Parallel programming models

Ing. Michal ÁČ

Armed forces academy of General Milan Rastislav Štefanik
Demänová 393
031 06 Liptovský Mikuláš
acmic@mosr.sk

Abstract: A parallel programming model describes an abstract of parallel machine by its basic operations (arithmetic operations, reading and writing to shared memory, receiving and sending messages), their effects on the state of computation, the constraints of when and where these can be applied, and how they can be composed. A parallel programming model is often associated with one or several parallel programing languages or libraries that realize the model.

Keywords: parallel programming model, SIMD, NUMA, message passing, PRAM, BSP

1. Introduction

Parallel programming and the design of efficient parallel programs is well established in high performance scientific computing for many years. The more precise simulation models of larger problems need greater and greater computing power and memory space. In the last decades, high performance research included new developments and high performance computing can be observed. Very popular examples are simulations of weather forecast based on complex mathematical models involving partial differential equations or crash simulation from car industry based on finite elements methods.

In weather forecast, future development in the atmosphere area has to be predicted, and to do so, they can only be obtained by simulations. More and more precise simulations can be performed with high financial effort. And this is area where parallel computing is in place, to save financials with equal or higher computation potential. A low performance of the computer system used can restrict the simulations and accuracy of the results obtained significantly. In particular, using a high performance system allows larger simulations which lead to better results.

Therefore, parallel computers have often been used to perform computer simulations. In this article we will discuss parallel programming models.

A parallel programming model describes an abstract parallel machine by its basic operations (arithmetic operations, reading and writing to shared memory, receiving and sending messages), their effects on the state of computation, the constraints of when and where these can be applied, and how they can be composed. A parallel programming model is often associated with one or several parallel programing languages or libraries that realize the model.

Programming models abstract to some degree from details of the underlying hardware which increases portability of parallel algorithms across a wider range of parallel programming languages and systems.

In this paper I will focus on the most relevant models, this is not aimed at providing a comprehensive presentation or classification of parallel models and languages.

2. Model review

2.1 Parallel Random Access Machine (PRAM)

The parallel random access machine model was proposed by Fortune and Wyllie as a simple extension of the Random Access Machine (RAM) model used in the design and analysis of sequential algorithms. The PRAM assumes a set of processors connected to a shared memory. There is no limit on the number of processors accessing shared memory simultaneously.

The memory model of the PRAM is strict consistent, a write in clock cycle t becomes globally visible to all processors in the beginning of the cycle $t+1$, not earlier or later.

The PRAM model determines the effect of multiple processors writing or reading the same memory location in the same clock cycle. We determine EREW PRAM that allows a memory location to be exclusively read or written by at most one processor at a time, the CREW PRAM allows concurrent reading but exclusive writing, and CRCW PRAM allows simultaneous write accesses by several processors to the same memory location in the same clock cycle.

The PRAM model is unique in that it supports deterministic parallel computation, and it can be regarded as one of the most programmer friendly models. Several PRAM programming languages have been proposed, as Fork, PRAM simulator an so on. Methods for translating PRAM algorithms automatically for other models such as message passing exists.

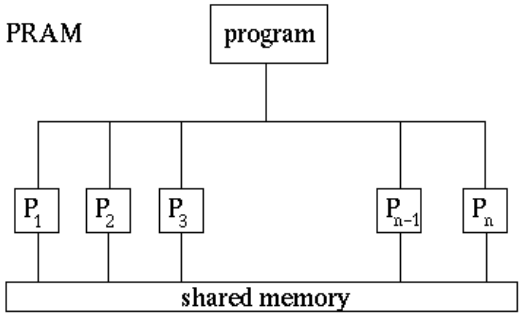

Fig.1. *Figure illustrates how a PRAM would look, with each processor sharing the same memory and by extension the program to execute.*

2.2 Unrestricted Message Passing

A distributed memory machine, sometimes called message-passing multicomputer, consists of a number of RAMs that run asynchronously and communicate via messages sent over communication network. In the simplest form, a message is assumed to be sent by one processor executing an explicit send command and received by another processor with explicit receive command (point-to pint communication). Command can be blocking (for processors get synchronized) or non-blocking (sending processor puts the message in a buffer and proceeds with its program, while the message-passing subsystem forwards the message to the receiving processor and buffers it there until receiving processor executes the receive command.

Message passing models such as CSP (communicating sequential processes) have been used in the theory of concurrent and distributed systems for many years. Message passing programs can quite easily be executed even on shared memory computers. Message passing gives the programmer the largest degree of control over the machine resources, including scheduling, buffering of message data, overlapping communication with computation, etc. Today message passing is the most successful parallel programming model in practice. As a consequence, numerous tools e.g. for performance analysis and visualization have been developed for this model. Early vendor-specific libraries were replaced in the early 1990s by portable message passing libraries such as PVM and MPI. Today, there exist several widely used implementations of MPI, including open-source implementations such as OpenMPI.

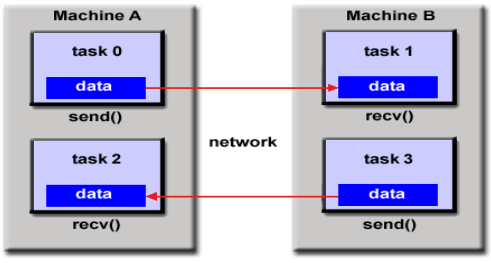

Fig. 2. Message passing model

2.3 Bulk Synchronous Parallelism

The *bulk-synchronous parallel (BSP)* model, proposed by Valiant in 1990 and slightly modified by McColl , enforces a structuring of message passing computations as a dynamic sequence of barrier-separated supersteps, where each superstep consists of a computation phase operating on local variables only, followed by a global interprocessor communication phase.

Many algorithms for the BSP model have been developed in the 1990s in the parallel algorithms community. The BSP model is mainly realized in the form of libraries such as BSPlib.

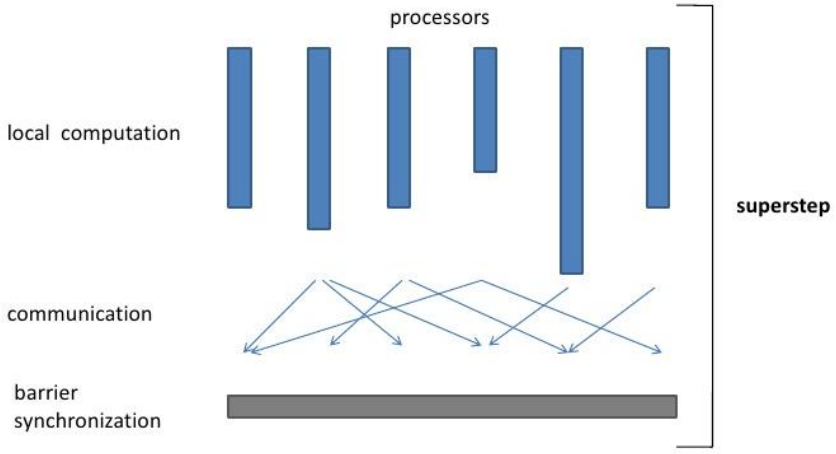

Fig. 3. BSP representation

2.4 Asynchronous Shared Memory

In the shared memory model, several threads of execution have access to a common memory. However, the threads of execution run asynchronously, i.e. all potentially conflicting accesses to the shared memory must be resolved by the programmer, possibly with the help of the system underneath.

If several processors share a physical memory via a bus, the cost model resembles that of a RAM, with the obvious practical modifications due to caching, and consideration of synchronization cost. This is called *symmetric multiprocessing (SMP).* If there is only a shared address space that is realized by distributed memory architecture, then the cost of the shared memory access strongly depends on how far the accessed memory location is positioned from the requesting processor. This is called *non-uniform memory access (NUMA).* In order to avoid remote memory accesses, caching of remote pages in the local memory has been employed, and been

called *cache coherent NUMA (CC-NUMA).* A variant where pages are dynamically placed according to access has been called *cache only memory access (COMA).* While NUMA architectures create the illusion of a shared memory, their performance tends to be sensitive to access patterns and artefacts like thrashing, much in the same manner as uniprocessor caches require consideration in performance tuning.

Shared memory programming has become the dominant form of programming for small scale parallel computers, notably SMP systems. As large-scale parallel computers have started to consist of clusters of SMP nodes, shared memory programming on the SMPs also has been combined with message passing concepts, especially OpenMP. OpenMP is gaining popularity with the arrival of multicore processors and may eventually replace Pthreads completely. OpenMP provides structured parallelism using fork-join styles.

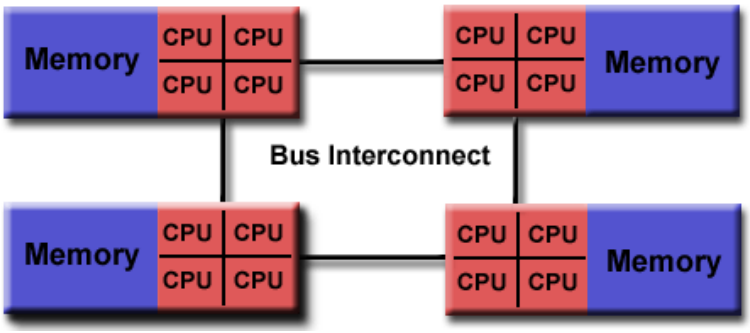

Fig. 4. NUMA model

2.5 Data Parallel Models

Data parallel models include SIMD (Single Instruction, Multiple Data) and vector computing, data parallel computing, systolic computing, VLIW computing, and stream data processing.

Data parallel computing involves the elementwise application of the same scalar computation to several elements of one or several operand vectors (which usually are arrays), creating a result vector. All element computations must be independent of each other and may therefore be executed in any order, in parallel, or in a pipelined way. The strength of data parallel computing is the single state of the program's control, making it easier to analyze and to debug than task-parallel programs where each thread may follow its own control flow path through the program.

A special case of data parallel computing is *SIMD computing* or *vector computing.* Here, the data parallel operations are limited to predefined SIMD/vector operations such as element-wise addition.

Systolic computing is a pipelining-based parallel computing model involving a synchronously operating processor array (a so-called *systolic processor array)* where

processors have local memory and are connected by a fixed, channel-based interconnection network.

In a *very large instruction word* (VLIW) processor, an instruction word contains several assembler instructions. Thus, there is the possibility that the compiler can exploit instruction level parallelism (ILP) better than a superscalar processor by having knowledge of the program to be compiled.

Most modern high- end processors have vector units extending their instruction set by SIMD/vector operations. SIMD languages include Vector-C and *C**. Fortran90 supports vector computing and even a simple form of data parallelism.

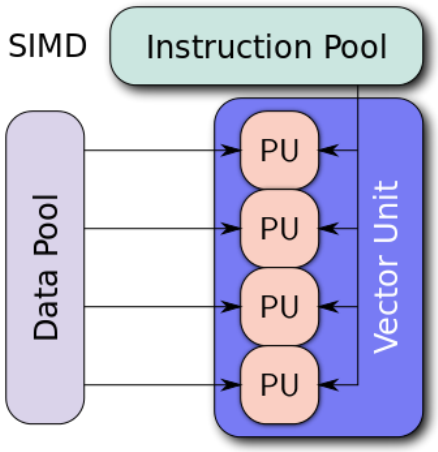

Fig. 5. SIMD model

2.6 Task Parallel Models

Many applications can be considered as a set of *tasks*, each task solving part of the problem at hand. Tasks may communicate with each other during their existence, or may only accept inputs as a prerequisite to their start, and send results to other tasks only when they terminate. Tasks may spawn other tasks in a fork-join style, and this may be done even in a dynamic and data dependent manner. Such collections of tasks may be represented by a *task graph*, where nodes represent tasks and arcs represent communication, i.e. data dependencies.

Task graphs also occur in *grid computing* , where each node may already represent an executable with a runtime of hours or days. The execution units are geographically distributed collections of computers. In order to run a grid application on the grid resources, the task graph is scheduled onto the execution units.

Grid computing has gained considerable attraction in the last years, mainly driven by the enormous computing power needed to solve grand challenge problems in natural and life sciences.

3. Conclusion

At the end of this review of parallel programming models lets speculate a bit about the future of parallel programming models. The future of computing is parallel computing, dictated by physical and technical necessity. Parallel computer architectures will be more and more hybrid, combining hardware multithreading, many cores, SIMD units, accelerators and onchip communication systems. We foresee a multi-layer model with a simple deterministic high-level model that focuses on parallelism and portability, while it includes transparent access to an underlying lower-level model layer with more performance tuning possibilities for experts. New software engineering techniques may help in managing complexity.

References

1. http://www.ida.liu.se/~chrke55/papers/modelsurvey.pdf
2. https://computing.llnl.gov/tutorials/parallel_comp/#Models
3. Thomas Rauber, Gudula Rünger, Parallel Programming Models