Comparing MultiCore, ManyCore, and DataFlow SuperComputers: Acceleration, Power, and Size

Sasa Stojanovic(1), Veljko Milutinovic(1), Dragan Bojic(1), Miroslav Bojovic(1), Oliver Pell(2), Michael J. Flynn(3), Oskar Mencer(4)

(1) University of Belgrade, (2) Maxeler Research London, (3) Stanford University, (4) Imperial College London

1. Abstract

This paper presents the essence of the newly emerging dataflow paradigm to computing, using the realities of the most recent implementations based on modern FPGA components. It sheds light on issues like speedup (as high as about 20 or more for a number of important applications), size reduction (about 20 for current implementations), and reduction of power dissipation (monthly electricity bills are about 20 times lower) – all that for the same production cost, compared to the existing computing technologies. Programming paradigm, debugging effort, and compile time are also covered. The analysis is based on the analytical, empirical, and anegdotical approaches.

2. Introduction

While MultiCore CPUs and GPUs are well known in supercomputing applications, DataFlow computing is only recently making inroads in this domain. DataFlow computing was popularized by a number of researchers in the 1980's, especially Professors Dennis and Arvind of MIT. In the dataflow paradigm an application is considered as a dataflow graph of the executable actions; as soon as the operands for an action are valid the action is executed and the result is forwarded to the next action in the graph. Creating a generalized interconnection among the action nodes proved to be a significant limitation to dataflow realizations in the 1980s. Over the past few years the extraordinary density achieved by FPGAs allowed FPGA based emulations of the DataFlow machines executing the dataflow graph corresponding to the application. Recent successful dataflow implementations are static, synchronous with an emphasis on data streaming, and applications in physics, banking, datamining, and education [1, 2]. Further applications can be opened if combined with cloud technology [3].

For purposes of comparison a possible classification of supercomputer systems is given in Figure 1. The four major branches of the classification imply the following architectures: MultiCore, ManyCore, FineGrainDataFlow, and CoarseGrainDataFlow supercomputers.

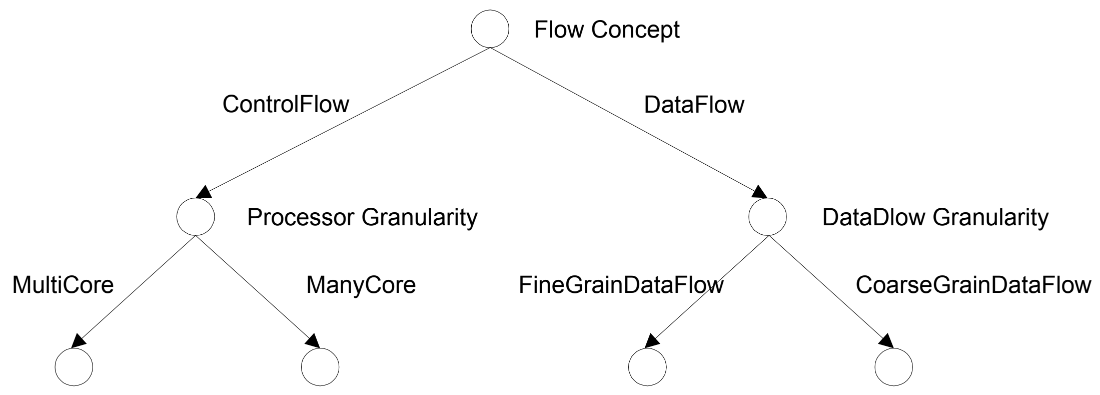

Figure 1: A classification of supercomputer architectures.

Programming of the first two architectural types follows the classical programming paradigm and the achieved speedups are well described in the open literature [4, 5, 6]. DataFlow machines use a different execution model and therefore it is generally efficient to utilize a different programming paradigm. Two issues are critical: the actual programming methodology used to generate the maximal speedup (for these applications, there is a large difference between what an experienced programmer achieves, and what an inexperienced one can achieve), and the overall characteristic of applications suitable for dataflow speedup.

2.1. Programming a dataflow machine

The programming models for MultiCore/ManyCore and DataFlow are different. The outcome of this difference is that in the first case one writes an application program in C, or another appropriate language, while in the second case one writes a reduced application program in C, or another appropriate language, plus a set of programs written in some standard language extended to generate a DataFlow machine (e.g. MaxCompiler, see Figure 2) or in some HDL (Hardware Description Language) to define the internal configuration of the DataFlow machine.

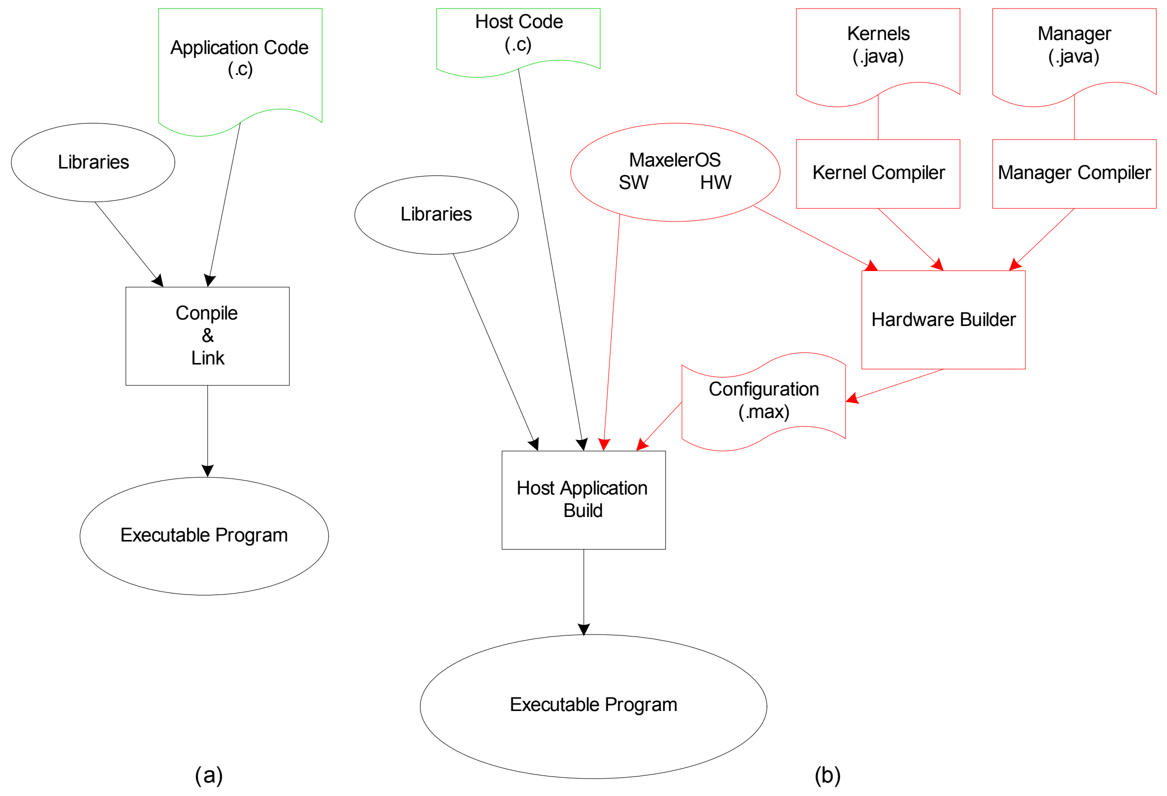

Figure 2: Files involved in the programming process for: (a) MultiCore/ManyCore and (b) DataFlow. The names of the blocks denote the block function. Host Code of DataFlow machines is a subset of the Application Code MultiCore/ManyCore, which is indicated by the size differences of the corresponding blocks (Courtesy of Maxeler). In this context, kernel describes one pipelined dataflow that takes data from input stream, processes it throughout pipeline and produces output stream with results. Manager orchestrates work of kernels and streams of data between kernels, memory and host CPU. Manager and kernels are compiled and hardware configuration is built. Configuration is then linked with host program together with run-time library and MaxelerOS in order to provide configuration of hardware, and streaming data to/from configured hardware.

Programming a dataflow application thus has two stages. The first part of the program (which builds a *.max* file configuration in Figure 2) describes a static synchronous dataflow graph for the part of the application to be executed on the DataFlow processing engine. The second part of the program consists of standard control flow code, plus calls to the static dataflow part of the program via functions.

2.2. What applications are suitable for dataflow?

A key question is what if any parts of a program can benefit from a Dataflow system and which cannot. There are two major factors that can determine this: (1) as described above, it must be possible to describe a static dataflow graph that represents the program (i.e. the inner body of a loop), (2) the computational density and amount of data in the program must be sufficient. That is best explained using Figure 3. In the case of Multi/Many Core systems, more operations means the higher slope of solid line (since more instructions take more time to execute), while for DataFlow system, it remains same until there is no enough area for dataflow implementation (since the dataflow hardware generates some number of results per clock cycle regardless of the number of operations, better explained in sections 4.2 and 4.3).

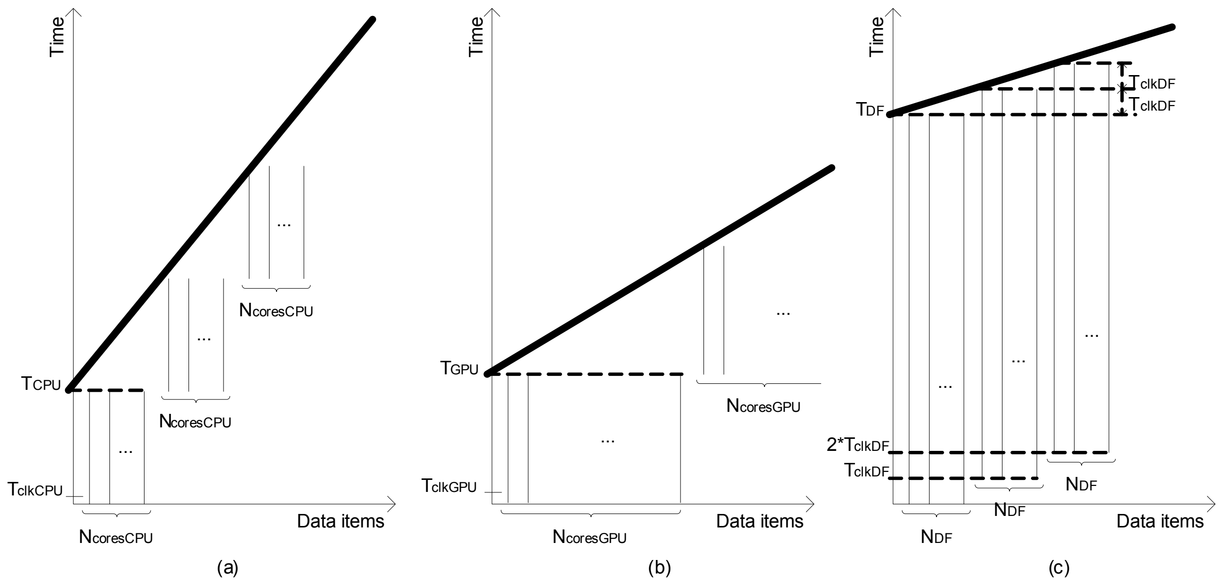

Figure 3: Graphical representation of execution of a loop on three architecture types. It is assumed that there is only one loop, in which iterations are composed of sequential code, and there are no data dependencies between two different loop iterations. On the horizontal axis are input data (discrete scale), while the time is on the vertical axis. One vertical line represents execution of one loop iteration. (a) The MultiCore execution model: $N_{coresCPU}$ – number of available cores in the MultiCore, T_{clkCPU} – clock period of the MultiCore, T_{CPU} – execution time on the MultiCore, the case of one loop iteration; (b) The ManyCore execution model: $N_{coresGPU}$ – number of available cores in the ManyCore, T_{clkGPU} period of clock in the ManyCore, T_{GPU} – one iteration execution time on the ManyCore; (c) Dataflow execution model: N_{DF} – number of implemented dataflows, T_{clkDF} – period of clock for a Dataflow system, T_{DF} – latency of Dataflow pipeline.

Ways in which a DataFlow system exploits available parallelism are similar to those seen in multiprocessors: by pipelining operations and by using multiple cores to work in parallel. There are two main differences. The first one is that the pipeline is made by concatenating all operations from one loop iteration, so it can be up to several orders of magnitude deeper than those in processors. The second one is that the number of cores (a core is one implementation of a dataflow graph) is determined at compile time, making the best possible utilization of the available on-chip area. An application can benefit from a DataFlow system only if it has enough data parallelism to keep all pipelines fully utilized. This is usually the case with loops that process all elements of some array in the same manner, without any dependency between iterations of the loop.

3. Comparing Total Cost of Ownership

3.1. Electricity Bills

Power usage is a major supercomputer consideration. To frame this as a question: in what timeframe does the electricity usage bills equal the initial supercomputer investment? Table 1 includes data from the websites of three reputable manufacturers of MultiCore, ManyCore, and FPGA (FPGA is most often used to implement DataFlow) and two real case studies. For this comparison, power consumption (P) and operating frequency (f), are relates as:

$P = C \times U^2 \times f$

where:

U – Operating Voltage, and

C – A technology type and system size related constant.

As a frequency of modern DataFlow systems, based on FPGA, is almost an order of magnitude lower than the frequency of the same sized MultiCore and ManyCore systems it is expected that the electricity usage of DataFlow system will be one order of magnitude lower than of the other two architectures. Lower monthly electricity bills are desirable, but for an end user it is much more important what will he/she get for his/her money. Comparing same sized systems, with the lower working frequency, the DataFlow system consumes approximately one order of magnitude less energy. If a DataFlow system outperforms a MultiCore system by one order of magnitude, then the difference in GFLOPS/$ is two orders of magnitude. As the performances of the DataFlow and ManyCore systems are comparable (see Table 1), the difference in GFLOPS/$ stays approximately one order of magnitude, in favor of DataFlow system.

Within a chip, energy is dissipated on calculations (energy spent on area where calculations are done) or energy is dissipated on orchestration of data calculations (energy spent on the rest of the area). Now the ratio of the area directly useful for calculations to the rest of area is the largest for DataFlow architectures, we expect that DataFlow systems are the most energy efficient. One contra argument is that a system with X times slower clock requires X times more area for the same amount of calculations per second, leading to conclusion that FPGA is more efficient only if its area directly useful for calculations is more than X times larger than the same area in a chip with a faster clock. Table 1 shows that this is true for modern systems.

Table 1 Declared and measured performance data for three representative systems of presented architectures.

	Intel Core i7-870	NVidia C2070	Maxeler MAX3
0. Type	MultiCore	ManyCore	FineGrainDataFlow
Declared data [7, 8, 9]			
1. Working frequency (MHz)	2930	1150	150
2. Declared performance (GFLOPs)	46.8	1030	450
3. Declared normalized speedup	1	22	9.6
Measured on Bond Option [10]			
4. Execution time (s)	476	58	50.3(*)
5. Normalized measured speedup	1	8.2	9.5
6. Measured power consumption of the system (W)	183	240(**)	87(**)
7. Normalized energy for the system	19.9	3.2	1
Measured on 3D European Option [11]			
8. Execution time (s)	145	11.5	9.6(*)
9.Normalized measured speedup	1	12.7	15.1
10. Measured power consumption of the system (W)	149	271(**)	85(**)
11. Normalized energy for the system	26.5	3.8	1

(*) – Showed results are for reduced precision that is large enough to give the same result SP or DP floating point.

(**) – Power consumption of the whole system: CPU+GPU card or CPU+Max3 card.

Let us consider three systems, such as those mentioned in Table 1, but scaled so that they have the same performance. Let us suppose that electricity bills reach initial investment of MultiCore in N years (N is a small integer). The data from Table 1 (rows 7 and 11) lead to the conclusion that one can expect that with ManyCore the same electricity bill is generated in about 6*N to 7*N years, and with DataFlow in about 20*N to 26*N years. Obviously, the smaller the N, the advantage of the DataFlow technology is more significant.

3.2. Space Costs

The second major supercomputer economic question is how large is the investment into the space to host a supercomputer? The space required for a particular computer is a function of (a) the required performance, (b) the performance of each individual compute unit and (c) the space required for each unit. (c) is determined not just by the physical size of the compute unit but also the power dissipation which limits the amount of resource that can be packed into a small area and still effectively cooled.

Let us suppose that we have 3 systems:

- The first one composed of 1U CPU server with 16 cores,
- The second one composed of 1U GPU server with CPUs and 2 GPU, and
- The third one composed of 1U MPC-X1000 with 8 MAX3 cards.

Assuming that the size of the third system is 40 units, that fits in a standard rack, Table 2 shows performance of the third system and required size of the first and the second systems with the same performance as the third system (size normalization is also done relative to the third system). The performance data, taken from Table 1, are used to scale the first two systems to match the performance of the third system. Here we imply the kind of parallel applications specified in the caption of Figure 3 (those that are ideally scalable).

Table 2. Size comparison of systems with the same performance.

	1U CPU server with 16 cores	1U GPU server with CPU and 2 GPU	1U MPC-X1000 with 8 MAX3 cards
System sizes based on declared maximal performances			
Required size (RU)	769	70	40
Normalized size	19.2	1.7	1
System sizes based on performances on Example 1 (Bond Option)			
Required size (RU)	757	184	40
Normalized size	18.9	4.6	1
System sizes based on performances on Example 2 (3D European Option)			
Required size (RU)	1208	192	40
Normalized size	30.2	4.8	1

One can conclude from Table 2 that the same performing system based on GPU would require several times more space than the one based on MAX3 cards. The CPU based server is inferior in size to both GPU and MAX3 based servers.

In reality, performance does not scale linearly, but there are some problems that are embarrassingly parallel where the previous analysis gives a realistic overview.

4. Elaboration of the Conditions when DataFlow is Superior

Choosing between accelerators requires a good understanding of the possible candidate architectures and how they achieve speed-up. For this reason, analytic models of these accelerators are useful for comparisons. This paper concentrates, between other criterions, on comparison of accelerator performances through analysis of the relationship between execution time and problem size.

4.1 Existing Solutions to Be Compared

For the purpose of comparison, we introduce three models of execution. Exact modeling is almost impossible because of many factors. Instead, we simplify models, so one can group several parameters into one, find out how parameters influence the execution time, and show circumstances under which one architecture has an advantage over another.

4.2 Axioms, Conditions, and Assumptions

In Figure 3, three drawings are given that represent execution time of iterations of some loop. We assume that the loop body contains only sequential code. On the horizontal axis is input data per iteration, and on vertical axis is time.

This analysis does not target all kinds of programs. Instead, it takes into consideration only those programs that have a lot of data parallelism concentrated in one or several nested loops, intended for processing of all elements of some n-dimensional data structure (n is a natural number). Further, this one or these several nested loops are referred to as Data Parallel Region, or just DPR. Usually accelerators speed-up one DPR at a time, and because of that, in the further analysis, we consider only one DPR, not the whole program. In the case of more than one DPR, the same applies for each DPR.

The size of machine code DPR is relatively small compared to the rest of the program, but it represents the most time consuming part of the program. Assuming a MultiCore, typical execution times of DPRs that are of interest for this analysis span from several minutes (or more likely from several hours) to several days. The DPRs with shorter execution time are not of interest, because they are executed fast enough on MultiCore.

Further, we introduce one simplification without loss of generality. We suppose that all DPRs include only one loop. This is possible because the set of nested loops can always be replaced with an equivalent code that includes only a single loop, using a technique called coalescing.

4.3 Analysis

Let us suppose that a single iteration of only loop in DPR is composed of sequential code only and that the bandwidth to/from memory is large enough to fulfill all requests without introducing additional latency. Later we analyze the effect of branch inclusion and limited memory bandwidth.

Let the number of operations in a single iteration of a loop be N_{OPS}, and the number of iterations N. Further, assume a parameter CPI_{CPU} that represents the ratio between average time spent per operation on one core and the clock period of MultiCore (this is the standard CPI definition slightly modified, so that DataFlow architectures can be encompassed, as well). Parameters CPI_{GPU} and CPI_{DF} are the same ratios for the ManyCore and DataFlow, respectively. Using these parameters and those shown in Figure 3, we calculate execution times of a single iteration (T_{CPU}, T_{GPU} and T_{DF}) and execution times of the whole loop (t_{CPU}, t_{GPU} and t_{DF}), for these three systems, as follows:

(a) $T_{CPU} = N_{OPS} * CPI_{CPU} * T_{clkCPU}$
$t_{CPU} = N * T_{CPU} / N_{coresCPU}$
$= N * N_{OPS} * CPI_{CPU} * T_{clkCPU} / N_{coresCPU}$,

(b) $T_{GPU} = N_{OPS} * CPI_{GPU} * T_{clkGPU}$
$t_{GPU} = N * T_{GPU} / N_{coresGPU}$
$= N * N_{OPS} * CPI_{GPU} * T_{clkGPU} / N_{coresGPU}$,

(c) For this system, it is important to consider relation between the capacity of the chip expressed as a maximum number of operations that can fit on the chip (N_{CAP}) and the size of dataflow graph expressed as a number of operations in the graph (N_{OPS}). There are two cases to consider:

1) The chip can contain one or more instances of the graph ($N_{CAP} >= N_{OPS}$):
$T_{DF} = N_{DEPTH} * CPI_{DF} * T_{clkDF}$
$t_{DF} = T_{DF} + (N – N_{DF}) / N_{DF} * T_{clkDF}$
$= N_{DEPTH} * CPI_{DF} * T_{clkDF} + (N – N_{DF}) / N_{DF} * T_{clkDF}$
where N_{DF} (number of implemented graphs) is $[N_{CAP}/N_{OPS}]$, and N_{DEPTH} is depth of dataflow graph ($N_{DEPTH} <= N_{OPS}$).
Total execution time (t_{DF}) is a sum of the time needed to get the first group of N_{DF} results (T_{DF}) and the time for the rest of results. Each next group of results require one additional cycle.

2) The graph cannot fit into the chip ($N_{CAP} < N_{OPS}$):
$T_{DFi} = N_{DEPTHi} * CPI_{DFi} * T_{clkDF}$
$t_{DF} = sum((T_{DFi} + (N – 1) * T_{clkDF}) + T_{reconf})$, for i := 1 .. N_{PARTS};
$= sum((N_{DEPTHi} * CPI_{DFi} * T_{clkDF} + (N – 1) * T_{clkDF}) + T_{reconf})$, for i := 1 .. N_{PARTS};
The dataflow graph is divided into N_{PARTS} smaller graphs, where $N_{PARTS} = ceil[N_{OPS}/N_{CAP}]$. Each one of them can fit into the chip. Total execution time (t_{DF}) is a sum of times needed for reconfigurations of system (T_{reconf} is time of one reconfiguration) and execution times of all graphs inferred from the original graph. Execution time of each particular smaller dataflow graph is analog to the first case where $N_{DF} = 1$.

From these equations, one can conclude that there are two parameters that describe the problem size: N_{OPS} (number of operations executed on each data item) and N (number of data items). In equations (a) and (b), these two parameters are multiplied. As a consequence, increasing the number of operations in a loop iteration, in these two cases, increases the execution time of each group of N_{cores} iterations, and that linearly increases the execution time of the entire loop (N_{OPS} and N are multiplied). In the case (c1), instead of N_{OPS}, there is $N_{DEPTH} <= N_{OPS}$. In this equation, N_{DEPTH} and N (multiplied by some constants) are added together. In this case, increasing N_{OPS} and N_{DEPTH}, but keeping $N_{OPS} <= N_{CAP}$, does not significantly increase execution time, only latency to the first result is increased. The case (c2) shows that the main limitation to high performance is the capacity of hardware used for DataFlow implementations. Today's technology brings enough capacity to make DataFlow systems well performing, often faster than control flow systems [10, 11].

In order to make it clear, N_{DF} is not a constant like $N_{coresCPU}$ and $N_{coresGPU}$. It is a parameter that depends on the size of one implementation of dataflow graph and the size of the chip; the ratio of these two sizes represents the maximum value for this parameter. For smaller loop bodies we are able to decrease execution time on a DataFlow system by implementing additional dataflow graphs (implementing the same dataflow graph several times). This is true only when the assumption about the required bandwidth to/from memory is true. When required bandwidth reaches the available bandwidth, speedup becomes limited only by available bandwidth to/from memory. Does this mean that DataFlow is no more the best choice? If we suppose the same bandwidth to/from memory of all three architectures (for the chips compared, that is not entirely true, but the behavior is explained in the next paragraph), and if some dataflow application is limited by the available bandwidth, then no other architecture can perform faster because if not limited by available processing power of hardware, it is also limited by the available bandwidth to/from memory. This also explains the differences between declared and achieved performances in Table 1. The declared performances suggest that ManyCore achieves a better speedup for applications with a larger computation/data transfer ratio, all that at a higher operational price.

One highly important property of DataFlow system in this context is that it needs same amount of data in each cycle. Except in several cycles at the beginning and at the end, each operation in loop body is executed once per cycle, but with data from different iterations. On the control flow systems, this is the case also; however, due to the lower level execution paradigm in DataFlow machines, overlapping of memory accesses and calculations is more effective. As a consequence, it is expected that the DataFlow system can easily utilize whole available bandwidth to/from memory, while other two architectures can do the same only in extreme case when available bandwidth is lower than required.

The other influence of memory on execution time is the latency of memory accesses. In control flow systems, it is seen through processor's stall time when it comes to a cache miss, possibly increasing CPI. In DataFlow systems, this latency is hidden by overlapping memory accesses with calculations.

In real cases, loop body sometimes contains control flow instructions. In the control flow architectures, these instructions have a negative influence on the execution time of the whole program, due to their execution time and increased CPI caused by control hazards. In DataFlow architectures every possible option of each control flow instruction is calculated at run time, and at the end of the flow, one of the calculated values is chosen. This influences only the latency for the first group of results. More importantly, area on the chip is not efficiently utilized, significantly reducing the effectively available capacity of hardware. For the most of the real applications implemented in DataFlow systems, this does not have any significant drawback [10, 11].

5. Conclusion

The optimal approach implies a hybrid solution that includes all three architecture types: MultiCore, ManyCore, and DataFlow. So far, hybrid approaches would typically include only two solutions (e.g., Mont-Blanc Barcelona supercomputer). Our conclusion goes one step beyond and adds a third component: DataFlow.

For a typical supercomputer workload mix, this means a considerable performance improvement, since the percentage of supercomputer code which is best dataflow-able is relatively high (e.g., [1]) and can bring speedups of 20-40 or even more.

Introduction of the third component into the hybrid implies appropriate changes in the programming model (e.g., Ompss or [12]) and an incorporation of the dispatcher software which is able to recognize what is best to move to the DataFlow component of the hybrid, and how to do it. It can be implemented for either compile time or run time.

Frame No 1: To Select or to Hybridize.

When a supercomputer team faces a new programming challenge, the first thing to do is to decide what supercomputer architecture to select for the highest performance, the lowest power consumption, and the smallest equipment volume. An alternative is to use a hybrid machine which includes all three architectural types (MultiCore, ManyCore, and DataFlow) and a sophisticated software dispatcher (partially implemented by the programmer, partly in the compiler and partially in the operating system) that decides what portion of the high level language code goes to what architectures.

Frame No 2: A Symbolic Comparison of MultiCore, ManyCore, and DataFlow

A well known anecdotal way to compare MultiCore and ManyCore supercomputers is to compare the two approaches with horses and chickens that pull a load wagon. Along the same anecdotal path, one can compare the DataFlow approach with ants that carry load in their back-packs. Further, an analogy can be established between power consumption and feeding the animals, between cubic foot and “housing” for animals, and between speed for Big Data and running fast up a vertical wall.

Feeding horses is much more expensive than feeding chicken, and feeding chicken is much more expensive than feeding ants (read “feeding” as “paying monthly electricity bills”). Electricity bills may double initial investments for MultiCore in only a few years, and not in many more years for ManyCore, while DataFlow machines need as much as a few decades for the same.

Stables for horses can be extremely large (some supercomputer centers are even building new buildings for their next generation machines). Chicken houses are much smaller then stables, but much bigger then ant holes. The cost of placing gates across the field has two components: (a) deciding where to put them (higher programmer effort) and (b) physical putting at decided locations (longer compile time).

Only ants can move fast up a vertical wall. Chicken and horses cannot. In other words, if an extremely large data set is crunched, the DataFlow approach is the fastest, as indicated in the non-anecdotic part of this paper.

Frame No 3: Definition of Multi/ManyCore and Fine/CoarseGrainDataFlow

Multi/ManyCore architectures are composed of general purpose processing cores, similar to factory where each worker works on everything. While MultiCore is composed of a small number of highly sophisticated and high speed processing cores, ManyCore is composed of a larger number of simpler and slower processing cores.

At the opposite side are DataFlow architectures that process data in a manner similar to factory where workers are arranged in assembly line and each worker works on only one operation on each product. FineGrainDataFlow architectures are composed of special purpose processing elements interconnected to form hardware through which data will be processed. CoarseGrainDataFlow architectures are composed of general purpose cores and DataFlow concept is used on a software level in order to utilize as much as possible of a parallelism available in program.

6. Acknowledgement

This work has been partially funded by the Ministry of Education and Science of the Republic of Serbia (III44009, TR32047, and III44006).

7. References

[1] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, and M. J. Flynn, "Surviving the End of Scaling of Traditional Microprocessors in HPC," IEEE Hot Chips 22, Stanford, USA, August 2010.

[2] R. Vetter, D. Kline, and K. Barnhill, "Building an Effective Interdisciplinary Professional Master's Degree," Information Systems Educators Conference (ISECON'12), New Orleans, USA, November 1-4, 2012.

[3] A. Gupta, L. V. Kalé, D. S. Milojicic, P. Faraboschi, R. Kaufmann, V. March, F. Gioachin, C. H. Suen, B.S. Lee, "Exploring the Performance and Mapping of HPC Applications to Platforms in the Cloud," HPDC, 2012, pp.121-122.

[4] S. Stojanovic, D. Bojic, M. Bojovic, M. Valero, and V. Milutinovic, "An Overview of Selected Hybrid and Reconfigurable Architectures," Advances In Computers [In Press].

[5] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel Computing Experiences with CUDA," Micro, IEEE , vol. 28, no. 4, July-Aug. 2008, pp.13-27.

[6] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," In *Proceedings of the 37th annual international symposium on Computer architecture*, New York, USA, 2010, pp.451-460.

[7] P. Gepner, V. Gamayunov, D.L. Fraser, "The 2nd Generation Intel Core Processor. Architectural Features Supporting HPC," 10th International Symposium on Parallel and Distributed Computing (ISPDC), July 2011, Cluj Napoca.

[8] NVidia, "Tesla C2050 and C2070 GPU computing processor," http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf

[9] P. Sundararajan, "High Performance Computing Using FPGAs," http://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf

[10] Q. Jin, D. Dong, A.H.T. Tse, G.C.T. Chow, D.B. Thomas, W. Luk, and S. Weston, “Multi-level Customisation Framework for Curve Based Monte Carlo Financial Simulations,” Lecture Notes in Computer Science, Volume 7199, 2012, pp.187-201.

[11] A.H.T. Tse, G.C.T. Chow, Q. Jin, D.B. Thomas, and W. Luk, “Optimising Performance of Quadrature Methods with Reduced Precision,” Lecture Notes in Computer Science, Volume 7199, 2012, pp.251-263.

[12] A. Prabhakar, V. Getov, “Performance Evaluation of Hybrid Parallel Programming Paradigms,” Performance Analysis and Grid Computing, Kluwer Academic Publishers, 2004.