

BMDFM – Binary Modular DataFlow Machine

The Mission Statement

Motivation

Nowadays parallel shared memory symmetric multiprocessors (SMP) are complex machines, where a large number of architectural aspects have to be simultaneously addressed in order to achieve high performance. Recent commodity SMP machines for technical computing can have up to 64 tightly coupled cores (good examples are IBM SMP machines based on POWER multi-core processor and SUN SMP machines based on SPARC and AMD multi-core processors). The number of cores per SMP node probably will be doubling next couple of years according to the unveiled plans of computer manufacturing companies.

Multi-cores are intended to exploit a thread-level parallelism, identified by software. Hence, the most challenging task is to find an efficient way of how to harness power of multi-core processors for processing an application program in parallel. Existent OpenMP paradigm of the static parallelization with a fork-join runtime library works pretty well for loop-intensive regular array-based computations only, however, compile-time parallelization methods are weak in general and almost inapplicable for irregular applications:

- There are many operations that take a non-deterministic amount of time making it difficult to know exactly when certain pieces of data will become available.
- A memory hierarchy with multi-level caches has unpredictable access latencies.
- A multi-user mode other people's codes can use up resources or slow down a part of the computation in a way that the compiler cannot account for.
- Compile-time inter-procedural and cross-conditional optimizations are hard (very often impossible) because compilers cannot figure out which way a conditional will go or cannot optimize across a function call.

BMDFM and transparent dataflow semantics

A novel BMDFM (Binary Modular DataFlow Machine) technology mainly uses dynamic scheduling to exploit parallelism of an application program, thus, BMDFM avoids mentioned disadvantages of the compile-time methods. BMDFM is a parallel programming environment for multi-core SMP that provides:

- Conventional programming paradigm requiring no directives for parallel execution.
- Transparent (implicit) exploitation of parallelism in a natural and load balanced manner using all available multi-core processors in the system automatically.

BMDFM combines the advantages of known architectural principles into a single hybrid architecture that is able to exploit implicit parallelism of the applications having negligible dynamic scheduling overhead and no bottlenecks. Mainly, the basic dataflow principle is used. The dataflow principle says: "An instruction or a function can be executed as soon as all its arguments are ready. A dataflow machine manages the tags for every piece of data at runtime. Data is marked with ready tag when data has been computed. Instructions with ready arguments get executed marking their result data ready".

The main feature of BMDFM is to provide a conventional programming paradigm at the top level, so-called transparent dataflow semantics. A user understands BMDFM as a virtual machine, which runs all statements of an application program in parallel having all parallelization and synchronization mechanisms fully transparent. The statements of an application program are normal operators, which any single threaded program might consist of - they include variable assignments, conditional processing, loops, function calls, etc.

Suppose we have the code fragment shown below:

```
a = udf0(i); // udf stands for User Defined Function
b = udf1(i+1);
b++;
printf("a = %d\n",a);
printf("b = %d\n",b);
```

The two first statements are independent, so the dataflow engine of BMDFM can run them on different processors or processor's cores. The two last statements can also run in parallel but only after "a" and "b" are computed. The dataflow engine recognizes dependencies automatically because of its ability to build a dataflow graph dynamically at runtime.

Additionally, the dataflow engine correctly orders the output stream to output the results sequentially. Thus even after the out-of-order processing the results will appear in a natural way.

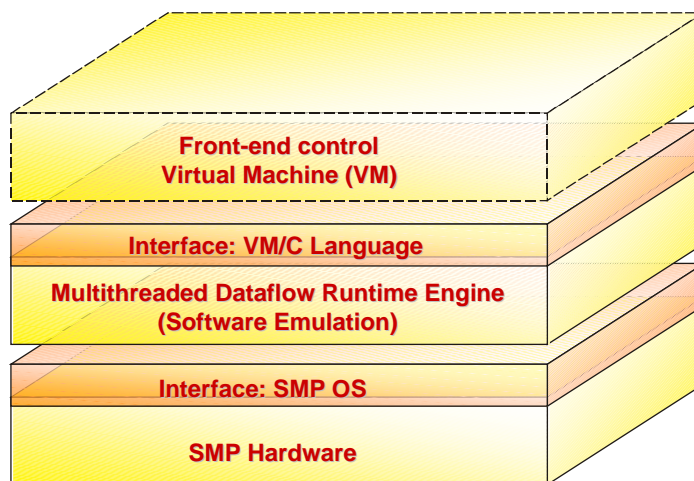
Suppose that above code fragment now is nested in a loop:

```
for(i=1;i<=N;i++){
  a = udf0(i); // udf stands for User Defined Function
  b = udf1(i+1);
  b++;
  printf("a = %d\n",a);
  printf("b = %d\n",b);
}
```

The dataflow engine of BMDFM will keep variables “a” and “b” under unique contexts for every iteration. Actually, these are different copies of the variables. A context variable exists until it is referenced by instruction consumers. Later non-referenced contexts will be garbage collected at runtime. Therefore the dataflow engine can exploit both local parallelism within the iteration and global parallelism as well running multiple iterations simultaneously.

Architecture of BMDFM

The basic concept of BMDFM is shown in the next Figure. The proposed approach relies on underlying commodity SMP hardware, which is available on the market. Normally, SMP vendors provide their own SMP Operating System (OS) with an SVR4 UNIX interface (Linux, HP-UX, SunOS/Solaris, Tru64OSF1, IRIX, AIX, MacOS, etc.). On top of an SMP OS, the multithreaded dataflow runtime engine performs a software emulation of the dataflow machine. Such a virtual machine has interfaces to the virtual machine language and to C providing the transparent dataflow semantics for conventional programming.



Basic concept of BMDFM

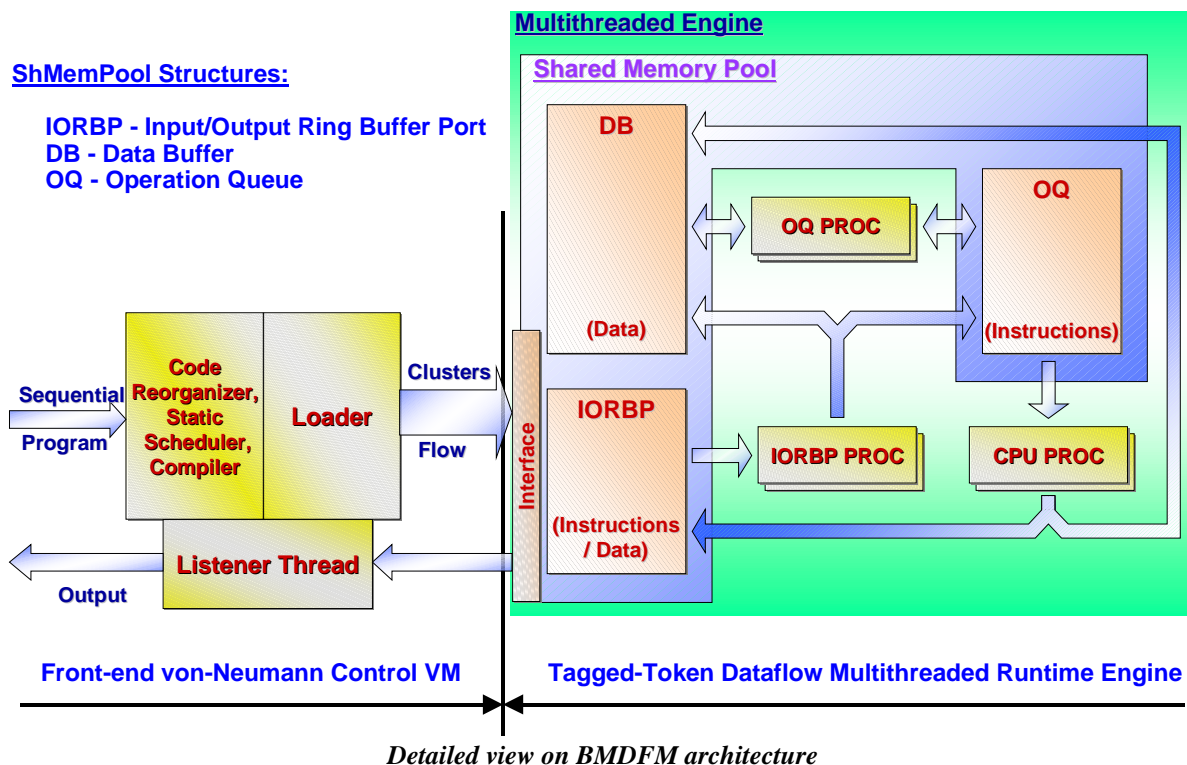
As it can be seen, BMDFM is built as a hybrid of several architectural principles:

- MIMD (Multiple Instruction Streams, Multiple Data Streams), which is sustained by commodity SMP.
- Implicit parallelism is ensured by dataflow emulation.
- Von-Neumann computational principle is good to implement the Front-end Control Virtual Machine.

The next Figure presents more detailed view on the BMDFM architecture. An application program (input sequential program) is processed in three stages: preliminary code reorganization (code reorganizer), static scheduling of the statements (static scheduler) and compiling/loading (compiler, loader). The output after the static scheduling stages is a multiple clusters flow that feeds the multithreaded engine via the interface designed in a way to avoid bottlenecks. The multiple clusters flow can be thought of as a compiled input program split into marshaled clusters, in which all addresses are resolved and extended with context information. Splitting into marshaled clusters allows loading them multithreadedly. Context information lets iterations be processed in parallel. Listener thread orders the output stream after the out-of-order processing.

The BMDFM dynamic scheduling subsystem is an efficient SMP emulator of the tagged-token dataflow machine. The Shared Memory Pool is divided in three main parts: Input/Output Ring Buffer Port (IORBP), Data Buffer (DB) and Operation Queue (OQ). The Front-end Control Virtual Machine schedules an input application program statically and puts clustered instructions and data of the input program into the IORBP. The ring buffer service processes (IORBP PROC)

move data into the DB and instructions into the OQ. The operation queue service processes (OQ PROC) tag the instructions as ready for execution if the required operands' data is accessible. Execution processes (CPU PROC) execute instructions, which are tagged as ready and output computed data into the DB or to the IORBP. Additionally, IORBP PROC and OQ PROC are responsible for freeing memory after contexts have been processed. The context is a special unique identifier representing a copy of data within different iteration bodies accordingly to the tagged-token dataflow architecture. This allows the dynamic scheduler to handle several iterations in parallel.



Running under an SMP OS, the processes will occupy all available real machine processors and processor's cores. To allow several processes accessing the same data concurrently, the BMDFM dynamic scheduler locks objects in the shared memory pool via SVR4 semaphore operations. Locking policy provides multiple read-only access and exclusive access for modification.

Conclusions

The dataflow systems are probably worth another look at this time. The community has gone through a shared-distributed-shared hardware implementation cycle since the peak of dataflow activity and applying some of what have been learned in that time to software based systems seems appropriate.

BMDFM is a convenient parallel programming environment and an efficient runtime engine for multi-core SMP due to the MIMD unification of several architectural paradigms (von-Neumann, SMP and dataflow):

- At first, it is a hybrid dataflow emulator running multithreadedly on commodity SMP. The SMP ensures MIMD while dataflow exploits implicit parallelism.
- At second, it is a hybrid multithreaded dataflow runtime engine controlled by a von-Neumann front-end VM. The dataflow runtime engine executes tagged-token contextual parallel instructions (opposite to the restricted fork-join paradigm) while the von-Neumann front-end VM initializes contexts and feeds the dataflow runtime engine with marshaled clusters of instructions.
- At third, it is a hybrid of static and dynamic parallelization. The von-Neumann front-end VM tries statically to split an application into parallel marshaled clusters of instructions while the dataflow runtime engine compliments to the static parallelization methods dynamically.

BMDFM is intended for use in a role of the parallel runtime engine (instead of conventional fork-join runtime library) able to run irregular applications automatically in parallel. Due to the transparent dataflow semantics on top, BMDFM is a simple parallelization technique for application programmers and, at the same time, is a much better parallel programming/compiling technology for multi-core SMP computers. Visit <http://bmdfm.com> site for more information and free downloads for most available SMP platforms.