A Shared Memory Model on a Cluster of PCs

Dominique Houzet
IRIT-ENSEEIHT-INP
2, rue Camichel, 31071 Toulouse, France
E-mail: houzet@enseeiht.fr

Abstract

The GFLOPS project's aim is to develop a parallel architecture as well as its software environment to implement scientific applications efficiently. This goal can be achieved only with a real collaboration among the architecture, the compiler and the programming language. This paper investigates the C// Parallel language and the underlying programming model on a global address space architecture. The main advantage of our paradigm is that it allows a unique framework to express and optimize both data and control parallelism at user level. The evaluation of C// has been conducted on a cluster of PCs.

1. Introduction

The GFLOPS computer [1] [2], which is a cluster of PCs, provides an elaborate shell to support global memory access, prefetch, atomic operations, barriers, and block transfers. Based on this design, a simple but powerful parallel extension to the C language has been designed with the goal of extracting the full performance capability out of the target machine. The basic approach is to provide a full C on every node, with a rich set of assignment operations on the collective global address space. With GFLOPS, the semantics of the hardware primitives for global operations are essentially at the same level as the language primitives. The programming environment includes a globally shared address space, a compiler that automatically partitions regular programs with loops, a library of efficient synchronization and communication routines and a parallel debugger. This work originated from the SYMPATI project. This first project led to the joint development of the Line Processor SYMPATI2 [3] developed by our laboratory and the CEALETI-Saclay French Nuclear Agency. SYMPATI2 provides excellent performances as shown through our response to Preston's Abingdon Cross benchmark [4].

This paper describes the experiences gained by jointly designing a high level programming language and a complete parallel system. Section 2 outlines the GFLOPS's implementation. Section 3 describes the C// language and programming environment. Section 4 presents macrobenchmarks to evaluate overall performance of this environment on the GFLOPS machine and illustrate the benefits of the approach.

2. The GFLOPS architecture

![Figure 1. The GFLOPS architecture.](image-url)
The GFLOPS architecture is organized as shown in figure 1. Memory is physically distributed over the processing nodes. The machine has a single address space consisting of a large number of processing nodes connected to a low-latency, high-bandwidth interconnection network. 16 nodes form a first-level ring. A communication cell, on the right of the figure, is used to build a larger configuration with up to 512 processors organized as a hierarchical multiring structure like KSR1 [5]. The distant communications are performed with messages. The messages use the FIFO in each network module to reach the target memory. The routing protocol used is a wormhole like protocol [6].

Although GFLOPS provides the abstraction of globally shared memory to programmers, the system's physical memory is statically distributed over the nodes in the machine. Thus for performance reasons, much of the underlying software is implemented using message passing. The performances of all the layers of software that help manage locality (including the compiler, libraries and runtime system) depend on an efficient communication mechanism. Features in the PCI bus and the network module are combined to provide a streamlined interface for transmitting and receiving messages: both driver and user code can quickly describe and atomically launch a message directly into the interconnection network; a direct memory access mechanism allows data to flow between the network and memory.

All synchronization capabilities are available with instructions such as "barrier", "test and increment" and "wait" on the completion of all the distant accesses. Overlapping of remote accesses (software prefetching) with treatment of data is performed at compile time for basic blocks and for loops. This mechanism attempts to tolerate the latency of interprocessor communication when it cannot be avoided. Prefetching allows code to anticipate communication by requesting data or locks before they are needed.

The data coherence is an important aspect of shared memory parallel architectures. In the case of this first version of the GFLOPS network architecture, the data coherency is not hardwired in the network. The first reason is to simplify the hardware. The most important reason is that GFLOPS architecture uses PCI bus to communicate, which doesn't maintain the coherency of data. So it is not possible with such motherboards to maintain the coherency of data by hardware. It is not a critical point because in the case of realtime image processing and other parallel application fields, a way to optimize the algorithms is to program and optimize manually the data exchanges between the different processors. The data coherency is then automatically managed by the data transfer functions. The data coherency is only hardwired in the motherboards, but not among them. In other words, a data from a DRAM cannot be cached in a processor from another motherboard. A remote data will be kept locally only if it is stored in a local variable. This is done by the remote memory transfer functions incorporated in the program either by the programmer or the compiler. The coherency is managed with synchronization operations at the end of the transfers, to guarantee the validity of the transfers. The programming model implemented on this architecture provides these functions to the programmer to manage manually the data coherency to optimize communications.

3. The C// parallel language

The software part of the GFLOPS project provides different development tools. The first is a complete systemlevel simulator, written in C. An assembler language and a C++ compiler are associated with the simulator/debugger to help for program debugging and performance tuning. The entire system runs real applications and enables us to verify protocols, analyse system performance, and identify architectural bottlenecks. The C++ compiler is a modified version of the GNUCC compiler [7] configured for our architecture. At the C++ programming level, the declarations and data structure accesses must be described taking into consideration the real number of processors. This number is virtualized only by the C dataparallel language (C//) and is computed for each parallel loop according to the size of the array scanned.

Although the fastmessage capability of GFLOPS makes it a good vehicle to execute programs written in a message-passing style, it is better viewed by the programmer as a shared-memory machine. Many parallel object oriented implementations based on C/C++ have been proposed like Dataparallel C [8], splitC [9], C** [10], Compositionnal C [11], or HPC[12].

The C// language is a parallel extension of the C++ language. It is a small extension of C++ which significantly reduces the learning time for a C++ programmer, but offers powerful mechanisms for parallel programming and data partitioning. It is dedicated to synchronous and asynchronous shared memory parallel computers. It is designed in such way
that keeps the simplicity of programming and takes
advantage of the hardware and software resources of the
target machine. The application field aimed is the
scientific field where data parallelism is the most
important. The C choice makes the portability of the
applications on different architectures possible. The C//
language is based on two main concepts, an
asynchronous control flow on each processor
(asynchronous SPMD model as in the SplitC language
[9]) and a double sight of the memory space (a local and
a global memory access). C// maps a parallel processing
entity on a virtual processor and thus provides a
meaningful abstraction of the number of processors
(NPE). Each processor is identified by a single value
(MY_ID) from 0 to NPE-1. Each processor has its own
control flow. Synchronization points are used to
organize the different control flows in the machine. The
programming style is SPMD which allows both SIMD
and MIMD paradigms to be used. The C// language is
well adapted to regular data structures (arrays) and
irregular structures (graphs). Those structures are well
managed with the powerful pointer capabilities of the C
language. The C array-pointer duality allows an indexed
access to the arrays with the use of the arithmetic on
pointers. This duality is extended to parallel array
pointers in C//. The access to a distributed array
uses the arithmetic on global pointers. The global
address space is two dimensional from the viewpoint
of address arithmetic on global data structures and from a
performance viewpoint, in that each processor has
efficient access to a portion of the address space
(NUMA) located in its own memory bank (MY_ID).
The user can optimize the mapping of data structures on
the architecture with a simple and powerful declaration
syntax.

Figure 2 shows the compile steps from the C// source
code to the binary code. The first step is to write a C
code extended with parallel data and control constructs.
This code is then translated to a pure C ANSI code by
the C// compiler (made with LEX&YACC). We have
ported the GNU CC compiler to generate the assembler
code for different architectures, namely GFLOPS [2]
and SYMPHONIE [13]. Every stage of the compilation
integrates the target architecture dependent
optimizations.
The C++ language has the following salient features:

- A program is comprised of a thread of control on each processor from a single code image.

- Threads interact through reads and writes on shared data referenced by global pointers, and through systolic communication among neighboring processors. The compiler can syntactically distinguish local accesses from global accesses.

- Split-phase variants of read and write are provided to allow the long latency of remote access to be masked. They correspond to data prefetching at compile time [14]. This is done automatically for loops using remote reads or writes. A synchronization statement waits for the completion of all the pending reads or writes.

- Bulk transfer within the global address space can be specified in either blocking or non-blocking forms.

- Remote reads are changed in remote writes to improve oneway communication in those algorithms where the communication pattern is known in advance. It is a kind of data forwarding technique implemented at compile time. Threads can synchronize on the completion of a phase of remote writes, as in data-parallel programs.

- Both explicit and implicit data-parallel mapping and access are available through a simple and powerful declaration of arrays. It is possible either to control explicitly the mapping and access to the data, according to the target architecture, or to leave the compiler to perform a static mapping.

### 3.1. Data types used

C++ uses two types of data, the scalar and parallel types. All the standard C types are recognized (void, int, char ...). If the target machine has a scalar processor, all the scalar variables will be processed by this processor, and broadcast to the other processors if needed. Otherwise the scalar variables are duplicated on all the processors, to define a regular memory organization, and allow fast memory accesses on those variables. Function parameters and local variables are stored in each processor local stack (which is private).

#### 3.1.1. Parallel arrays

Parallel variables are spread among the memory banks. Global pointers are used to address the global shared address space. Each processor has only a part of the parallel variable in its memory bank. Its size corresponds to the whole size divided by the number of processors. But each processor can have access to the entire variable. Many different mappings of parallel variables are available through a concise and powerful syntax declaration:

```
Type ([size:] {dist} {bord}) + name ([m]) *;
```

{expr} is an optional expression. ()+ is repeated one or more times. ()* is repeated zero or more times. The type can match all the standard C types. LD (Left descriptor) defines the dimensions distributed among the virtual processors. The real number of processors is a symbolic value NPE. A parallel variable is used as if there were as many virtual processors as the size of the distributed array. The mapping of this array on the architecture is performed statically at compile time. The ':' character specifies the physically distributed dimensions. At least one of the distributed dimensions must be a multiple of NPE in order to keep a regular mapping on all the memory banks. The size of each dimension is the only mandatory parameter. Several optional parameters, having default values, are used to define the explicit mapping of the array variable on the architecture. The dist parameter defines the way to distribute each dimension: in a cyclic way (default value) or with consecutive blocks of a given size. The edge parameter defines the way to scan the edges of the array: like a mesh or like a torus. RD (Right Descriptor) defines the distribution unit of the array among the memory banks. All the dimensions on the right of the declaration are used as a block which is local for each virtual processor.

The RD descriptor calculate the address in the local block unit of the array, and the LD descriptor calculate the place of the block in the global address space. Address arithmetic on RD is local to a memory bank. Arithmetic on LD is global among the memory banks.

#### 3.1.2. Global pointers

Global pointers address the whole global memory space. A pointed object is entirely located in a memory bank. The declaration is obtained with the *overall key word:

```
Type *overall ([Dim]) Name;
```

The Dim parameter corresponds to the size of the pointed object. The Name pointer points to a given memory bank and a given object in this memory bank (a 2D definition). A memory access with a global pointer is the same as an array reference:

```
[LD]Name[RD];
```
A global pointer reference may be manipulated by performing address arithmetic of two forms. Local addressing treats the global address space as segmented among processors and acts on a global pointer as on a standard C pointer, i.e., an incremented pointer refers to the next location on the same processor. Global addressing treats the global address space as linear with the processor component varying fastest. Addresses wrap around from the last processor to the next offset on the first processor. C// extends the C array-pointer duality to the parallel variables. In the following example:

```c
int [256:][256:]Matrix[3][3] ;
```

Matrix is a global pointer. Its initial value is the address of the first block of 9 integers located in the local memory bank (MY_ID) of each processor.

### 3.2. Parallel object mapping

Both explicit and implicit data-parallel mapping and access are available through a simple and powerful declaration of arrays. The explicit mapping is used to optimize the declaration and the access to a parallel variable according to the target architecture. This explicit mapping on the target machine is static. There is a compromise between full virtual mapping of a parallel variable (which can be complex for the compiler) and full explicit mapping (which can be complex for the programmer). This mapping is controlled with the LD descriptor of the parallel array declaration. In this way, it is possible to minimize the static communications between processors according to the structure of the target architecture. As an example, a 256x256 matrix mapped on a 64-processor ring, with treatments using communications on the matrix lines, can be declared as:

```c
int [256:block][256]Matrix;
```

Each processor has a block of 256/64 consecutive lines of 256 points in its local memory bank. The C corresponding declaration is: int Matrix[4][256];

### 3.3. Data parallel Control

The most important parallel control structure is the parallel loop "forall":

```c
forall (Parallel_object [PatternDims]*)
{ Statements ; }
```

The parallel object structure corresponds to the loop. The pattern defines the way to scan each dimension of the parallel variable.

```c
Pattern_dims:: ; Ind, Begin, End, Step
; ;
```

A not scanned dimension is specified with ":". The default values of the parameters are ":" for Ind, 0 for Begin, the dimension's length for End and 1 for Step. Thus ":" refers to the current position in the loop:

```c
int [256:][256]Matrix ;
forall (Matrix) [.] [.] Matrix = ...
```

The C// compiler translates this parallel construct in a sequential one, which incorporates an inner loop dedicated to the processor number virtualization. The parallel address calculation is also managed by the C// compiler.

Several synchronization routines are available to synchronize the different control flows in the machine.

### 3.4. Data coherency

The SPMD dataparallel programming model used allows user-level management of data coherency with the use of parallel language features. These features are primitives of synchronization and primitives of communication. Such a special programming model has the potential to provide superior performance because it provide the coherence system with information not available in more general-purpose systems. We have tried to evaluate with different examples the extra effort required of programmers, which seems to be an acceptable burden. This evaluation has been conducted mainly with regular Image Processing algorithms. This burden could be more important with highly irregular algorithms. Also, this efficiency will be effective when the primitives are tightly coupled with the network hardware interface. In our project, we have designed both the hardware and the software in order to obtain a nearly optimal solution.

#### 3.4.1. Communication

The first primitives are bloc transfer primitives like in SplitC [9]. These primitives are used to anticipate memory accesses with bloc transfers in order to overlap communications with computation. These primitives are constructs integrated in the C// language like the Forall() parallel loop construct. They define the scanning of a parallel object to be transferred:
These three operations are loop constructs managing block transfers when data are not consecutive. They operate like the Forall() construct with the virtualization of the number of processors. The hardware implementation uses deliberate update of remote data. The data coherency will be managed with synchronization. Here is an example of the matrix multiply in C// which broadcast line by line:

```c
/* Matrix A, B and C are mapped by column */
forall (A ; i ; j) {
  /* Software Prefetching of one line of the matrix A */
broadcast (A ; : ; ) ;
  result=0;
  for(k=0; k<256; k++) /* Local Accesses */
    result += [i][k]A * [k][j]B ;
  [i][j]C = result ;
}
```

The second communication available is to use fine grain memory access. A remote data is accessed through the network seen as a global address space. The virtual address is converted into a real address by the MMU and provided to the network. The network is thus used as a kind of PCItoPCI bridge from the local board to the remote board. A single memory access generates a communication in the network. The overhead of access to the network is thus limited to a single access. A typical block transfer requires three network accesses to configure the size and addresses of the communication. With small configurations, the overhead of access to the network is the main part of the network latency. Thus with a limited additional cost it is possible to improve fine grain communications by reducing remote latency. Also, as shown by Karlsson [15], the limiting factor in performance with distributed architectures is the latency rather than the bandwidth of message-level interconnect.

### 3.4.2. Synchronization

The synchronization operations are used to manage data coherency. The operations available in our language are implemented as a library of synchronization functions. These functions must be implemented in the network hardware in order to obtain efficient coherency. Here are the main operations which are hardwired in the GFLOPS network:

- The first mechanism provided to implement producer-consumer style synchronization is the remote fetch&increment operation implementing locks. This operation can be performed on any address in the global address space. This access hardwired in the network module, operates a fetch&increment (lock_inc() function) or a fetch&add (lock_add() function) on the accessed variable. The fetch&increment operation is essentially the cost of a remote read. The exclusive access is guaranteed by the remote network module which lock the remote address for all the other accesses from the network. Even for a local variable, the access is performed by the network module to guarantee the exclusivity of the access.

- `synchro()` which operates a global synchronization. This second mechanism is a global hardwired barrier synchronization, performed in 3 clock cycles in the GFLOPS network when all the threads call the barrier at the same time. Software routines implemented in a scan style, can be used to synchronize several sets of threads simultaneously. This global synchronization barrier is for all the processors from a mask, implemented with a wired-OR of synchronization signals (one per network module). This barrier is used to start or finish a treatment together.

- `wait_write()` which waits on the completion of all the previous writes. This third mechanism is a flag used to signal one thread that all the remote write accesses are ended. The software dataforwarding technique can be implemented with this flag by changing remote reads into remote writes and waiting for the end of their completion. This is possible only in those algorithms, where the communication pattern is known in advance. In that case, when a processor has to perform a remote read, it is the remote processor that reads the data and writes it in the remote memory. Then the first processor can read the data. This transform is performed at compile time with regular algorithms.

- `wait_read()` which waits on the completion of all the previous reads. This mechanism uses a flag in each network module which is set by a counter of ended remote reads. For instance, this flag is used to implement blocking remote reads, and software dataforwarding by anticipating the remote reads, and waiting for their completion to treat them.

- `int(PE)` which generates an interrupt to a remote processor used to implement signals. A processor interrupt is generated from the network module, to signal the end of several remote reads, or to signal the end of
several remote writes to the target processor. These interrupts are used to validate the coherency of data at the end of one or more transfers, and are incorporated in the communication functions for block transfers. Single data accesses coherency must be managed by means of global synchronization. All these primitives are directly accessible to the programmer with runtime library functions. It is possible, with such functions, to implement a message passing style of programming like PVM or MPI. This is possible with the use of interrupts at the end of transfers, which corresponds exactly to the message passing “send” and “receive” communication functions. Thus, a single architecture can implement efficiently both programming models: shared memory and message passing.

It is possible, with these synchronization functions, to manage data coherency, either directly by the programmer or by the compiler. In some cases, it may be possible for an optimizing compiler to obtain the performance of this special programming model without the special syntax [16]. In fact, the programming effort introduced to manage data coherency with applications such as image processing is quite limited. It can be more important for very irregular applications. It is the reason why we are studying now the implementation of data coherency management directly inside the compiler.

4. Performance evaluation

We have implemented a number of scientific and Image processing programs on GFLOPS and profiled their execution to determine the C// global performance. The matrix size used in the algorithms is 256x256. We present in table 1 several results of low-level and intermediate-level image processing algorithms in order to illustrate the performance of both the architecture and the software environment. The images have 256x256 points and the configuration is with 16 processors.

These results were obtained by extrapolation from the 2-processor prototype results and from the network simulation results. This was possible because those algorithms are predictable. We have added an overhead of communication delay to the 2-processor results. This overhead is calculated manually according to the amount of communications and to the delay introduced by a 16-processor network which is obtained with VHDL simulations.

The Hough transform operates on the Sobel edge extraction image which has 5% of edge pixels. The source image is 8-bit gray level image. We have used those algorithms to implement a surgery real-time image processing application. In this application, the images are captured by an optical fiber and the aim is to keep the image stable on the screen. The first step is a curve detection with the generalized Hough transform to obtain the angle of the image and the second step is then the rotation and translation of the image.

We have also compared the impact of the parallel cost introduced by the parallel compiler compared with a sequential compiler. We have evaluated for this purpose different image processing algorithms on a sequential machine, programmed with both a sequential program and a parallel program. We have used the same GNU-C compiler and the same operating system and the same machine. Thus the difference is the cost of the parallel compiler which generates shared memory address calculation in the parallel loops. The results obtained presented in table 2 show that the impact is not important when the array index operations are not complex. For more complex operations such as the image rotation, the impact can be more important if the treatment on data is limited compared to the communication. In that case the impact of address calculation is important. It is also possible to improve the C// compiler to obtain better parallel address calculations. Also, if the index treatment is adapted to the GNU-C compiler optimizations, we can obtain better results, which seems to be surprising. It depends mostly on the GNU-C behavior.

We have implemented this environment on SHVS (Standard High Volume Servers) with two and four processors (Pentium-Pro and UltraSparc). We have used multithread libraries with Linux SMP operating system and solaris 2.6 on SUN servers. The impact of the C// environment is quite similar to the previous study on GFLOPS. The main performance reduction with SVHS is not due to the environment but to their architecture with a shared common memory bus which is saturated when we implement shared memory applications.
Table 2. Impact of the parallel constructs on the execution time with GFLOPS.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>1 processor seq. time</th>
<th>1 processor parallel time</th>
<th>overhead of time</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotation</td>
<td>21 ms</td>
<td>28 ms</td>
<td>33 %</td>
</tr>
<tr>
<td>binarisation</td>
<td>17 ms</td>
<td>17 ms</td>
<td>0 %</td>
</tr>
<tr>
<td>transposition</td>
<td>9.6 ms</td>
<td>19.3 ms</td>
<td>7 %</td>
</tr>
<tr>
<td>Hough transform</td>
<td>275 ms</td>
<td>206 ms</td>
<td>-25 %</td>
</tr>
<tr>
<td>Sobel 3*3 filter</td>
<td>40 ms</td>
<td>40 ms</td>
<td>0 %</td>
</tr>
<tr>
<td>erosion 3*3</td>
<td>23 ms</td>
<td>23 ms</td>
<td>0 %</td>
</tr>
</tbody>
</table>

5. Related Work

Up to now, shared memory machines were mainly specific design machines, and most of the clusters of workstations or PCs have been used with message passing models, like SP1 and SP2 from IBM, or fast user-level messages. For instance, fast user-level messages were supported without shared memory on the CM5, though the protection mechanism was relatively static. Also, among workstation networks, user-level IPC can be found in the Princeton Shrimp [17], the HP Hamlyn interface [18] to Myrinet [19], and Dolphin’s snooping interface [20] for the SCI [26] cache coherent protocol. The Myrinet switching fabric, like ServerNet [21], has been used by several research projects to produce high performance message passing interface. These include the Myrinet API [22], Active Messages (AM) [23] from the University of California at Berkeley, or Fast Messages [24] from Illinois University.

Distributed shared memory machines can be grouped into systems that supports more-or-less shared-memory programs, such as might run on a machine with hardware coherence, and those that require a special programming notation style. Nitzberg and Lo [31] provide a survey of early VM-based systems. A first solution used by Memnet [32], Merlin [33] and SESAME [34], was to use the page-based automatic update approach to support shared memory. These systems do not provide a mechanism for high bandwidth, low overhead block data transfer such as deliberate update. Recent work by the Alewife group [35] at MIT has addressed the implementation of software coherence on a collection of hardware-coherent nodes [36] [37]. Another approach is to use distributed information in the form of timestamps and write notices to maintain sharing information. It is the case of TreadMarks [38] and AURC [39] which is a multi-writer protocol designed for the Srimp network interface [17]. Like Cashmere [40] based on the Memory Channel from DEC [41], it relies on remote memory access to write shared data updates to home nodes.

Concerning systems that provide a shared address space entirely in hardware, DASH [25] is a cache-coherent multiprocessor that uses prefetching. The KSR1 [5] provides a shared address space through cacheonly memory. The Scaleable Coherent Interface [26] also specifies mechanisms for implementing large, shared address space. The Cray T3D [27] integrates both message passing and hardware support for shared address space. Message passing in T3D is flexible and includes extensive support for DMA. Like GFLOPS, the T3D does not provide cache coherence. Several recently proposed architectures are based on the integration of shared memory and message passing in some form. FLASH [28] includes a microcoded coprocessor for message handling including shared-memory protocol messages. The *T [29] and Typhoon [30] architectures offer user-level message handling using a second processor dedicated to the network interface.

A variety of systems, similar to our solution, implement coherence entirely in software, without VM support, but require programmers to adopt a special programming model. In some systems, such as SplitC [9] and Srimp's Deliberate Update [17], the programmer must use special primitives to read and write remote data. In our case, these primitives are prefetch() and forward(). In others, including Shared Regions [42], Cid [43], and CRL [44], remote data is accessed with the same notation used for local data, but only in regions of code that have been bracketed by special operations. The Midway system [45] requires the programmer to associate shared data with synchronization objects, allowing ordinary synchronization acquires and releases to play the role of bracketing operations. Several other systems support interprocess communication via sharing of special concurrent objects [46].

6. Conclusion

The diversity of algorithms in the scientific and image processing fields needs the use of general purpose architectures with a high level programming environment. The parallel language is a key tool to help the programmer. C// is a high level data parallel language using a concise and powerful syntax extension from the C language. It provides several parallel constructs and routines which help the programmer to write applications. These constructs are coherent with the resources available in the target architecture. The aim was to develop a compiler that uses directly and efficiently the architecture
features. Thus, by the joint design of both the compiler and the architecture, the abstraction level provided by the C// language takes fully advantage of the architecture. We have also evaluated the impact of the parallel constructs on the execution time and obtained that it is negligible when the computation part of the algorithm is more important than the communications. In this paper, we have investigated the GFLOPS capability in supporting fine and medium grain parallelism for a shared memory environment. The C// parallel language, designed for global name space multiprocessors, has been ported to GFLOPS and currently to the SYMPHONIE machine. The porting of the language to such platforms has been relatively easy due to the easiness of the GNU CC portability. Our work focuses now on the implementation of real applications.

7. Acknowledgments

The following members of the GFLOPS team contributed significantly to the success of the project: Remy Loupias developed the first prototype board, Mernouch Oroudji wrote the simulator/debugger, Philippe Petillon designed the first VLSI, Julien Bordes designed the second VLSI, Olivier Caprani designed the second prototype, A. Fatni designed the C data-parallel language and its compiler, D. Houzet and J.L. Basille supervised the project progress. The GFLOPS project is funded by a CNRS PRC-ANM contract, a CEA-LETI Nuclear Agency collaboration, INP funds (BQR) and CNFM National Education funds.

8. References


