Fraunhofer GPI-Space – Parallel Programming Development Software

- Dr. Mirko Rahn –

1. Introduction

Domain users are not expert at programming modern high performance computing systems. This statement implies a great many related things: in terms of the energy costs alone, today's computing systems can cost millions of euros per year. The energy costs in fact, have become a significant share of the total overall cost of a computer. In return, these computers provide incredible possibilities in the form of available computational operations. By the end of this decade, computers will exist that can perform 10^{18} (one followed by eighteen zeros) calculations per second, that is a quintillion or, a million times a billion operations in a single second.

This huge number describes the theoretical maximal performance, and that is precisely where the problems begin for the domain users. Many fields of science could benefit from simulations that can only be carried out by means of such a powerful capacity, but *the* computer does not exist. Today's systems instead consist of thousands of units, each containing thousands of processing cores, linked together via networks. Added to these are special subunits that communicate with the other components over various communication channels.

Each individual processing core must be carefully programmed: for example, the cores can perform more than one operation per cycle, but only if the data to be processed is presented in the proper sequence. A failure to accomplish this means that only 12 percent and, for the newer cores, only 6 percent of the available capacity is used - with no reduction to the cost. The situation with the specialized components is even more complicated and the potential loss in efficiency if not optimally programmed is unacceptably high.

The size of the computers enables the processing of huge data volumes, although the data still has to be stored somewhere. This necessitates the connection of large, complex file systems - separate high performance systems with their own specific characteristics. The costs of the data transfer between the computing and file systems are substantially higher that the costs of the data transfer within the computer itself and the bandwidths used are much smaller. For these reasons, applications require detailed planning regarding when and what data is to be exchanged between the computing and file systems. Otherwise, the computer will be awaiting a data transfer and no productive processing is being performed.

The actual transfer of data into or out of a file system is just one aspect of data transport. Each unit contains a vast hierarchy of memories that are becoming ever smaller and faster, semiconductor drives, core memories, various caches, and registers within the processors. Data transport is substantially more expensive (in the form of power required) between the levels of the hierarchy and it takes much longer than if performed at just one level of the hierarchy. It may even be appropriate to do a local re-computation of the data instead of a data transfer.

That is not all. Even with very reliable and durable components, such a large number of components can lead to a high probability of a partial or even a total failure of some individual components. (In Germany, there are approximately 1500 lottery millionaires, although the chances of my bet winning are diminishingly remote.) The failure of an application on top of this cannot be allowed. The consequences would be the loss of all previous calculations. This sounds like something you could recover from, but it is not really because restarting the application will in all likelihood result once again in a failure. In other words, with the lack of fault tolerance in the components, no results can be achieved.

The question that begs asking is: How can the domain user make efficient use of these machines?

One answer stands in direct contrast to our introductory premise: The domain users must become expert in programming modern, high performance systems. For many years, this was the standard response and great effort was expended in training the domain users. New generations of hardware arrived accompanied by adjustments and further software development. Of course, parallel software has been around for a long time, but the methods and tools (mainly MPI and OpenMP) have their roots in the 1990s and were never envisioned for use in computers with more than 1000 processors. Technological possibilities are now changing so fast and today's computers are so complex that this approach is no longer viable.

The possibilities to extend and improve the existing methods and tools have been to a great extent exhausted and the realization has spread among leading researchers that a radical new approach is necessary. [SPPEXA 2011]

Consequently, we must rephrase the question: What aid can be provided to domain users to enable them to build effective, scalable, portable, resource-efficient, and resistant applications?

This attitude is the starting point for much thought and effort in Europe and around the world. [EESI 2011, Dongarra 2011] Many answers unfortunately remain trapped at the stage of improving and expanding. [Müller 2009] In this category, for example, are the debugging programs optimized to handle hundreds of thousands of cores. Using the existing platforms, the debugging of individual cores produces a detailed process log. It is no easy task to prevent the volume of such a log from becoming unmanageable, without the possibility of losing some important information. It is even more difficult to represent the results and there are still very few tools for automatic analysis.

In more recent research, several other approaches are beginning to take form in which applications are being split into smaller parts. For example, Zuckermann, Knauerhase, and Gao divide applications into small functional units and detail the relationships between the functional units. [Zuckermann 2011] They have demonstrated that such splitting has many advantages: The division of labor among existing resources is facilitated, the data transfer can be controlled at higher levels, parallelism is easily extracted and put to use, and finally, simple optimizations can be implemented regarding the capacity and/or the power.

The idea of splitting a program into its functional units and dependencies is not new. In 1992, David Gelernter spoke in general terms about splitting coordination and computation functions. [Gelernter 1992] Computation in this context meant an elementary step that produced a set of output data from a set of input data. This kind of activity worked regardless of whether it was part of a parallel application or not. An activity has no information about where the input data originates, whether it is

part of a larger volume of input, or what will happen with the output data in the future. These are issues to be answered at a different level, the coordination level. It is important at that level to know where the data is stored, what operations to execute with the data, and how data can be separated or integrated.

The key point made by Gelernter is that for the programming at both of these levels, coordination and computation, different languages can and should be used. He proposed a general language for the coordination level, which is orthogonal to the languages for the computation level and shows how such a language not only allows the design of portable programs for heterogeneous computers, but also that these programs can be automatically run in parallel as well as being efficient and flexible at the same time.

Other approaches aim for the use of completely new computer models. In the 1990s, Jack Dennis grappled intensively with how software could be designed for parallel computing. [Dennis 1997] He succeeded in constructing a model that is similar to a functional language and recently, his ideas have been the subject of renewed attention. [Dennis 2011] The well-known advantages of functional languages are their modularity and the direct correspondence to algebraic structures, which qualifies these tools as both powerful and user friendly. [Backus 1978, Hughes 1989]

This is the point at which the innovative software from Fraunhofer ITWM enters the story: The basic concepts are the splitting of the coordination and computation, and the use of a functional programming language at the coordination level. At the computational level, no special language is required for the software, primarily to ensure the interoperability with existing components.

The following sections introduce the major components of Fraunhofer GPI-Space software and two examples are used to illustrate how applications for modern high performance systems may be developed in the future.

2. FP3

The basic ideas introduced above, splitting of coordination and computation and use of a functional programming language at the level of coordination, leave room for questions: What functional language should be used? How is this language used for programming? How can applications programmed in this way be executed? How do such applications communicate with each other and with the external world? How do the different functional units communicate within the application?

Fraunhofer ITWM has addressed these questions and, together with many years of application design experience in the industrial sector, has bundled the answers in the development of Fraunhofer GPI-Space software.

2.1 Petri nets as the coordination language

Programming at the coordination level should represent dependencies between the simple functional units. Whereas the elementary functional units either already exist or will be created by the experts, the coordination should be programmed by the domain users. It is the domain user after all, who is best informed about the specific relationships between the elementary functional units. It is also the domain user that wants to try out the new combinations, the different methods, or to integrate the additional steps for pre- or post-processing of the data.

Against this background, the tool of choice is a simple and specific graphic interface. This can reduce the programming complexity so that the domain user is not confronted with unfamiliar terms, but rather can concentrate fully on solving the problems within the domain, which is the real challenge. Hierarchical designs ought to be possible. The use of graphically created applications should also be possible in other contexts. At the same time, specific graphic notations should have a well-defined purpose, independent of the domain.

Such requirements frequently occur when the end user enjoys programming freedom when using the system. Specific solutions are often proposed, for example, in image or signal processing. Exactly as advocated by David Gelernter, this is where a general language should be used, i.e., Petri nets.

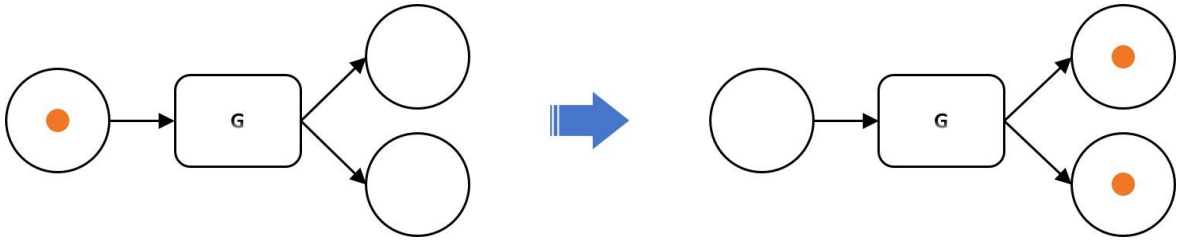
In his dissertation, Carl Adam Petri examined the possibilities of adding resources to running calculations on an as-needed basis, without interrupting the process and without informing any of the existing resources about the new resources. He discovered an elegant solution that only connected resources locally with other resources. Nevertheless, all resources must be able to operate autonomously, i.e., asynchronously. Petri just needed an opportunity to describe the asynchronous and distributed systems and soon thereafter, the development started on the nets that bear his name today. [Petri 1962]

What makes the Petri nets interesting for us besides their simple graphic and hierarchical structure is their local nature (i.e., no global state), their concurrency, (i.e., there is no total assignment of events, just data dependencies) and their reversibility (i.e., the causal can be determined from one result). Incidentally, these are all properties that Petri intentionally borrowed from Physics for use in Computer Science. [Brauer 2006] In addition, Petri nets are often used in the engineering disciplines as a modeling tool as well and the theory behind them is very well researched and understood.

The advantages of Petri nets as a mathematical modeling language are formulated very well by van der Aalst: Petri nets have precise execution semantics that assign specific meanings to the net, serve as the basis for an agreement, are independent of the tools used, and are what enable process analysis and solutions. Furthermore, because Petri nets are not based on events but rather on state transitions, it is possible to differentiate between activation and execution of an elementary functional unit. In particular, interruption and restart of the applications are easy, which is a fundamental condition for fault tolerance to hardware failure. Lastly, van der Aalst notes the availability of mature analysis techniques that besides proving the correctness, also allow performance predictions. [vdAalst 1996]

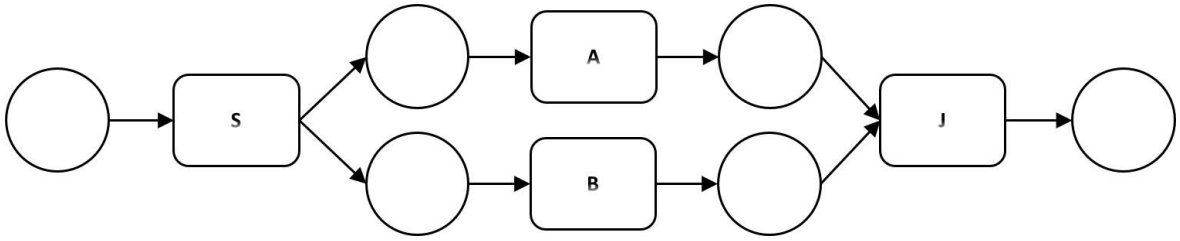
Let's see what a simple Petri nets looks like:

There are boxes for transitions and circles for places. Transitions stand for the elementary functional units and places are the placeholders for the small, black, so called tokens that represent the data. Places and transitions are connected by directional arrows. A transition can fire when all of its input places contain tokens. When fired, a token from the input place is removed and a token is produced at each output place. The illustration shows the before and after situations for Transition 'f'.



Transition 'h' has two output places, i.e., it produces two tokens when fired.

A kind of automatic parallelism is present, for example, when more than one token is at the input place for Transition 'g' . In this illustration, three incarnations of 'g' can be executed simultaneously.



A different kind of concurrency exists between Transitions 'a' and 'b': If there is a token at the input place for Transition 's', a token will be produced at the input place for 'a' and 'b', thus, each of these transitions can fire in parallel. Transition 'j' synchronizes the Transitions 'a' and 'b: It can only fire when both 'a' as well as 'b' have been fired.

In the Fraunhofer GPI-Space software, somewhat more complex Petri nets are used. First, the tokens do not have to be black, but may be of any color. The colors denote certain types. For example, it makes a difference if a token represents a number or a text. Each place can only contain tokens of a certain type and the transitions are also restricted to colored inputs and outputs. Furthermore, other Petri nets may be contained within a transition.

With these expansions, the Petri nets used in Fraunhofer GPI-Space software have become a complete, formal, standard functional programming language. After construction, Petri nets can be used in an unlimited number of different contexts. They are locally specified. The nets are expanded with a small built-in language for expressions, which allows simple functional units to be integrated directly in the Petri nets.

The description of the Petri nets within the framework of the Fraunhofer GPI-Space software required the development of an XML schema. The use of XML ensures portability, recourse to advanced external tools to validate the syntax, and the availability of powerful editors. A special feature is the mechanism for polymorphic programming: Similar to the C++ templates, generic Petri nets can be formulated and while being strictly defined, still work with many different types. This is an area to be specified in a later specialization step.

2.2 Distributed runtime environment

Up to this point, we have only discussed the design of applications in Fraunhofer GPI-Space. A complete programming environment requires much more: There must be an instantiation that executes the programs. This instance is referred to as a runtime environment and is the interface between application and specific computer.

In the case of the Fraunhofer development environment, the application runs in a high performance system. This is a distributed computer, constructed from many components linked together by a network. Correspondingly, the runtime environment of GPI-Space is also distributed.

The main player in the runtime environment is a component that we call an "agent." The agent is a program that contacts other agents either locally or via a network. In the process, asymmetrical relationships develop: One of the agents plays the role of "worker", while the other is the "master". The main task for the master is to distribute the work among his workers. The workers perform the work assigned to them or forward it, in turn, to their workers.

In this way, a network of masters and workers is formed, where masters can also be workers and so on and in Fraunhofer GPI-Space, it can can take on arbitrary structures. This is the prerequisite for working with a number of heterogeneous computers. For example, if we have a computer in which some of the components are more powerful than others, it is reasonable to install more "workers" on these components than on the less powerful components. We can also define how many hierarchical levels the runtime environment will have. Depending on the job, it may be appropriate to work with more or fewer hierarchies.

The structure of the runtime environment is not only flexible, it is also dynamic. While an application is being executed in the runtime environment, the structure can change: New agents can be integrated

in the runtime environment and existing agents can be removed from the runtime environment. This is a key condition of fault tolerance, where faults are manifested as a result of changes to the structure of the computer. In order for a runtime environment to be completely fault tolerant, an agent can store its internal state and return at some later time to continue the work at this point.

The work that is performed by the workers is actually an elementary functional unit presented in the form of an executable program. This program executes separately to prevent possible failures from corrupting the runtime environment. The runtime environment, depending on the type of functional units and the type of failure, can decide whether the functional unit will be repeated, perhaps by some other component.

The user communication with the runtime environment is carried out by a component that we refer to as the "orchestrator." The orchestrator is a program that accepts the jobs and enables status queries and the interruption and deletion of jobs. The orchestrator also has fault tolerance and can be executed remotely from the runtime environment, for example, on the user's laptop.

The job the orchestrator accepts is not a direct XML representation of the Petri net, rather it requires a specific internal format. This situation is comparable with an operating system that does not process the source text of an application directly, rather works with a specific binary format.

Analogous to this comparison, there is another component of Fraunhofer GPI-Space called the compiler that generates the internal representation of the runtime environment from the XML representation of the Petri net. The compiler performs not only validation of the semantics, it also provides the routines for the plausibility check and routines for optimizing the given net. Besides the internal format of the runtime environment, the compiler generates all of the interfaces needed for the elementary functional units to access the data represented by the tokens in the Petri net.

The orchestrator sends the accepted job to the runtime environment where it is analyzed by an agent. The agent extracts all executable activities from the Petri net and generates new jobs that it either forwards to its worker or processes by itself. This process ends, perhaps, only after multiple steps because the transitions in the Petri net may themselves contain complete Petri nets. When the job is finally completed, the agent inserts the results in the Petri net and the process begins again from the start as long as there are no new activities to be extracted from the net. The network is then processed completely and the job result is provided in the orchestrator for the user.

2.3 Virtual Memory

Applications are described as a Petri net and executed within the distributed runtime environment. The elementary functional units are extracted from the Petri net and, as mentioned above, as they depend only on their input data, are transformed to output data. Under normal conditions, the output data of one elementary functional unit are the input data for another elementary functional unit. The functional units communicate. This means Fraunhofer GPI-Space must provide a mechanism that permits the elementary functional units to exchange data.

The data exchange is not only important between functional units, i.e., within a single application, but it must also be possible to communicate between various applications within Fraunhofer GPI-Space and, between Fraunhofer GPI-Space applications and the external world.

All of these types of communication are managed in a virtual memory. For our purposes, this is an abstract layer, seen as a single large memory block by the applications. Fraunhofer GPI-Space provides methods that facilitate areas of a specific size to be reserved within the virtual memory, so that data can be written or read in these reserved areas and so reserved areas can be reopened. These methods are available not only to the applications within GPI-Space, but can also be used by the external world.

All transfers to or from the virtual memory are asynchronous, which means they are processed in the background while the application is processing other data. Furthermore, the global state of the virtual memory is separated from local application errors: Applications cannot access the virtual memory without selection, but only via the methods supplied by GPI-Space.

This abstract layer can be implemented in several ways. One way is to use a parallel file system as an underlying real memory. The virtual memory for Fraunhofer GPI-Space is implemented on the basis of a Global Memory Layer as described in section . As explained there, it is possible to address the entire main memory of the distributed computer as a single, large memory block. Transfers are asynchronous, performed with wide band width and low latency.

2.4 Conclusion

Fraunhofer GPI-Space software consists of three main layers: The virtual memory as the communications layer, the distributed runtime environment as the execution layer, and the closely linked Petri net interpreter as the control layer.

We have discussed how each layer is designed and why the decisions were made. Figure 1 is an attempt to capture the basic structure (NRE means "node runtime environment"). The three layers of Fraunhofer GPI-Space are seamlessly integrated, although they are also available as individual modules for other applications.

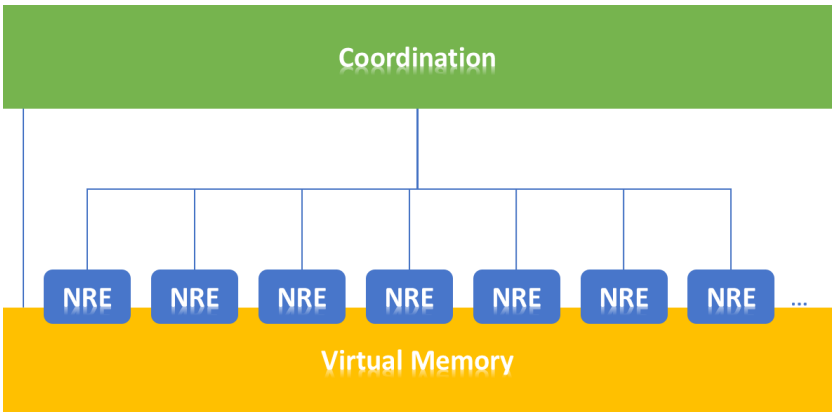


Figure 1: FP3Architecture

This paper has praised the graphic nature of Petri nets, but has gone into some detail only on the XML representation. Of course, we are also working on graphic editors that are compatible with Fraunhofer GPI-Space. Figure 2 shows the user interface of a prototype for the processing of seismic data: The area at the center is for the design of the application, at the right side is a library of prefabricated modules, and on the left is a structural view of the application.

Overall, Fraunhofer GPI-Space is the result of intensive efforts concerning the issue of programmability and the high performance systems of the future. Many important questions have been answered and the software has proven to be outstanding when used in a number of projects at the Competence Center High Performance Computing and Visualization at Fraunhofer ITWM.

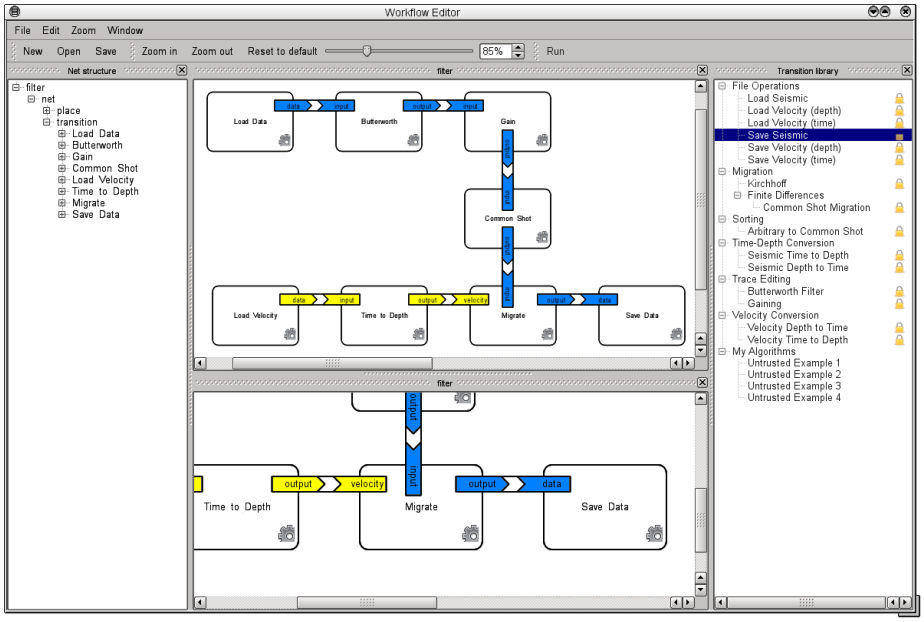


Figure 2: Prototype of a graphical user interface

The two examples presented below illustrate how applications for complex high performance systems can be developed with the aid of Fraunhofer GPI-Space.

3. Example applications

The two examples presented here illustrate how to program complex high performance systems with the aid of Fraunhofer GPI-Space. Both of the selected applications were developed at Fraunhofer ITWM using GPI-Space software. These are just two of the many applications developed by ITWM for customers throughout industry. These include applications controlled with the aid of a specialized graphic interface as well as applications designed to make older, existing programs more fault tolerant.

3.1 Parallel SU

The SU Package (Seismic Un*x) is more than a 20-year old collection of programs for processing seismic data. [SU] Many geophysicists around the world use programs contained in this collection and SU has become a quasi-standard. However, there is a problem: The programs were written at a time when parallel data processing was not common, they are simply sequential programs. Furthermore, over time the volume of seismic data has grown dramatically, numerous terabytes is the normal state. This means the programs from the collection are nearly useless, because processing realistic data sets simply takes too much time.

Fraunhofer GPI-Space makes it quite easy to use parallel processing for a relevant section of the program from the collection. This refers to sections that process data piece by piece, without any complicated dependencies between the pieces. The respective Petri net has a very simple design:

The knowledge about the data is inserted in the transitions LOAD and WRITE. For each independent piece of data in the input file, the transition LOAD produces a token at the input place for the transition WORK, which then is able to work the pieces in parallel. The transition WRITE consequently writes the results in parallel in the output file. The transitions LOAD and WRITE come from a library of seismic transitions maintained at Fraunhofer ITWM. The transition WORK calls up an existing program internally from the SU collection.

Despite a relatively simple structure, this is a very efficient application: The processing of a 121 gigabyte data set with the sufrac program lasts 2751 seconds. The processing time for the above program is only 228 seconds when 16 CPUs are used and only 127 seconds if 32 processing units are used. This represents an improvement from more than 45 minutes down to just over 2 minutes.

This example illustrates how simple tasks can be solved quickly with the support of Fraunhofer GPI-Space software. The result is an efficient parallel program.

3.2 Kirchhoff Depth Migration

The Kirchhoff Depth Migration is a basic method for reconstructing the subsurface on the basis of recorded seismic signal data. Other methods exist that provide a better image quality, but the Kirchhoff Migration continues to be used frequently because of its speed.

From an implementation perspective, the Kirchhoff Migration consists of determining the contributions of all collected time series on all surveyed sections of a subsurface. In more abstract terms, it is the use of a function 'f' applied to all elements of the cross products of two values X and Y.

In traditional programming language, this challenge is solved through the use of a nested contribution loop:

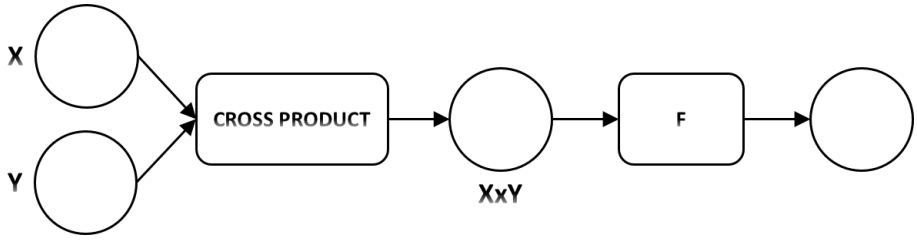
```
foreach x in X: foreach y in Y: f (x,y)
```

But just a moment, there is a second solution. The inner and outer loops can be exchanged:

```
foreach y in Y: foreach x in X: f (x,y)
```

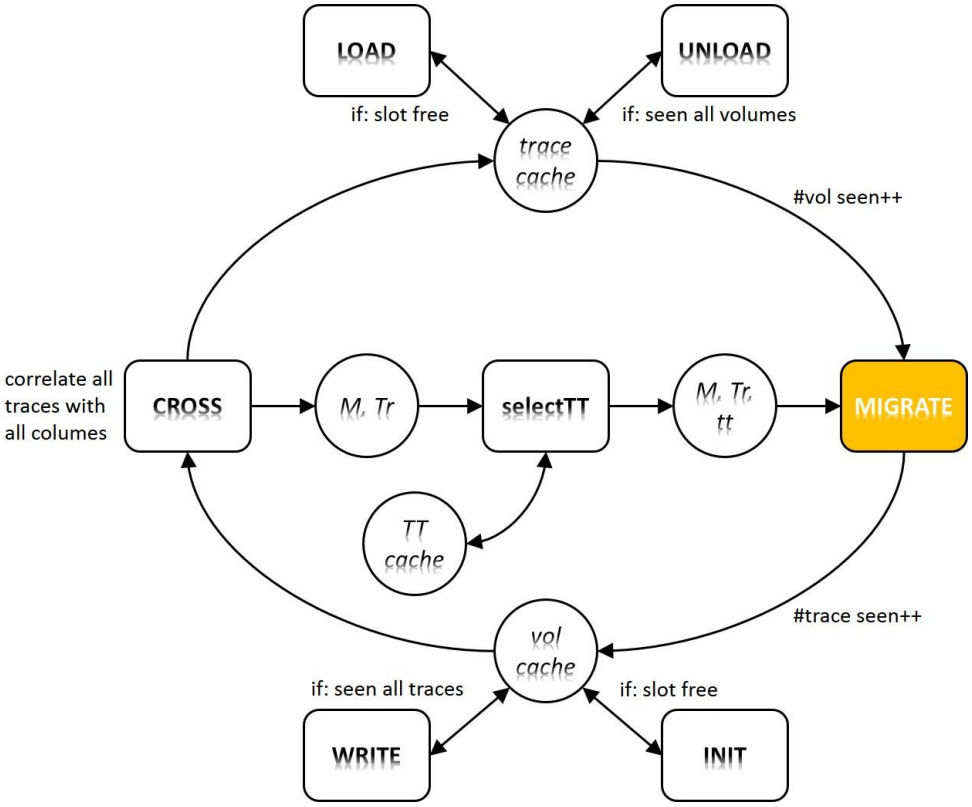
At first glance, the difference between the two solutions may appear marginal, but what if X and Y have different sizes? It may be the case, for example, that the value X fits completely in the memory, but the value Y does not. In the second solution, X will be read exactly one time along with one of the y values in Y. Similarly, in the first solution, because Y cannot be stored in the memory, for every x value in X, Y is read again.

An optimal solution can always be found even for complicated dependencies, but the decision for one kind over the other is made before the detailed inputs are known and, as a result, will generally never be optimal. It would be so much nicer to formulate the computation of the cross product in the abstract, like this:



This expression includes both of the above solutions. Furthermore, it describes all interpolations between these solutions. It only describes what needs to be done and not how to do it. The software runs automatically, driven solely on data dependencies and selects the perfect input and processing sequence appropriate for the computer. Especially if there are sufficient resources, the function 'f' can be used at the same time on different elements of the cross product.

Now back to the Kirchhoff Migration: the computation of the cross product is at the core and the Petri net looks like this:



On the left side, we see the abstract cross product of the traces and sub volumes. What is interesting is the yellow transition. It is the only transition in this Petri net that implements geophysical knowledge. All of the other transitions are generic and could be used directly for other processes with the same structure.

This application also demonstrates outstanding performance, as illustrated by a small example. Processing an input size of approximately 1 gigabyte by a manually programmed version took 403 seconds. The Fraunhofer GPI-Space version required only 170 seconds. The reason is the superior handling of input data induced dependencies. Of course, this could also be programmed manually, but the effort would be unacceptably high.

Scalability is also outstanding. A somewhat larger example using approximately 47 gigabytes of input required 2200 seconds with 32 CPUs and 1048 seconds with 64 CPUs. That is less than half of the processing time, an effect that is sometimes produced when individual work packages get smaller and are more suitable to the memory hierarchy.

In summary, these examples aptly show the possibilities to be realized with an abstract formulation of the task. The detail problem should lighten the load of the runtime environment and it has the right information to do that, namely the dependencies.

4. Conclusion

Domain users are not expert at programming modern high performance systems. This is a fact that has too often been ignored in the past. Many applications, some quite efficient, have been created by domain users, computer centers continue to train, and developers work to improve and expand the tools. This approach no longer works. The main reason is the growing complexity of computers. This view is gaining a broader consensus among leading researchers and the search for alternatives is heating up.

The foundations for more efficient and simpler programming of more complex systems were laid down a long time ago. Names like Carl Adam Petri and David Gelernter represent an integrated approach to be used in managing such computers.

On the basis of these foundations and its broad experience in the design of industrial applications, Fraunhofer ITWM has created Fraunhofer GPI-Space as a design and development software. Its major components, the virtual memory, distributed operating environments, and the Petri net interpreter, have been seamlessly integrated in the software - while the individual modules are also available to other applications.

At ITWM, Fraunhofer GPI-Space software has proven itself outstanding for various customer projects in the industrial sector. The efficient execution of complex applications on complex computers is possible thanks to the abstract formulation of the software. For example, applications created by the software have demonstrated better performance than manually programmed versions. Other applications have integrated existing programs into the software, making the integrated application highly tolerant to hardware faults. With improved graphic interfaces, library supplements, and new domains, the future looks bright for Fraunhofer GPI-Space.

5. References

[v.d.Aalst 1996] W.M.P. van der Aalst. Three Good reasons for Using a Petri-net-based Workflow Management System. Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), 1996.

[Backus 1978] J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. 1977 Turing Award Lecture, Comm. ACM 21(8), 1978.

[Brauer 2006] W. Brauer, W. Reisig. Carl Adam Petri und die ,,Petrinetze". Informatik-Spektrum 29(5), 2006.

[Dennis 1997] J. B. Dennis. A Parallel Program Execution Model Supporting Modular Software Construction. Proceedings of the Conference on Massively Parallel Programming Models, 1997.

[Dennis 2011] J. B. Dennis, G. R. Gao, X. X. Meng. Experiments with the Fresh Breeze tree-based memory model. Computer Science - R&D, 2011.

[Dongarra 2011] J. Dongarra, P. Beckmann. The international Exascale Software Roadmap. International Journal of High Performance Computer Applications. Volume 25(1), 2011.

[EESI 2011] European Exascale Software Initiative, report at http://www.eesi-project.eu

[Gelernter 1992] D. Gelernter, N. Carriero. Coordination languages and their significance. Commun. ACM 35(2), 1992.

[Hughes 1989] J. Hughes. Why Functional Programming Matters. Computer Journal 32(2), 1989.

[Müller 2009] M.S. Müller, M. Resch, A. Schulz, W. Nagel. Tools for High Performance Computing. Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, 2009.

[Petri 1962] C. A. Petri. Kommunikation mit Automaten. Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[SPPEXA 2011] Software for Exascale Computing. Proposal to the German Research Foundation to establish a Priority Program in the multidisciplinary field of High Performance Computing, 2011.

[SU] http://www.cwp.mines.edu/cwpcodes/index.html

[Zuckermann 2011] S. Zuckerman, J. Suetterlein, R. Knauerhase, G. R. Gao. Using a "codelet" program execution model for exascale machines: position paper. Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT '11, San Jose, California, 2011.