A SURVEY ON TRADITIONAL PLATFORMS AND NEW TRENDS IN PARALLEL COMPUTATION

Genci Berati, MSc
University of Tirana, Albania

Abstract

The information processing is in continuous progress. High Performance Computing is now a trend. The Parallel Computing is a synonymous phrase for High Performance Computing. Parallel Computing is the main field of information processing in our age. Both, the hardware systems and the software platforms are developing very fast to support the simple, rational and easy parallel data processing and programming. This paper shows an overview of issues and improvements in parallel processing. This paper deals with likewise the qualities and the favorable circumstances of distinctive stages for parallelism. Herewith are dealt with different architectures, innovations for parallelism and comparisons of results of parallel processing. The main question is to introduce the Dataflow model and some solid illustrations of Dataflow arrangement. The paper compares the traditional control flow parallel platform in contrasts to the data flow innovation.

Keywords: Control Flow Computing, Dataflow Computing, High Performance Computing (HPC), Parallel Computing Technologies

Introduction

Parallel techniques in traditional control flow computing, mostly are very powerful and effective, but metaphorically speaking, it is like heaving many expensive experts leading the process of work. Those expensive experts cost money, power, time and space. One can ask: Is it possible to do the same work by using more workers rather than using expensive experts. A new alternative, supporting the metaphor of too many workers instead of some expensive experts, for very fast parallel computing is considered the data flow computing. Data flow computing was developed about 30 years ago as a way of solving the parallel processing problem, but because of the difficulties in technological implementation this alternative was left away over time. But, during this decade the technology on parallelism is making a comeback. The data flow computing is experiencing a rebirth, so it is worthy to invest in possibilities to use the technology in large numerical calculation. Data flow computers would provide plenty of computing power. However, a direct implementation of computers based on the dataflow model has been found to be a monumental challenge (Ben Lee, A. R. Hurson, 1993). This paper presents a survey of characteristics in dataflow computing and a comparison to the old parallel computing techniques. Dataflow is an alternative that in our allegory can be compared with too many specialized workers instead of many expensive experts. Experts are expensive and slow, since the workers can be cheaper and faster. The dataflow approach of computation offers many advantages for parallel processing. The hardware implementation of this approach is very difficult, but nevertheless there are really good successful efforts and still these efforts are continuing now a days. Since the early 1970s, a number of hardware prototypes have been built and evaluated (J. R. Gurd, C. C. Kirkham, and I. Watson, 1985) and simulation studies of different architectural designs and compiling technologies have been performed (Veen, A. H., 1986). The experience gained from these efforts has led to progressive development in dataflow computing. However, there are many doubts and the question still remains as to whether the dataflow approach is a viable means for developing powerful computers to meet today’s and future computing demands (http://www.numericmethod.com).

Von Newman computing model

The Von Neumann or control flow computing model consists of a program which is a series of addressable instructions, each of which either specifies an operation along with memory locations of the operands or it specifies the transfer of control to some other instruction. Essentially, the next instruction to be executed depends on what happened during the execution of the current instruction. The next instruction to be executed is pointed to and triggered by the PC. The instruction is executed even if some of its operands are not available yet. (http://www.computerhope.com)

In this article we are talking about two Control Flow parallel platforms (OpenMP and MPI), which is the traditional way of processing versus Data flow platform which presents the modern alternative for the same purpose.

Sequential algorithms and parallelism in control flow

What is important in our discussion is the parallelism potential of such architecture. Let’s see some terminology and basic concepts of parallelism.

Flynn’s Taxonomy of Parallel Architectures

A parallel computer can be characterized as a collection of processing elements that can communicate and cooperate to solve large problems fast.

A simple model for describing the parallelism in control flow machines is given by Flynn's taxonomy (http://en.wikipedia.org). This taxonomy characterizes parallel computers, according to the global control and the resulting data and control flows.

There are four categories of architectures:

1. Single Instruction, Single Data (SISD): There is one processing element which has access to a single program and data storage. In each step, the processing element loads an instruction and the corresponding data and executes the instruction. The result is stored back in the data storage. Thus, SISD is the conventional sequential computer according to the von Neumann model.

2. Multiple Instruction, Single Data (MISD): There are multiple processing elements, each of which has a private program memory, but there is only one common access to a single global data memory. In each step, each processing element obtains the same data element from the data memory and loads an instruction from its private program memory. These possibly different instructions are then executed in parallel by the processing elements using the previously obtained (identical) data element as an operand. This execution model is very restrictive and no commercial parallel computer of this type has ever been built.

3. Single Instruction, Multiple Data (SIMD): There are multiple processing elements, each of which has a private access to a (shared or distributed) data memory, see Section 2.3 for a discussion of shared and distributed address spaces. But there is only one program memory from which a special control processor fetches and dispatches instructions. In each step, each processing element obtains from the control processor and the same instruction and loads a separate data element through its private data access on which the instruction is performed. Thus, the instruction is synchronously applied in parallel by all processing elements to different data elements.

4. Multiple Instruction, Multiple Data (MIMD): There are multiple processing elements, each of which has a separate instruction and data access to a (shared or distributed) program and data memory. In each step, each processing element load a separate instruction and a separate data element, apply the instruction to the data element, and stores a possible result back into the data storage. The processing elements work asynchronously with each other. Multicore processors or cluster systems are examples of the MIMD model. (http://en.wikipedia.org)

Platforms of parallelism in control flow computing (OpenMP, MPI)

OpenMP (Open Multi processing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN, on most processor architectures and operating systems, including Solaris, AIX, HP-UX, GNU/Linux, Mac OS X, and Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior (http://www.OpenMP.org). De facto standard API for writing shared memory parallel applications in C, C++, and Fortran OpenMP API consists of: Compiler Directives, Runtime subroutines/functions, Environment variables.(OpenMP Compilers, 2013)

OpenMP is managed by the nonprofit technology consortium OpenMP Architecture Review Board (or OpenMP ARB), jointly defined by a group of major computer hardware and software vendors, including AMD, IBM, Intel, Cray, HP, Fujitsu, Nvidia, NEC, Microsoft, Texas Instruments, Oracle Corporation, and more.(OpenMP Tutorial at Supercomputing 2008)

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

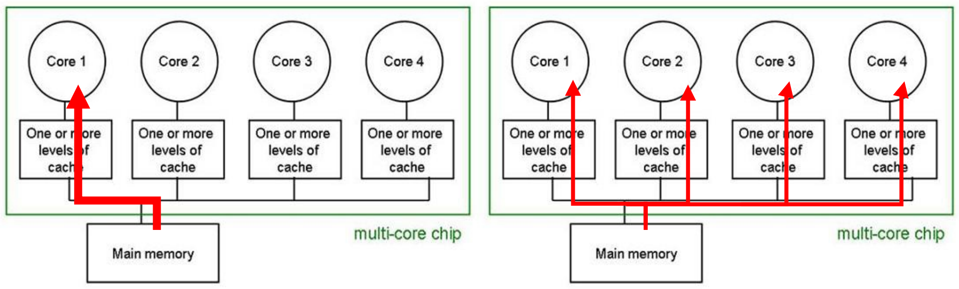
An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), or more transparently through the use of OpenMP extensions for non-shared memory systems.(http://people.sc.fsu.edu)

Matrix multiplication in c++ openmp

Let’s treat the C++ OpenMP parallelization introducing by using a simple example. Herewith includes an implementation of the matrix multiplication in OpenMP. The directive of processor #pragma omp parallel for default(none) shared(a,b,c) is the a simple modification of the normal source code in c++ which can parallelize all the loop below this directive. Let’s see the entire program for matrix multiplication.

```
/* Matrix_multiplication_in_openMP.cpp */
const int size = 1000;
float a[size][size];
float b[size][size];
float c[size][size];
int main()
{   // Initializing of matrix A, B, and C with zero values
   for (int i = 0; i < size; ++i) {
      for (int j = 0; j < size; ++j) {
         a[i][j] = (float)i + j;
         b[i][j] = (float)i - j;
```

```
            c[i][j] = 0.0f;
        }
    }
    // Perllogariten vlerat e elementeve te matrices
    // C <- C + A x B
    #pragma omp parallel for default(none) shared(a,b,c)
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            for (int k = 0; k < size; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
    return 0;
}
```



Figutre 1.a) Serial execution uses only one core b) ParaleleOpenMP execution

This platform works very well in shared memory systems like multi core computers. Let's suppose that our source code is executed on a shared memory machine. What happens in the source code after we add our parallelization directive? Let' suppose that our machine has 4 cores. A sequential algorithm uses just one core to accomplish the loop. While the use of the arallelization directive makes active all cores like in figure 1.

The directive, #pragma omp parallel for default(none) shared(a,b,c) does implement the parallelization process. The runtime creates 3 additional ”worker” threads at start of openmp parallel region. OpenMP programs start with a single thread; the master thread. At the start of the parallel region master creates a team of parallel “worker” threads (FORK). Statements in parallel block are executed in parallel by every thread. At the end of parallel region, all threads synchronize, and join the master thread (JOIN).

Mpi platform in C++

MPI is a directory of C++ programs which illustrate the use of the Message Passing Interface for parallel programming. MPI is a library of message passing routines. The library allows a user to write a program in a familiar language, such as C, C++, FORTRAN77 or FORTRAN90, and carry out a computation in parallel on an arbitrary number of cooperating computers. This platform is used in distributed memory parallel systems.

Herewith is a simple example which uses the MPI library, which executes from each core an execution thread for matrix multiplication loops:

```
/* shumezim_matricash_MPI.cpp */
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define TAG 13
int main(int argc, char *argv[]) {
  double **A, **B, **C, *tmp;
  int numElements, offset, stripSize, myrank, numnodes, N, i, j, k;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &numnodes);
  N = atoi(argv[1]);
  if (myrank == 0) {
    tmp = (double *) malloc (sizeof(double ) * N * N);
    A = (double **) malloc (sizeof(double *) * N);
    for (i = 0; i < N; i++)
      A[i] = &tmp[i * N];
  }
  else {
    tmp = (double *) malloc (sizeof(double ) * N * N / numnodes);
    A = (double **) malloc (sizeof(double *) * N / numnodes);
    for (i = 0; i < N / numnodes; i++)
      A[i] = &tmp[i * N];
  }
  tmp = (double *) malloc (sizeof(double ) * N * N);
  B = (double **) malloc (sizeof(double *) * N);
  for (i = 0; i < N; i++)
    B[i] = &tmp[i * N];
  if (myrank == 0) {
    tmp = (double *) malloc (sizeof(double ) * N * N);
    C = (double **) malloc (sizeof(double *) * N);
    for (i = 0; i < N; i++)
      C[i] = &tmp[i * N];
  }
  else {
    tmp = (double *) malloc (sizeof(double ) * N * N / numnodes);
    C = (double **) malloc (sizeof(double *) * N / numnodes);
    for (i = 0; i < N / numnodes; i++)
      C[i] = &tmp[i * N];
  }
  if (myrank == 0) {
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        A[i][j] = 1.0;
        B[i][j] = 1.0;
      }
    }
  }
  stripSize = N/numnodes;
  if (myrank == 0) {
    offset = stripSize;
    numElements = stripSize * N;
    for (i=1; i<numnodes; i++) {
      MPI_Send(A[offset], numElements, MPI_DOUBLE, i, TAG, MPI_COMM_WORLD);
      offset += stripSize;
    }
  }
  else {
    MPI_Recv(A[0], stripSize * N, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
  }
  MPI_Bcast(B[0], N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  for (i=0; i<stripSize; i++) {
    for (j=0; j<N; j++) {
      C[i][j] = 0.0;
    }
  }
  for (i=0; i<stripSize; i++) {
    for (j=0; j<N; j++) {
      for (k=0; k<N; k++) {
             C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
  if (myrank == 0) {
    offset = stripSize;
    numElements = stripSize * N;
    for (i=1; i<numnodes; i++) {
      MPI_Recv(C[offset], numElements, MPI_DOUBLE, i, TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      offset += stripSize;
    }
  }
  else {
    MPI_Send(C[0], stripSize * N, MPI_DOUBLE, 0, TAG, MPI_COMM_WORLD);
  }
  if (myrank == 0 && N < 10) {
    for (i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        printf("%f ", C[i][j]);
      }
      printf("\n");
    }
  }
  MPI_Finalize();
  return 0;
}
```

Modern alternative architectures for parallel computing (dataflow)

Dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. Dataflow architectures do not have a program counter, or (at least conceptually) the executability and execution of instructions are solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable: I. e. behavior is undetermined. Although no commercially successful general-purpose computer hardware has used dataflow architecture, it has been successfully implemented in specialized hardware such as digital signal processing, network routing, graphics processing, telemetry, and more recently in data warehousing. It is also very relevant in many software architectures today, including database engine designs and parallel computing frameworks.

Synchronous dataflow architectures tune to match the workload presented by real-time data path applications such as wire speed packet forwarding. Dataflow architectures that are deterministic in nature enable programmers to manage complex tasks such as processor load balancing, synchronization and accesses to common resources.(EN-Genius, June 2008)

Meanwhile, there is a clash of terminology, since the term Dataflow is used for a Subarea of parallel programming: for dataflow programming.

The execution is driven only by the availability of the operand! No Program Counter is used and global updateable store, which are the two features of von Neumann model that become a challenge in exploiting parallelism are missing in DataFlow architecture.

The execution algorithm of the dataflow instructions in pseudocode can be:

```
WHILE(AVAILABLE_OPERATIONS
(STATE)) {
STATE = EXEC(AVAILABLEOPERATION(
STATE),STATE)
}
OPERATIONS “FIRE” WHEN
ALL INPUTS ARE AVAIALBLE
```

One of the successful implementations of the dataflow philosophy is Maxeler Data Flow Engine(http://www.maxeler.com/). The implementation of the algorithm needs to modify the C source code, to write one or more kernel files, one (or more) ava file, one manager file for transforming the kernel(s), simulator builder, hardware builder and a series of default programs for transferring the code to data flow engine module.

A modified Java program for execution in dataflow architecture is:

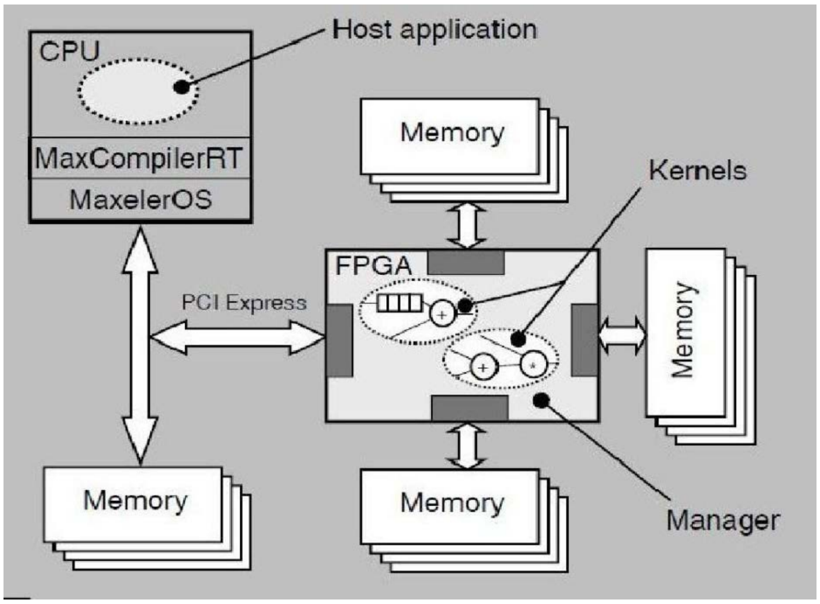


Figure 2. Execution in data flow engine

Figure 3. Example of a Java kernel progra

Conclusion

Parallel computing is a trend of our age and is pushed and forced into the computing paradigms just as Object Oriented was pushed in the previous millennium into the programming issues. There are several types of parallel computing, regards to the hardware architectures (shared or distributed memory systems and data flow architecture systems). The hardware is parallel so the Kernel is parallel. There are many levels of difficulties in the parallelism. Some problems do not have work-efficient parallel algorithms that allow the effective parallelism. Some of the parallel algorithms do not have the same level of numerical stability as well-known sequential algorithms. It is needed a very careful benchmarking process to be secure for the effectiveness of the chosen architecture for parallelization, because some time choosing an inappropriate platform can yield to a failure parallelism.

As final conclusion, I would like to stress the fact that the DataFlow computing is the future of parallelism in data processing.

References:

Ben Lee, A. R. Hurson, 1993: ISSUES IN DATAFLOW COMPUTING: In Corvallis, Oregon University Press 97331-3211, page 2

J. R. Gurd, C. C. Kirkham, and I. Watson, 1985: The Manchester Prototype Data-Flow Computer: In journal "Communications of the ACM", Vol. 28, page 34

Veen, A. H., 1986: "Dataflow Machine Architecture: In journal "ACM Computing Surveys", Vol. 18, No. 4, page 365

Retrieved on 14 October 2014 from: http://www.numericmethod.com/About-numerical-methods/system-of-linear-equations/gauss-elimination (Updated on 2014, October 13)

Retrieved on 11 October 2014 from:http://www.computerhope.com/jargon/c/contflow.htm

Retrieved on 11 October 2014 from: http://en.wikipedia.org/wiki/Flynn's_taxonomy

"OpenMP Compilers". OpenMP.org. 2013-04-10. Retrieved 2013-08-14.

OpenMP Tutorial at Supercomputing 2008

Retrived on 15 January 2015 from: http://people.sc.fsu.edu/~jburkardt/cpp_src/mpi/mpi.html

"HX300 Family of NPUs and Programmable Ethernet Switches to the Fiber Access Market", EN-Genius, June 18 2008.

Retrieved on 11 January 2015 from: http://www.maxeler.com/technology/