supercomputers of different architectures.

The major question with these collections of systems is: How do we programme them in a uniform, efficient and reasonably simple way? How do we integrate computers of different speeds, processors, memory sizes, computation widths[6] and byte sexes?

Is it possible to develop a Language system which could cope with the necessities of High Productivity Computing, High Performance Computing, High Throughput Computing, Big Data, Visualisations, Internet of Things... and be seamlessly applicable to an indefinite number of processing stations?

"What might a language look like in which parallelism is the default? How about data-parallel languages, in which you operate, at least conceptually, on all the elements of an array at the same time? These go back to APL in the 1960s, and there was a revival of interest in the 1980s when data-parallel computer architectures were in vogue. But they were not entirely satisfactory. I'm talking about a more general sort of language in which there are control structures, but designed for parallelism, rather than the sequential mindset of conventional structured programming. What if do loops and for loops were normally parallel, and you had to use a special declaration or keyword to indicate sequential execution? That might change your mindset a little bit." (Guy Steele, Dr Dobbs Journal 24 Nov 2005).[7]

With these questions in mind the development of the Virtue system started several years ago.

IV. Virtue

The result of this approach is the first version of **Virtue** – a grammatically extremely simple, yet

[3] Practically it is possible to automatically regain a certain knowledge on the data element context in specific cases, as e.g. when a loop transiting an array has a fixed number of iterations – based on pre-execution knowledge (by the programmer) of e.g. how many elements are in the processed data structure. There are certain "parallelising" compilers on the market for several much used languages, but they can do the "parallelisation" of the algorithm described in a serial language only for very special cases. Theoretically, the lost context information is not regainable.

[4] It is probably important to mention that between humans, e.g. in mathematics, the algorithms (as for example mathematical formulas) are described in a contextfull way, however, due to our own restrictions, we always calculate the results stepwise, usually completing as much as possible work on one element of an array before proceeding to the next, which is usually by applying the given formula completely on individual data points. This inherently human approach may have also lead to the adoption of strictly serial programming languages in the past.

Parallel programming mindframe, on the other hand, starts with the same contextfull algorithm (e.g. formula), but, as opposed to the serial mindframe, executes not the whole formula on the first data structure element, and then again on the next, but executes the first operation from the formula on each data element of the structure (e.g. array) in parallel, before proceeding to the next operation from the formula.

[5] This is, naturally, dependent on a particular language implementation, although it is 'automatic' due to the fact that the actual execution parallelisation has to be developed by the language implementor, and the "programmer" does not have to think at all about how much and which of her (or his) algorithm is executed in parallel, and on how many computing resources.

[6] I.e. 32 or 64 bit, as well as other bit sizes – 8 and 16 for microcontrollers, possibly 128 or other number of bits for special computing equipment, half, normal and double, as well as extended floating point formats etc.

[7] In Virtue there are logical rules which define which combinations of objects can be processed in parallel and which must be processed in serial. These rules are internal and are actually inherently semantic regarding the verbs applied to the objects, i.e. data.

semantically extremely rich Language, which is internally and externally "resizeable". What we mean by "resizeable" is that the amount of words which are understood (i.e. executed) by Virtue can be effortlessly expanded (externally, by defining new words), and that the internal knowledge of computing can be both expanded and shrunken. So for example an embedded version of Virtue would have a reduced amount of words as well as data types, and a Visualisation version would have to know specific words, whereas a Modelling version could have special words for often used algorithms. Furthermore, the "resizeability" of Virtue is automated by it's possibility to encompass a wider range of computers, depending on the scale of the problem to be solved.[8]

Actually *Virtue* is defined to be a *Virtual Executor*, a kind of system-ware/between-ware, a *Language* system taking the role of an *Inter-actor* between a *Computer System* and a *Human*.

Let's look at the particulars of what we want to say with this definition of Virtue's aims:

A. Computer System

A computer system in the above definition is actually any possible hardware or software device which understands the grammar of Virtue. These then could be just simple appliances, or the Computer System could be a grid of supercomputers or whatever else, like server farms, clusters etc.

As said, any computer system can be part of the Virtue linguistic space, as long as it understands the common standardised Virtue Data structure. For new knowledge, i.e. when a computer system with less features (e.g. a Virtual Executor which can process only integer numbers) has to solve problems it can not solve based on these, it will automatically invoke (if possible) another network based Virtual Executor which knows how to process this data and/or words. New words, if they can be processed in a Virtual Executor, would be just transferred as explanations, but data which can not be locally processed will always stay on the computer that was asked to process this part of the overall data structure. From the perspective of the Human user, the data would appear as if it would be locally processed and saved.

And finally, the most important feature of a Computer shall certainly be to know at the lowest, hardware or firmware level of execution, that which in Computer Science, Mathematics, Logic... we already have developed as long proven methods and algorithms, and which are regularly used.

B. Virtual Executor

The Virtual Executor is actually the basic (or very expanded) Virtue linguistic system on a Computer system. It is the Virtual Executor who enables Computer systems to understands the Virtue language – so it is the Virtual Executor who is the collocutor in the Human-Computer interaction.

An important feature of a Virtual Executor is that all instructions and definitions ever written in the Virtue language will suddenly start being 10% faster, without any additional effort (except the installation of a better version of Virtue) if the Virtual Executor is sped up by 10%. Contrary to this, no C written executable programme in the world will suddenly start being faster if a new compiler version is available, which produces a 10% faster code, without the specific effort of recompiling all the executables.

C. Human

The Human is the driving force of Human development. To be able to express all of his Human innovation capacity, and to intelligently behave in the present day information-oriented world, it is necessary for her to be able to constantly process all kinds of data and thoughts.

In this sense a Human presently more and more needs an assistant, which will perform operations like computations, data searching and problem solving, he needs in the intellectual, informational field - something we for a long time use in the physical environment – amplifiers. To move a huge rock, we use an amplifier of our own feeble movements. To solve a huge problem, we need to use an amplifier of our huge problem solving abilities. We actually need this which William Ross Ashby decades ago said, we need a "computer as an intelligence amplifier". And exactly this is what is the final aim of Virtue – interactive communication with the Human, algorithm to Virtue – answer to Human. Or question to Virtue – algorithm to Human.

Virtue is aimed towards allowing the Computer to become an intelligent human assistant.

D. Language

Long years of investigations into Human languages, all the worlds linguistic efforts show that the Language is the primary means of Human external world model forming, and his absolutely prime and most developed communication and understanding system.

Therefore, it is absolutely essential, to be able to actually fully integrate computers into the modern world Human society as intelligence amplifiers, to raise the level of Computer's language understanding towards the level of a full Human language, i.e. allowing the Human-Computer interaction language to be able to provide external world models for both of them.

V. DATA TYPES

Virtue has complex data structures (scalars, arrays of any dimension, array elements which are themselves structures of scalars or arrays…) and diverse data types (booleans; multidimensional fuzzy logicals; characters; numbers – real, complex, quaternion, octonion; addresses; functions; labels; files; or possible specific types like planetary ephemeris, pixels etc.).

[8] The present experimental implementation of Virtue uses only SMP. Work is in progress to encompass TCP/IP communication protocols with other computers running the Virtue system, by which all those processors/computers become a single large compute-assistant. The internal architecture of the experimental implementation is already prepared for multi-computer parallelism.

The inclusion of e.g. functions as data types, inter-mixable with any other data-type, as well as the form in which the programme itself is internally represented - in the exactly same way as data, enables Virtue to use, for example, variable functions whenever necessary, or to define function parallelism (e.g. by applying a vector of functions to a vector of values).

Files are also scalar data types, which means that they can be inside a multidimensional structured array (there is no theoretical limit to the number of open files in Virtue, the same as with the dimensionality of arrays). Each use of an argument which is a File (or contains a File in its structure) will read the data from the file, and provide the data defined by a "prototype record" supplied when the file was opened (i.e. to open a file a name and a prototype record is given) for the pending operation. This linguistically very simple and understandable approach allows for very complex data manipulation across a huge range of input data-sets and processing systems.

Virtue has an almost fully orthogonal approach towards data – any operation that has sense is usable. However, nonsensical operations (like e.g. square root of a character) are strictly disallowed. The execution framework of Virtue is very forgiving, yet strict, in imposing the processing rules for different combinations of data structures and data types.[9]

Further important features of Virtue include the complete lack of reserved words or symbols[10], synonyms, word dependant hierarchical contexts, continuations, and the possibility to use any data whatsoever as a name for something else.[11]

VI. Serial versus Parallel processing

The Virtue Language is designed to keep as many logically parallel algorithmic structures expressed in such a way that the parallelism is obvious. However, an algorithm is, per definitionem, a series of operations, possibly intermixed with decision making.

Virtue tries to keep consistency inside the two sets of operations, the:

- serial operations
- parallel operations

However, it should be noted, that the whole language can be implemented on a purely serial computer, but can also be implemented on parallel data processing equipment (multiprocessor, vector, cluster, grid...). The consistency of semantic distinction between parallel (i.e. parallelisable) and serial (non-parallelisable) instructions in an program on many levels does not a-priory dictate the implementation of Virtue. Although multidimensional logical values are parallel constructs, it would be counter-productive to parallelise a logical operation on 2, 4 or 8 individual values (or wouldn't it?). However operations on conformable multi-dimensional arrays are also parallel constructs, but it would be very advisable to implement them in parallel on several independent processing units.

It is important to understand that not all constructs may be safely executed in parallel, although applied to conformable arrays of enough elements. Although, for example, the application of functions on vectors/arrays may be a parallel operation, it must be serialized if the function has side effects.

Such a strict distinction is necessary, as there is no a-priory order in parallel operations, and there is no communication possible between strictly parallely executing operations.

Already during the analysis and/or during the processing Virtue will introduce into the internal Data Description (in this particular case of a function) a flag indicating if it may be executed in parallel, or the processing must be serialised. So, for example, by using input/output commands or global variables in a function which otherwise could be executed parallely, Virtue will automatically flag that condition, and serialise the execution. In the same time, these conditions in a functional sentence will prevent it from being memoised.

As another example of usage of the Data Description, an argument whose any element was produced by a random number generator (which indicates stochastic processing) applied to a normally memoisable function will prevent the memoisation of this function with that particular argument. This is necessary as otherwise simulations relying on random number generation would not properly work if memoisation is used. It will, naturally, normally work with all data which was not produced by manipulation of a random number.

It is obvious that with such internal structures the Human does not have to take care of the way his problems will be solved on a hardware level – serially, parallely, or in a concurrent combination of the two (for example more processors executing different functions on a vector, and using vector processors to execute parallel operations inside those functions). Virtue will execute SIMD, MIMD, MISD and SISD, even concurrently, as possibly in the above example. This makes the Virtue system widely adaptable to the underlying hardware, as any of the hardware architectures possibly used will excel in at least some aspects of execution.

VII. The Implementation of Virtue

The Virtue Language is fully defined in it's primary form.

The computer implementation is presently in Alpha 0.6 state, and parallelisation is implemented on SMP systems.

---

[9] For example, a dyadic (two argument) operation on arrays whose dimensions are not equal is prohibited, except if one of the argument array dimensions is a subset of the other argument's array dimensions, or one of the arguments is a scalar.

[10] Except the symbol '::', which reverts any redifined internal Virtue word or symbol to its original meaning, and the symbol '.', which always marks the end of a sentence.

[11] For example the sentence `'A vector of numbers from 1 to 1000' 1000 INTERVAL AT SET.` will save the given string under the name of a vector from 1 to 1000. To recall the string from memory, such a vector shall be given, as for example: `900 INTERVAL (901 902) CATENATE 98 INTERVAL 902 + CATENATE GET.` This sentence will then give the answer "`A vector of numbers from 1 to 1000`".

The experimental implementation is constantly parallely developed and tested on a very wide range of different computers, ranging from the mid-1980-ies Sun3 (16MHz/16MiB and 20MHz/24MiB) workstations up to modern day blades, with various operating systems and their generations (SunOS, Solaris, NetBSD, FreeBSD, Linux, Win), various processors (68k, MIPS, SPARC, PowerPC, AMD, Intel), on 32 and 64 bit processors in single-processor and SMD multi-processor, multi-threading and multi-core computers. Such a wide range of computers, both historically and speed-wise, for the experimental Virtue implementations allows for a development of a very easily adaptable system, and the behaviour of the old Sun3 systems shows that even on them the execution seems very fast for the amount of data which can be represented in the memory of those computers[12].

The internal speed measurements which the Virtue Executor has (and they are important for the future multi-computer implementations, to allow load balancing) have provided us with quite a lot of important data on the behaviour of different computer systems and different processors, so an investigation into the "speed of a computer" is presently being performed, with some results to be presented soon.

VIII. Conclusion

Virtue is a pliable language, shrinkable and extensible, but generically compatible as a Language on all possible levels of implementation. Therefore it could be implemented in hardware, as a specific processor, it could be small and embedded, but it could automatically grow towards huge systems and big data. The idea of a Language even in the inter-computer communication, allows for Virtual Executors which do not recognise a word the user used, to consult (if on network) other known, but probably bigger Virtual Executors about the meaning of the word, and/or help in the execution of a task.

The final aim of this long-term effort is to provide a "human assistant", an intelligent, selflearning and selforganising "assistant" in this more and more complex life environment.

From the level of being a chip, or a programme in a chip, able to monitor, coordinate and regulate processes in complicated systems, up to the level of data mining in distributed environments, Virtue is aimed to provide a seamless integration.

As a mathematical processor, as a Grid Library Application, as a Modelling Tool, Visualisator or a data/calculation preprocessor...

References:

[1] John Backus "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Communications of the ACM, Vol 21, No. 8, pp. 613-641, August 1978., http://delivery.acm.org/10.1145/360000/359579/a1977-backus.pdf?ip=93.136.170.6&id=359579&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&CFID=415707851&CFTOKEN=76568762&__acm__=1393661168_49edb86547c11a2260a8d1ad8731f4b4

[2] Barendregt, H.P. and Eekelen, M.C.J.D. and Hartel, P.H. and Hertzberger, L.O. and Plasmeijer, M.J. and Vree, W.G. (1987): "The Dutch parallel reduction machine project". Future Generation Computer Systems, 3 (4). pp. 261-270. ISSN 0167-739X, http://doc.utwente.nl/55757/1/dutch.pdf

[3] Borna Bebek, Zorislav Šojat: "Optimal Distribution of Organisational Resources", Diplomatic Academy Year–Book, Edition: Central European Initiative – International Conference "Diplomacy for the twenty-first century: knowledge management", Vol. 2/2000, No. 1, HR: Ministarstvo vanjskih poslova i europskih integracija; 2000

[4] EGI Glossary; European Grid Initiative WIKI, https://wiki.egi.eu/wiki/Glossary_V1, 22/2/2014, 21:56

[5] "Gannett's Law", http://0x07bell.net/WWWMASTER/CrayWWWStuff/Glaw.html

[6] Tomislav Grubesa, Goran Topic, Valentin Vidic, Zorislav Sojat, Karolj Skala: "Towards the Design and Application of Grid Portals", Hypermedia and Grid Systems, MIPRO, Opatija, Croatia; 06/2005

[7] Roland N Ibbett & Nigel P Topham (1996): "HIGH PERFORMANCE COMPUTER ARCHITECTURES - A Historical Perspective", http://homepages.inf.ed.ac.uk/rni/comp-arch/index.html

[8] Sinisa Marin, Robert J. Thorpe, Zorislav Sojat: "A Modular Robot Programming System", 1989; London, GB: RDP Technology.

[9] Sinisa Marin, Mihajlo Ristic, Zorislav Sojat: "An Implementation of a Novel Method for Concurrent Process Control in Robot Programming", Third International Symposium on Robotics and Manufacturing: Research, Education and Application, ISRAM '90, Burnaby, BC/CA; 1990

[10] John Owens "Data Level Parallelism (2)" EEC 171 Parallel Architectures, UC Davis, http://www.nvidia.com/content/cudazone/cudau/courses/ucdavis/lectures/dlp2.pdf 1/3/2014, 6:13

[11] Nikola Pavkovic, Karolj Skala, Valentin Vidic, Zorislav Sojat: "Bioinformatics Application Oriented IT Deployment Model",

[12] The classical double recursive algorithm for calculating the n-th number of a Fibonacci series could be expressed in Virtue for example thus: `MONADIC DUPLICATE 3 < @%a IF JUMP DUPLICATE 1 SUBTRACT Fibonacci! SWAP 2 SUBTRACT Fibonacci! ADD RETURN %a DISCARD 1; @Fibonacci SET.` The sentence, translated into English, says: "Check if the requested Fibonacci series element number is less then 3, if not subtract 1 from this number and calculate that Fibonacci (series element), then subtract 2 from the same number and calculate the Fibonacci, otherwise (if the number was < 3) jump to conclusion, forget that number and just return 1, then remember this phrase, which expects one object, as the word 'Fibonacci').

The calculation of all the first 1500 Fibonacci series numbers (slightly over the limit of IEEE double FP) on a 1987. Sun3/60 (SunOS 4.1.1, original Sun cc compiler, MC68020+MC68881, 20MHz, 24MiB 32-bit RAM - ~3MIPS, 192 kFLOPS) using this algorithm with the memoisation word RESULT - that is inputing the sentence `1500 INTERVAL MONADIC Fibonacci RESULT; EACH.` (in English: "Make an interval vector from 1 to 1500, then for each element of the vector calculate the Fibonacci, but first check if you already remembered the result.") - the vector of those first 1500 (!) Fibonacci numbers will be produced after just 27.827s, whereas the next time the same sentence is entered, the results will come in just 4.461s. Such speed is fully acceptable for normal work! Modern day common computers are several hundred times faster (not several thousand times, as would be expected from the MIPS/GIPS relationship. Just one example: a recently tested 1.67GHz Intel Atom in a reputable notebook with 2GiB 667.0 MHz DDR2 SDRAM memory, on Windows 7/32, Microsoft Visual Studio 2010 Express C compiler, performed the above operations in 0.21s (first time), and in 0.045s (afterwards), a speedup compared to Sun-3 of only around 100 times, despite the high sounding numbers. The reason is the more and more intensively felt "von Neumann bottleneck" - the memory vs. processor speed and channel constraints!)

Parallel Numerics, Theory and Application; Salzburg : University of Salzburg Austria(2005):217-222; 2005

[12] Karolj Skala, Zorislav Sojat: "Towards a Grid Applicable Parallel Architecture Machine.", Computational Science - ICCS 2004, 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part III; 2004

[13] K. Skala, Z. Sojat: "Image Programming for Scientific Visualization by Cluster Computing", Autonomic and Autonomous Systems and International Conference on Networking and Services, ICAS-ICNS; 2005;

[14] Karolj Skala, Nikola Pavkovic, Zorislav Sojat: "Scientific Visualization by Cluster Computing", 8th COST 276 Workshop, Trondheim, Norway; 05/2005

[15] Zorislav Šojat: "Jezik kao samoorganizirajući stroj", Interbiro '82, Zagreb, Yugoslavia; 1982

[16] Zorislav Sojat: "Operating System Based on Device Distributed Intelligence", 1st Orwellian Symposium, Baden Baden, Germany; 1984

[17] Zorislav Šojat: "Ustrojenje jezika APL", 1989; Zagreb, YU: Filozofski fakultet Sveučilišta u Zagrebu

[18] Zorislav Sojat, Sinisa Marin: "ISOCOM 20 Filter System User Documentation: Flow Through Filter Language and Graphics Organisation Language", 1992; BTS, Purley, GB.

[19] Zorislav Šojat: "Nanoračunarstvo i prirodno distribuirani paralelizam", Glasilo Instituta Ruđer Bošković. 3(7/8):20-22.; 2002

[20] Zorislav Sojat, Karolj Skala: "Multiple Programme Single Data Stream Approach to Grid Programming", Hypermedia and Grid Systems, Opatija, Croatia; 2004

[21] Guy Steele, Jan-Willem Maessen: "Fortress Programming Language Tutorial", Sun Microsystems Laboratories, 2006; http://stephane.ducasse.free.fr/Teaching/CoursAnnecy/0506-Master/ForPresentations/Fortress-PLDITutorialSlides9Jun2006.pdf

[22] Jack J. Woehr: "A Conversation with Guy Steele Jr.", Dr. Dobbs, April 1, 2005, http://www.drdobbs.com/jvm/a-conversation-with-guy-steele-jr/184406029