The challenges of Parallel programming and many-core architectures

Mick van Gelderen 4091566
Arian Stolwijk 4001079

November 2013

Since 2002 the way to increase performance is by using multiple cores and the number of cores keeps increasing. Utilizing the full potential of the entire processor in a parallel way is very difficult for programmers. This paper discusses various papers that describe parallel programming models that are designed helping programmers to increase the performance of their programs. Finally we've compared the solution by performance, ease of use and scaling.

1 Introduction

Over the past few decades, processors have increased their performance. From 1986 to 2002 the annual increase in growth was around 52% per year. Since 2002 however, the limits of power, heat dissipation, available instruction-level parallelism, and long memory latency have slowed uniprocessor performance, to about 20% per year [3]. Rather improving uniprocessors, Intel joined IBM and Sun in improving processor performance by using multiple processors.

With the introduction multiple processors there is a switch of focus from instruction level parallelism (ILP) to *thread-level parallelism* (TLP) and *data-level parallelism* (DLP). Whereas compilers and hardware can exploit ILP implicitly, TLP and DLP require the programmer's attention to write parallel code in a performant way [3].

Exploiting all the advantages of this kind of processors is not be trivial, and one of the main challenges is how to utilize all the available on-chip computing power [2].

Exploiting all the advantages manually is extremely difficult, especially when it's not the programmers main field of interest, like physical scientists [4]. Also generality of solutions for different type of architectures is an problem [2]. Not only CPUs with two, four, six or eight cores are considered, also GPUs with even more simpler computational cores are increasingly more used for general computational tasks [5]. Besides multiple cores computations can also be parallelized across multiple computational nodes which consist out of multiple computational cores [1].

There are several parallel programming models that can help utilize multi-core or many-core processors. Parallel programming models can be divided into two kinds: *data-parallel* and *task-parallel* programming models. Data-parallel focusses on solving data-independent problems and trying to distribute data on different parallel computing nodes. With task-parallel programmers focus on decomposing the problem into sub-tasks that can run in parallel.

In the next section we will summarize four relevant papers about parallel programming models, then the papers and solution from the papers are compared by several criteria and finally some conclusions.

2 Discussion of different papers

To investigate parallel programming for many-core architectures we've considered four papers by different authors. The papers discus different parallel programming methods or provide their own parallel programming solution. The following sections each summarize a paper.

2.1 Performance Analysis of Current Parallel Programming Models for Many-core Systems

Performance Analysis of Current Parallel Programming Models for Many-core Systems [2] is a paper written by Yangjie Cao et al. in 2013.

To solve problems faced by many-core processor systems, extensive and intensive research is carried out on parallel programming models. A good programming model is usually judged on its generality. A good programming model can provide an essential bridge between hardware and high-level programming languages.

Task-parallel programming is becoming increasingly popular in the last decade. Several task-parallel libraries have been developed. These libraries allow programmers to easily generate parallel tasks by handling how the parallelization is realized on the underlying hardware. Task scheduling and load balancing are transparently completed by the runtime systems.

There are three typical task-parallel programming models described: Intel Thread Building library (TBB), MIT Cilk and OpenMP 3.0. Several experiments are done in order to compare these programming models.

Cilk is a task-based parallel programming model language. Cilk philosophy is that the programmer can expose parallelism using a few keywords, while the Cilk runtime does the hard work of scheduling the computation to run efficiently on a given platform. The only addition to the programming language are three keywords: `cilk`, `spawn` and `sync`.

TBB is a parallel programming library developed by Intel. It is mainly used the way of task instead of thread to express parallelism. The library can map the task efficiently on threads. TBB is a library so does not require special languages or an extra compiler. TBB tasks are a special kind of C++ objects, which allows the TBB model to explore a variety of parallelism. It supports nested parallelism, so you can build larger parallel components.

OpenMP, which stands for Open Multiprocessing, was published by a group of major hardware and software vendors. It gives programmers a simple and flexible interface for developing parallel applications. The main aim is to simplify programming of multi-platform, shared-memory, parallel programs by insertions of pragmas into C/C++ code or special comments in Fortran code. These insertions instruct the compiler which code should be parallelized. In the view of OpenMP, tasks are units of work which are executed in parallel. A disadvantage of OpenMP is that it lacks fine grained control.

To compare the different parallel programming models the authors conducted experiments to evaluate and compare the performance of different models. There were different benchmarks used (each model has different benchmarks). The experiments were conducted on real many-core systems.

From experiment results most test applications obtain different levels of performance speedup. Most applications achieve linear speedup when the number of cores are increased. However when the number of cores is increased further, the speedup becomes slower or even decreases.

The authors conclude that parallel programming models are crucial for easy-to-understand, concise and dense representation of parallelism. The experiments show that there are still many problems: (a) with scaling to many-core systems, (b) low utilization and (c) resource competition.

2.2 Parallel Phase Model: A programming Model for High-end Parallel Machines with Manycores

Parallel Phase Model: A programming Model for High-end Parallel Machines with Manycores [1] is a paper written by Ron Brightwell et al. in 2009.

Distributed parallel programming has mostly been done using the message passing model, such as MPI. This has been very successful but still require programmers to handle low-level tasks including explicit management of data distribution, data locality management, communication preparation and scheduling, synchronization and load balancing in order to achieve good performance. With the introduction of multi-core and many-core machines will make developing applications even more difficult because there will be node-level parallelism and cluster-level parallelism.

A parallel programming model provides abstractions for programmers to express parallelism and exploiting the capabilities of the underlying hardware architecture. A programming model is typically implemented in a programming language, runtime library, or both. On physical shared memory machines, models such as POSIX Threads and OpenMP are also very useful; but in reality, high-end parallel machines tend to be distributed memory machines.

This paper presents a programming model for parallel machines to address those parallel programming difficulties. This model is called the Parallel Phase Model (PPM). A main design points of the library are ease of use and performance.

The PPM abstraction includes the following principles:

- Virtualization of processes: programs can have an unbound number of virtual processors rather than the fixed number of physical processors.

- Virtualization of memory: virtual processors ”communicate” through shared variables: globally-shared at cluster level and physically shared at node level.
- Implicit communication: shared variables make communication between processors implicit rather than explicit.
- Implicit synchronization: the programming language constructs have built-in implicit synchronization in the semantics.
- Automatic data distribution and locality management
- Layered parallelisms: global level or node level parallelism can be separately expressed.

PPM adds some programming language constructs (to C):

- **Declarations:** `PPM_global_shared` or `PPM_node_shared` can be added before a variable declaration. Global shared is one variable for all systems while node shared creates a variable for each system in the networked cluster.
- **Control constructs:** `PKK_do(K) func(arguments);` will run the function “func” in parallel on K instances.
- **PPM Functions:** Special functions that is started by the `PPM_do` construct.
- **Parallel Phase Construct:** this are constructs which provide a mechanism for implicit synchronization of parallel execution and shared variable updates across multiple instances of the PPM functions.
- **System variabler:** The PPM programming environment exposes some variables such as `PPM_node_count`, `PPM_cores_per_node` and `PPM_node_id`.
- **Utility function:** various utility functions.

PPM is a SPMD model, which means that there is one copy of the same program on each physical node of the cluster. All copies run in parallel.
PPM has a few design focuses

- Algorithm parallelism expressiveness
- Layered parallelisms
- Guidance for good programming style
- Simple and implicit synchronization
- Simple memory model

Table 1: Code Size (Number of lines)

Application	PPM Program	MPI Program
Conjuncate Gradient	161	733
Matrix Generation	424	744
Barnes Hut	499	N/A

- No need for explicit communication
- Shared data coherence does not reply on hardware cache capability
- Supporting both synchronous and asynchronous mode on different nodes
- Automatic scheduling of computation and communication needs, cores, and network resources

PPM is implemented as a source-to-source compiler. A light runtime will do most of the optimizations. Using this runtime it does all the memory sharing, scheduling, remote communication management, among others.

PPM was tested on a Cray XT4 machine with a total of 9660 compute nodes, each node having 4 cores. Three different application were considered:

1. Conjugate Gradient Solver of Linear Systems
2. Sparse Matrix Generation for Multi-scale Collocation Method
3. Barnes-Hut Simulation

In summary, the PPM model provides good performance for unstructured applications on current multi-core clusters.

The PPM implementation are much smaller than MPI implementation of the same application. However the MPI programs have very fine-grained communication messages for good performance. In PPM this is all implicit so they can be much smaller, while having the same or better performance.

The paper concludes with the statement that it presented a parallel programming model for the next generation high-end machines; machines with a cluster of nodes where each node has a large number of processing cores.

2.3 Exploiting Multi- and Many-core Parallelism for Accelerating Image Compression

Exploiting Multi- and Many-core Parallelism for Accelerating Image Compression [4] is a paper written by Cheong Ghil Kim and Yong Soo Choi in 2011. It describes how they used Intel TBB and OpenCL to improve the performance by running a 2D DCT (discrete cosine transform) in parallel.

As discussed in section 1, the current solution to get better performance is adding more cores to CPUs and GPUs. That is why parallel computing techniques such as incorporating multiple processing cores and other acceleration technologies are increasing in importance. In order to take advantage of multi-cores, programs should be written to accomplish their tasks using multiple parallel thread execution.

Using GPUs, which have many cores requires a different type of parallelism: massively parallel programming. OpenCL is a new standard for task-parallel and data-parallel heterogeneous computing for, among others CPUs, GPUs and DSPs.

There are different platform specific solutions, like BrookGPU, Cg or CUDA, or some general parallel programming languages such as OpenMP and MPI. Unfortunately they are platform specific or hard to use on GPUs for GPU-based heterogeneous system platforms. Therefor OpenCL is proposed for programming GPUs and accelerators including multi-core CPUs without modifying computing kernels.

Multi-core CPUs First steps of parallelization are Instruction Level Parallelism (ILP), where multiple instructions can be executed in one clock cycle. Simultaneous Multi-Threading (SMT) permits multiple independent threads of execution to better utilize the resources for Thread Level Parallelism (TLP). The next trend is the processors using chip multiprocessing (CMP). As a result dual and quad core systems are introduced, as well as six and eight cores.

To extract high performance from multi-core architectures, Intel provides TBB, a C++ runtime library that targets desktop shared memory parallel programming.

Many-core GPU Differently from multi-core CPUs, many-core GPUs consists of much simpler, but more processing units helping for massively parallel computing. It can handle many threads through Single Instruction Multiple Threads (SIMT).

GPUs used to be very difficult to program, but from 2001 and onwards each generation of GPUs added more features for the GPU to be programmed, which also make them potentially useful for more general computation-centric tasks.

The paper gives an overview of two parallel programming technologies which enables to exploit task- and data-level parallelism: (1) TBB and (2) OpenCL. TBB is an open source runtime C++ library that targets shared desktop shared memory parallel programming. Because it's a library it integrates well into existing languages without changes to the compiler. OpenCL is a recent standard to support parallel execution on CPUs, GPUs, DPSs or other special purpose coprocessors. The main OpenCL C program includes the OpenCL runtime. The OpenCL kernels are compiled at runtime and loaded into memory. Input and output data locations are set up right before the kernel execution, see figure 1.

The experiment in the paper uses DCT (discrete cosine transform). This is basically large 2D matrix data computations. The paper implements four different serial DCT versions: (1) implemented directly with serial processing, (2) with precomputing coefficients, (3) based on (2) but using direct matrix multiplication and (4) is the implementation of row-column decomposition method. Another input variable is the matrix

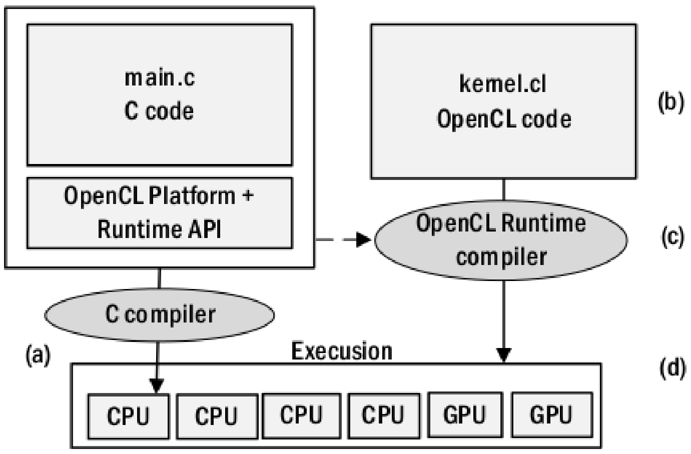

Figure 1: OpenCL execution

size, which is 256, 512, 1024 or 2048. For performance benchmarking the paper used TBB and OpenCL. From the results we see that the OpenCL implementation outperforms the one on TBB. For the parallel DTC implementations the speedup compared to the serial ones are 4.8 and 6.9 times for TBB and OpenCL, respectively. Especially it shows linear speedup, as the increase of 2D data sets.

2.4 Data Parallel Programming Model for Many-Core Architectures

The paper *Data Parallel Programming Model for Many-Core Architectures* [5] by Yongpeng Zhang, written in 2011, describes a data streaming framework on GPUs.

GPUs have proved successful providing significant performance gains by exploiting data parallelism in existing algorithms. Programmers however are used to writing sequential programs. The paper proposes a framework GStream, a general-purpose, scalable data streaming framework on GPUs.

The amount of data that has to be processed continues to grow. However Processor frequencies have reached their limits. Traditional instruction level parallelism can no longer provide worthy performance benefits. A higher degree of parallelism has to be extracted, both in algorithmic or implementation level, to fully utilize emerging multi and many-core architectures.

Massive data-parallelization on the GPU can already be achieved with the NVIDIA GPU programming model CUDA. CUDA encourages users to create light-weight software threads at scale of thens of thousands, which is magnitudes larger than the maximum hardware concurrency inside the GPU. Although CUDA has great performance benefits, it requires deep understanding of the underlying architecture. Therefore it is desirable to develop a new programming model to reach an acceptable balance between programmability, portability and performance that accommodate the increasing number of cores per chip.

The paper looks at the possibilities of the stream processing domain. Usually used in video encoding/decoding scenarios. Other domains such as data analysis and computationally intensive tasks are also discovering the benefits of the underlying streaming paradigm. The GStream streaming framework developed by the authors contributes the following: (1) abstraction expresses data-parallelism more naturally than task-oriented parallelism, (2) extreme concise and intuitive abstraction, (3) data-parallel approach reduces data dependencies, (4) first work to exploit streaming applications on clusters of GPUs, (5) the validity of the abstraction reaches beyond streaming.

Two key concepts of GStream are *filter* and *channels*. A filter encapsulates the computing kernels that can be accelerated by the GPU. The data-parallelism inherently to filter kernels is facilitated by simple APIs to manipulate data in channels, which represent data links between any two filters.

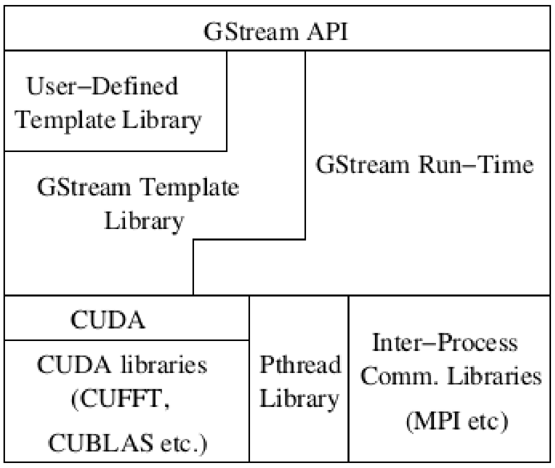
GStream API
User–Defined Template Library
GStream Template Library
GStream Run–Time
CUDA
CUDA libraries (CUFFT, CUBLAS etc.)
Pthread Library
Inter–Process Comm. Libraries (MPI etc)

Figure 2: GStream Software Stack

Figure 2 describes the GStream Software stack. It uses CUDA for data-parallelism and POSIX threads for task-parallelism. None of the concrete components are mandatory, as long as they are replaced with a library with the same functionality.

The paper tested the acceleration by the GStream library with five benchmarks. Using four different implementations: (a) A native single-threaded C/C++ program; (b) a multi-threaded C/C++ program using the GStream library without GPU support; (c) a multi-threaded program using GStream with GPU support and (d) a native CUDA implementation.

For three of the five benchmark applications the speedup is between 3 and 30 times. Comparing implementations (a) with (b) and (c) with (d) shows that the GStream imposes little overhead to the overall system.

3 Comparison

For comparison and evaluation of different kinds of parallel programming models there are a few criteria.

- **Efficient exploitation of all the hardware:** If there are N cores in a processor, how well does the programming model use all the N cores, using good scheduling, load balancing and memory management.
- **Ease of use for programmers:** As utilizing all hardware is very difficult to do manually, programming models and frameworks can help to make it easier for programmers. Also how the method is implemented in the programming language, as pragmas, language extensions or library APIs is of importance to the learning curve and maintainability.
- **Scaling:** Can the solution be used on different architectures, how does it scale to many-core processors? Does it only work on CPUs or also GPUs, DSPs or even across a cluster.

Performance [2] presents a good comparison for how the Cilk, TBB and OpenML programming models scale to more cores. Each programming model is tested with different benchmarks so it's difficult to actually compare the actual speedup ratios. What's also visible is that the speedups become less when the number of cores increase, which can be explained by Amdahls's Law [3]. OpenCL and TBB are compared in [4] with a discrete cosine transform application, this paper focusses on computations on the GPU, which shows that for that application the OpenCL method performs better than TBB. [1] compares MPI with PPM for clusters of multi-core machines and shows that the PPM is comparable in performance. Finally [5] presents GStream and does a performance comparison between GStream and CUDA for GPUs and show that the difference in performance between the two is in fact very small.

Ease of use Especially [1] focusses on the ease of use for parallel programming. It compares their PPM solution with the message passing model MPI and get a substantial reduction of code size of the same problem (as shown in table 1). It adds extra declarations to the C programming language and needs an extra source-to-source compiling step. TBB described in [4, 2] is a C++ library and does not need special languages or compilers. It is not further described if using TBB tends to be easy or difficult. Cilk adds three keywords to the programming language and aims for simplicity [4]. Simplicity is also an aim of OpenMP [4]. Besides [1] the other papers don't provide any metrics that support claims about some method being easy to use for programmers or not.

Scaling One of the reasons of the programming models described in the four papers is that they can abstract platform specific things. For example GStream's architecture (figure 2) can use CUDA, but the concrete implementation can be exchanged for something else, so it is not NVIDIA dependent anymore [5]. Also the programmer doesn't have to think about the number of actual cores anymore, because that's somewhere the programming model can manage. Between the papers is a difference that [1] describes an model that even generalizes over physical machines to a network cluster using an intelligent allocation mechanism. For user applications this is slightly more costly on a lower amount of cores but pays off when the number of cores approaches 100 or more. There the programmer does not need to worry about which node and core does the computation, only that the computation kernel should be parallelized. Other programming models like GStream and OpenCL focus more on GPUs [4, 5].

4 Conclusion

It is clear that utilizing the full potential of the processor is crucial for high performance. Being able to utilize that power it should be made easier for programmers to write parallel programs. Most programmers are good at writing sequential programs but writing good parallel programs can be very difficult. Even though the numbers of cores keep increasing, the speedup is also very much dependent on the ratio of the program that can be parallelized [3]. The four papers summarized in section 2 describe different programming models that can help achieving this, but [2] showed that not all programming models scale that well to many-core architectures.

Many of the parallel programming models discussed in the papers describe how you can rewrite existing algorithms to be distributed across the cores of a processor. Usually this is done by sprinkling some extra language constructs or library calls, for example instead a `for` statement a `cilk_for` statement. Maybe the answer does not lie in finding ways to parallelize sequential algorithms but in educating programmers to think sequentially. An other approach towards the same goal is finding ways of programming that allow parallelization which are more intuitive to programmers. Event based programs for example are for a lot of programmers much easier to understand. If we manage to find efficient algorithms for effectively finding parallelizable events, better parallelization can be achieved while it is essentially coming from the programming level.

The ideal model is one that is easy to use and learn while having the effective computational power scale linearly with the amount of cores. The PPM model seems to be closest to that from the models we have discussed and it is definitely a good starting point for further developments. However, the industry seems to be heavily focussed on bringing the technique from GPU's to the CPU and maybe we need to take an entirely different approach.

References

[1] R. Brightwell, M. Heroux, Zhaofang Wen, and Junfeng Wu. Parallel phase model: A programming model for high-end parallel machines with manycores. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 92–99, 2009.

[2] Yangjie Cao, Baodong Wu, Yongcai Tao, and Lei Shi. Performance analysis of current

parallel programming models for many-core systems. In *Computer Science Education (ICCSE), 2013 8th International Conference on*, pages 132–135, 2013.

[3] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2007.

[4] Cheong-Ghil Kim and Yong Soo Choi. Exploiting multi- and many-core parallelism for accelerating image compression. In *Multimedia and Ubiquitous Engineering (MUE), 2011 5th FTRA International Conference on*, pages 12–17, 2011.

[5] Yongpeng Zhang. Data parallel programming model for many-core architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2065–2068, 2011.