Efficient parallelization strategy for the SPH method

John BIDDISCOMBE (ETH Zürich)
David LE TOUZÉ (Ecole Centrale de Nantes)
Francis LEBOEUF (Ecole Centrale de Lyon)
Jean-Christophe MARONGIU (ANDRITZ Hydro)
Guillaume OGER (HydrOcean)
Jérôme SOUMAGNE (ETH Zürich)

Dissemination level: PUBLIC

– March 15, 2010 –
Content

1. Existing HPC hardware .................................................................5
   1.1. Cray XT5 (CSCS) .................................................................6
       1.1.1. High Speed Interconnect ...........................................7
   1.2. Blue Gene®/P ................................................................. 8
       1.2.1. System overview ....................................................... 8
       1.2.2. Interconnect ..............................................................8
       1.2.3. Execution process modes ...........................................9
   1.3. I/O nodes and File Systems on large machines ....................... 10
   1.4. Cluster based Supercomputers ........................................... 11
   1.5. MPI Limitations and best practices ....................................... 12
   1.6. Cell Broadband Engine ................................................... 13
   1.7. Compute Unified Device Architecture (CUDA) ....................... 14
       1.7.1. Architecture overview ..............................................15
       1.7.2. Next generation: Fermi’s new features .......................16
       1.7.3. Limitations and best practices ....................................16
   1.8. Arithmetic Intensity ........................................................... 17
       1.8.1. Application to SPH ......................................................19
   1.9. x86 Architectures and Cache Management ............................ 19
       1.9.1. Iterating over lists .....................................................21
       1.9.2. Associativity .................................................................22
   1.10. Monitoring Hardware Counters: PAPI & TAU ...................... 22

2. Specificities of the SPH codes for their parallelization ............... 25
   2.1. Specificities of the SPH formalism ....................................... 25
       2.1.1. Lagrangian nature of SPH ......................................... 26
       2.1.2. Explicit nature of SPH ...............................................26
       2.1.3. Acoustic problem and CFL condition .........................26
   2.2. Review of existing parallel SPH codes ..................................27
       2.2.1. Parallel SPH codes based on MPI library ....................27
       2.2.2. GPU SPH codes .........................................................28
   2.3. SPH-flow parallelization features ........................................28
       2.3.1. Overall parallel algorithm of SPH-Flow .......................28
       2.3.2. Non blocking MPI point-to-point communications ..........29
       2.3.3. Particle domain updating .........................................30
       2.3.4. Load-balance control .................................................31
       2.3.5. Scalability study and MPI profiling ............................31
   2.4. ASPHODEL parallelization features ......................................34
3. Efficient parallelization of SPH codes on modern architectures....37
   3.1. Efficient inter-node communications on CPU clusters.........................37
   3.1.1. Domain decomposition and load-balance control.............................37
   3.1.2. Importance/efficiency influence of masking MPI communication latencies ....38
   3.2. Multi-threading and efficient SPH parallelization..................................38
   3.2.1. Memory access tests and new developments.....................................39
   3.2.2. Recommendations regarding OpenMP algorithms..............................39
   3.2.3. Recommendations regarding GPU algorithms......................................40
   3.3. Efficient parallelization on hybrid CPU/GPU architectures .................42
   3.3.1. Hybrid OpenMP/MPI...........................................................................42
   3.3.2. Recommendations regarding SPH parallelization on hybrid architectures....43

4. References...........................................................................................................45
1. Existing HPC hardware

HPC hardware has gradually converged towards a relatively small number of solutions over the last few years as the prices of commodity products have continually fallen and high-end vendors have adopted the same core processors that are used in desktop machines and clusters. The categories that are examined in this document are as follows:

- Rack based HPC machines with custom interconnect and filesystem
  - AMD Shanghai/Istanbul: Cray XT series,
  - IBM Power Processors: Blue Gene (L/P)
  - Intel Nehalem/Itanium: SGI/Sun

- Clusters of rack based commodity machines linked by Infiniband or Gigabit Ethernet
  - ‘Generic’/Rocks clusters, AMD/Intel processors

- Custom Clusters containing GPGPU arrays (new)
  - nVidia Tesla/Upcoming Fermi architectures
  - IBM PowerXCell: (RoadRunner)

Broadly speaking, the categories which are most likely to be used by the partners in this project are X86 architecture style racks of compute nodes and GPGPU clusters which are in a new development phase. It is interesting to compare some properties of Blue Gene (and Cell based) machines as they offer an insight into the way software must be tailored to a specific platform (though this applies equally well to GPGPU machines). In this section, a brief description of some existing hardware platforms is presented, followed by a discussion of how these architectures affect the way programs run on them.

The most important trend that can be observed in modern supercomputing is that the number of cores is increasing rapidly, but the system memory is lagging behind in terms of its throughput. This means that as the raw FLOPS count increases, it is becoming increasingly difficult to feed the compute cores with the data required. Figure 2, shows a table of vital statistics of recent AMD x86 processors which illustrates the trend nicely. The problem is even more striking when Stream Processors such as the Cell Broadband Engine and GPUs are considered (see sections 1.6 below). For all architectures, maximum performance can only be achieved if the management of memory is very carefully considered. This may be by careful coding combined with knowledge of the hardware (making use of compiler and language libraries/optimizations/hints in traditional high level languages), or more specific...
tuned kernels which are custom written for a particular memory layout (as one may find with GPU/stream style programs).

1.1. Cray XT5 (CSCS)

The leadership machine installed at CSCS is a Cray XT5 supercomputer, it will be used within the NextMuSE project not only to benchmark/test certain SPH codes but it will also serve as a test platform for large-scale ICARUS + SPH steering and experimentation.

The Cray is based upon AMD Opteron Istanbul processor with dual hex core CPUs per node. The installation at CSCS is arranged as 4 nodes per blade, 8 blades per rack, 3 racks per cabinet, 10 cabinets per row and 2 rows. There are 1844 compute nodes (22128 cores), 20 I/O nodes (Lustre) and 4 login nodes. The compute nodes have 16GB memory or 1⅓ GB per core. The total memory available is 29TB. The configuration (other than number of cabinets) is typical for a Cray machine of this generation.

Each CPU is a Hex core AMD Opteron 2.4GHz Istanbul with a theoretical performance per core of 9GFlops, the peak theoretical performance of the machine is 212 TFlops.

Figure 1: The 20 cabinet liquid cooled Cray XT5 (Rosa) installed at CSCS

Figure 2: The trend in processor design has led to a 6x core count increase (2009), with a 3x memory transfer improvement, and for the next 12 core generation, a 5x increase relative to 2003. (Note that GT/s refers to Transfers, which includes error checking bits, the rate is 4/5 in GB/s).
1.1.1. High Speed Interconnect

The Cray (like other large scale machines such as Blue Gene) differs largely from a ‘Generic’ cluster built from commodity parts in that it has a custom built high speed interconnect. There are in general, two distinguishing features of a custom interconnect on a machine such as this.

- Torus Topology with high speed routing (via Cray SeaStar chip in this case)
- Low Latency – High Bandwidth transfer of data/packets

The SeaStar chip is connected one per compute node (12 cores for the XT5, 16/24 for upcoming XT6) and is capable of routing 57.6 GB/s per chip – The SeaStar is effectively a 6-port switch and traffic on a single port is 9.6 GB/s which implies that in the absence of other traffic, compute nodes can communicate with each other (one way) at a peak data rate of 9.6 GB/s. However, the connection between nodes is made using a 3D torus, where traffic is routed along +/- X/Y/Z connections (hence 6 port switch) between compute nodes. The connection of nodes depends upon Cabinet/Rack/Blade/Socket/etc (using a combination of attributes) and since the torus wraps around from N to Zero in all X/Y/Z directions, the direction chosen by the routing mechanism inside the SeaStars ensures that bi-directional traffic does not collide. Figure 3 shows a simple example in 2D.

![Traffic from A to B](image)

*Figure 3: Routing on the Cray inter-connect uses a different path for bi-directional traffic. This avoids collisions of data. In practice the XT5 uses a 3D connection so traffic into/out of the page is present too.*

The Torus arrangement enables a higher sustained bandwidth to be achieved between message intensive application nodes. Future Cray supercomputers will make use of a new inter-connect based on the successor to SeaStar (code named ‘Gemini’) and may route traffic on a higher dimensional torus (5 dimensions) – this will in turn reduce contention and increase throughput accordingly. The Torus arrangement with communications chips bound to compute nodes make the system highly scalable with the dimensions of the torus simply expanding as more nodes are added. This is much more efficient that would be the case with a central switch connected to a commodity cluster.

The Cray MPI library is built upon a portals library which is specific to the SeaStar based interconnect. The Seastar communicates using DMA directly to the host memory of the CPU which allows for very low latency and high throughput message passing.
1.2. Blue Gene®/P

In 2007 IBM introduced the Blue Gene/P system as its next generation of massively parallel supercomputers, based on the same architecture that is in the Blue Gene/L system. Below is a quick description of the system components.

1.2.1. System overview

The system contains the following components:

- **Chip:** The Blue Gene/P base component is a quad-core 32 bit chip. The frequency of a single core is 850MHz.

- **Compute card:** One chip is soldered to a small processor card, one per card, together with memory (DRAM), to create a compute card (one node). The amount of DRAM per card is 2 GB or 4 GB.

- **Node card:** The compute cards are plugged into a node card. There are two rows of sixteen compute cards on the card (planar). From zero to two I/O Nodes per Compute Node card can be added to the node card.

- **Rack:** A rack holds a total of 32 node cards.

- **System:** A full petaflop system consists of 72 racks.

Memory constraints on a Blue Gene machine arise because the memory is physically soldered to the compute card, so it cannot be replaced with larger DRAM modules and is fixed at the manufacturing size. 2GB for 4 cores leaves just 512MB per core (difficult for many applications to work with) and since nodes have no direct IO, this represents a hard limit as no swap space is possible. As the processor speed is much lower in Blue Gene machines than x86 based ones, allowing a single core to make use of all available memory is not a good strategy as many more overall nodes will be required to produce the same work than the equivalent x86 machine.

The Blue Gene is best suited for very scalable applications with a low memory per core requirement, and delivers the best overall Flops/Watt available in current machines.

1.2.2. Interconnect

The Blue Gene has 3 separate networks,

- **3D Torus network** for Program communication similar to that described for the Cray XT5, but making use of network controllers embedded into a combined node ASIC chip. The routing algorithm is similar to that described for the Cray in that bidirectional traffic to/from the same nodes will not collide.

- **Global communication network** for collective operations. This requires special software calls that use MPI extensions to make use of the network.
• Barrier network. A specialized interrupt driven network for barrier operations which improves scalability for very large node counts.

1.2.3. Execution process modes

The Compute Nodes on Blue Gene/P are implemented as four cores on a single chip with 2 GB or 4 GB of dedicated physical memory in which applications run. A process executes on in one of the following three modes: VN, DUAL or SMP mode.

Virtual Node (VN) mode

In the VN mode, each compute node can run four separate processes or MPI tasks. Shared memory is available between processes of one node.

Dual (DUAL) mode

In DUAL mode, each physical Compute Node executes two processes per node with a default maximum of two threads per process. Each task in Dual mode gets half the memory and cores so that it can run two threads per task.

Symmetric multiprocessing (SMP) mode

In Symmetric Multiprocessing (SMP) mode each physical Compute Node executes a single process per node with a default maximum of four threads. The Blue Gene/P system software treats the four cores in a Compute Node symmetrically. This mode provides the maximum memory per process.

![Compute node configuration options](image)

*Figure 4: Compute node configuration options*

It is important to consider the memory size per core when running simulations. Depending on the execution mode chosen (VN, DUAL or SMP) the amount of memory allocated will be different. Therefore, when using the VN node which implies that resources are not shared, the amount of memory per core/node will be 512MB or 1GB. Instead, when using the SMP mode (MPI+OpenMP), 2GB or 4GB will be available. The ideal simulation running on Blue Gene would thus use many cores (this implies to have a certain scalability of the application), use both OpenMP and MPI and consume less than 2 or 4GB per node.
1.3. I/O nodes and File Systems on large machines

When building an HPC machine made from blades of compute nodes/IO nodes/login nodes etc. one generally cannot assume that the compute nodes will have direct access to the filesystem. They must therefore send data via the designated IO nodes in order for it to pass through to disk (or elsewhere). In general, the IO and Login nodes will have visibility to an external network and the compute nodes will not (though in some cases, clusters are built with multiple network cards per node allowing both internal network routing and external network visibility).

There are two common parallel file systems in use within the HPC community. Lustre (open source) and GPFS (IBM). Terminologies differ between the two filesystems, but they both basically enable parallel disk IO by striping data across disks. Multiple disk arrays are served by multiple servers, which coordinate the writing of data according to file/data offsets per ‘stripe’. Stripes can be configured to be different sizes and allocated differently to different processes (on Lustre) which enables a system to serve a large number of requests from multiple applications. On GPFS, striping is handled automatically by the system. Scalability is provided by adding more disk arrays and more servers. In Lustre terminology, the disk (arrays) are referred to as Object Storage Targets (OST), and the nodes which control them as Object Storage Servers (OSS). File IO nodes are thus OSS nodes. Adding more OSTs and OSSs increases the total throughput, but metadata must be exchanged between OSS nodes to coordinate how a file is read/written on a per stripe/block basis. Note that GPFS differs from Lustre in that it has multiple metadata servers, whereas Lustre uses a single one, multiple servers improve the performance when many requests are taking place concurrently (generally when many files are being manipulated simultaneously on the system).

Since IO traffic must be routed via these specialized nodes, it pays to follow certain guidelines when writing (and to a lesser extent reading) from disk to/from a single shared file.

- Collective communication is nearly always better than independent communication. When sending data from a large number of compute nodes to a relatively small number of IO nodes, a large number of smaller messages will be sent, and more metadata exchanges take place. If certain nodes act as collectors of data from neighbouring compute nodes and these are forwarded on as a series of larger IO requests, the total latency is reduced. Both MPI-IO and HDF5 support this kind of activity.

- Vendors of large machines (e.g. Cray) have made optimizations to their MPI implementation so that they can take advantage of knowledge about the IO subsystem. On the XT5 system, Cray includes several “Collective Buffering Alignment Algorithms” which can improve performance by using knowledge about the Lustre file striping settings and OSS node arrangements. Setting an environment variable – for example MPICH_MPIIO_CB_ALIGN prior to application execution allows the MPI-IO layer to select different collective buffering algorithms. A simple collective buffering algorithm may accumulate data from N nodes based on the data offset – divided between IO nodes, but if the striping pattern divides the data into x MB blocks per OSS, then it is better to collect the data by taking into account this strided offset to avoid sending data twice before reaching the disk. A number of algorithms are implemented and each system MPI documentation should be consulted for hints and settings to ensure that optimizations specific to an installation are not being missed.
• Latest off-the-shelf MPICH implementations now contain specific optimization hints/flags for Lustre and GPFS, these should be consulted even on home built clusters if supported file systems are installed.

• Writing one file per node should be avoided. On large systems with parallel file systems, one file per node can lead to an explosion in the amount of metadata being handled by the file system. Whilst the raw data throughput can be acceptable at the time of writing actual data results, the strain on the servers for other users of the system, and at post processing time when dealing with thousands or millions of files, can lead to an inefficient workflow which could have been be avoided.

On some systems, IO nodes are connected only to disk arrays and their own dedicated network ports, however on others (clusters in particular), the IO nodes are simply a gateway to disks and also have visibility to other networks. Within the NextMuSE project, CSCS is implementing parallel IO for direct visualization which will take advantage of IO nodes (or other nodes designated for IO usage) to route data in parallel off the system for concurrent analysis.

1.4. Cluster based Supercomputers

The principal difference between custom ‘branded’ HPC machines and those that are built as clusters of commodity nodes is the network infrastructure. Figure 5 shows the general layout of such a system. The machine consists of racks of compute nodes, which communicate with each other via a central switch. Choices for the communication infrastructure include Infiniband, Myrinet, Quadrics, Gigabit Ethernet and a number of other vendor specific solutions. The most frequently chosen fabric in modern clusters is Infiniband which provides extremely good performance under a wide variety of conditions. Note however that routing of Infiniband traffic over Ethernet networks (which are generally used for the global networks in computing facilities) is not directly possible (though some solutions are being trialled).

![Figure 5: A cluster solution has an interconnect which is a central switch rather than a dedicated torus. Multiple switches may be present if the filesystem has a dedicated network and login/head nodes also have global network visibility](image-url)
Given the above example, several possibilities are available. A single network connects all nodes so that computational traffic competes with IO and external communication to other networks via a single switch (all nodes have a single network card). Alternatively, a single switch may be used for internal computation (all compute nodes have a single network card), and a second switch handles IO and global network traffic (login(IO) nodes have two cards).

In smaller clusters, it is common for all compute nodes to have two network cards, and in these cases, it pays to make sure that applications are routing traffic through the (usually faster) network for application communication, whilst IO and other network communication takes place over the second network. This may require the careful control of Environment variables on the system to ensure that (for example) Infiniband data is sent via one port and Ethernet data via another (In this case, MVAPICH would be configured to ‘know’ that one card is Infiniband, another not).

In the case where each IO node is connected to the filesystem (which will assume is a parallel one such as Lustre/GPFS – see elsewhere in this document for details) via a dedicated 20Gb/s link, total data rate will be multiplied by the number of IO node links, the total speed may be hundreds of Gb/s – but ultimately the real speed is determined by the disk array capabilities.

Additionally – though IO speeds from the compute nodes to and from disk may be very high, if the data is to be post processed in any way on another machine, one must be aware that this will be routed via head/login nodes off one filesystem to the ‘global network’ – which may be only accessible via a slower (10Gb/s for example) Ethernet link. This external link may be a bottle neck for large data transfers, particularly if the head/login nodes also use the same externally visible network for their own internal filesystem communication (as is the case when imported home directories are mounted over NFS and made visible to the whole cluster).

### 1.5. MPI Limitations and best practices

The MPI implementation on the both Blue Gene/P and Cray systems is derived from the MPICH2 implementation of the Mathematics and Computer Science Division (MCS) at Argonne National Laboratory. Custom machine vendors usually take the standard open source MPICH2 implementation and add low level drivers to support their own interconnect and file system specific optimizations.

Moreover the Blue Gene architecture supports all the MPI functions and provides the user with MPI extensions to optimize process communication, run time execution. The current MPI implementation on the Blue Gene/P system supports the MPI-2.1 standard. The only exception is the process creation and management functions (MPI_Comm_spawn, etc).

An efficient MPI application on these machines observes the following guidelines:

- Overlap communication and computation using MPI_Irecv and MPI_Isend, which allow direct memory access (DMA) to work in the background.
- Avoid load imbalance. This is important for all parallel systems. However, when scaling to the very high numbers of tasks that are possible on large systems, it is important to pay close attention to load balancing. Optimizing load imbalance will pay off more significantly than optimizing single node performance on large machines.
Avoid buffered and synchronous sends; post receives in advance. Posting receives in advances allows the MPI library to use the user supplied data buffer directly to copy received data into, instead of an internal buffer. MPI internally uses an expected and unexpected message buffer, large unexpected messages received trigger relatively expensive memory allocations and copies.

The MPI standard defines several specialized communication modes in addition to the standard send function, MPI_Send(). Avoid the buffered send function, MPI_Bsend(), because it causes the MPI library to perform additional memory copies. Using the synchronous send function, MPI_Ssend(), is discouraged because it is a non-local operation that incurs an increased latency compared to the standard send without saving memory allocation.

Avoid vector data and non-contiguous data types. While the MPI-derived data types can elegantly describe the layout of complex data structures, using these data types is generally detrimental to performance. Many MPI implementations, (including the Blue Gene/P MPI implementation), pack (that is, memory-copy) such data objects before sending them. This packing of data objects is contrary to the original purpose of MPI-derived data types, namely to avoid such memory copies. Memory copies are particularly expensive on Blue Gene/P due to the relatively slow processor clock compared to the fast network hardware capabilities. Avoiding non-contiguous MPI data types and memory copies in general, improves application performance.

1.6. Cell Broadband Engine

It is unlikely that any of the partners in the NextMuSE project will be targeting the Cell processor for their simulations, however, in 2008(2009) the part Cell based Roadrunner machine at the Los Alamos National Laboratory in the USA was the fastest machine in the world and the first to achieve a Petaflop computation. Based on a combination of AMD Opteron and Cell processors it represents a novel design and shows one of the possible trends in the evolution of future supercomputers.

Conventional architecture CPUs excel at general purpose programming and application control, whilst highly specialized streaming processors provide a workhorse for high sustained computation. The current evolution of CPU/GPU computation mirrors this by placing a large number of streaming cores under the control of a central CPU - whilst the Cell processor itself may be discontinued, the essential philosophy of its conception lives on in hybrid CPU/GPU form.

The Cell setup includes one Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The SPE consists of a Synergistic Processing Unit which is the actual SIMD-capable streaming processor and a Memory Flow Controller. The SPU computes on 128bit wide data, 16*8bit, 8*16bit, 4*32bit integers or 4*32bit floats in each instruction – running at a clock speed of 3.2GHz, this equates to 25.6GFlops. Maximizing the computation to sustain this potential is a non trivial task, and in section 0, an example of is presented showing some results of careful optimization which is able to approach this rate.

The SPU is capable of executing 2 instructions per clock using even and odd pipelines, with the even one containing the conventional ALU logic responsible for integer and floating point operations, whilst the odd pipeline handles memory load, prefetch, and other flow logic operations. To maximize the actual throughput, the compiler exposes many of the low level
hardware controls to allow the programmer to achieve the ideal scenario of concurrent int/float operations in one pipeline whilst the memory load/store operations take place in the other pipeline.

The SPUs are connected via an Element interconnect bus, which is theoretically capable of 205GB/s (~8*25.6) transfers to and from on chip memory though in practice the achieved data rates will not approach this. (Application tests show 40%-75% performance is feasible under carefully controlled conditions). On chip memory is 256kB, and there are 128 128bit registers on which operations take place. Memory is coherent, with no specific cache - giving full control of all memory accesses to the programmer. This contracts highly with cache based programs written for conventional processors.

The memory controller on board, is capable of performing independent DMA transfers to and from the external RAMBUS XDR system memory using dual channel Octal data rate 400MHz (3.2GHz effective) transfers, giving a total bandwidth of 25.6 GB/s. In practice, mixed streaming read/write operations result in an effective transfer rate of around 20GB/s due to internal state changes as the memory controller alternates between read and write modes.

1.7. Compute Unified Device Architecture (CUDA)

NVIDIA’s GeForce 8800 was the product that gave birth to the new GPU Computing model. Introduced in November 2006, the G80 based GeForce 8800 brought several key innovations to GPU Computing.

In June 2008, NVIDIA introduced a major revision to the G80 architecture. The second generation unified architecture - GT200 (first introduced in the GeForce GTX 280, Quadro FX 5800, and Tesla T10 GPUs) - increased the number of streaming processor cores from 128 to 240. Each processor register file was doubled in size, allowing a greater number of threads to execute on-chip at any given time. Hardware memory access coalescing was added to improve memory access efficiency. Double precision floating point support was also added to address the needs of scientific and high-performance computing (HPC) applications.
1.7.1. Architecture overview

CUDA is the hardware and software architecture that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages.

A CUDA program calls parallel kernels. A kernel executes in parallel across a set of parallel threads. The programmer or compiler organizes these threads in thread blocks and grids of thread blocks.

The GPU instantiates a kernel program on a grid of parallel thread blocks. Each thread within a thread block executes an instance of the kernel, and has a thread ID within its thread block, program counter, registers, per-thread private memory, inputs, and output results.

A thread block is a set of concurrently executing threads that can cooperate among themselves through barrier synchronization and shared memory. A thread block has a block ID within its grid.

A grid is an array of thread blocks that execute the same kernel, read inputs from global memory, write results to global memory, and synchronize between dependent kernel calls.

In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and C automatic array variables. Each thread
block has a per-Block shared memory space used for inter-thread communication. Grids of thread blocks share results in Global Memory space after kernel-wide global synchronization.

1.7.2. Next generation: Fermi’s new features

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 streaming multiprocessors (SM) of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The main improvements this new architecture aims to reach are the following:

- Improve Double Precision Performance (Full IEEE 754-2008 32-bit and 64-bit precision) - 8x the peak double precision floating point performance over GT200
- ECC (Error Correcting Code) memory support
- True Cache Hierarchy with Configurable L1 and Unified L2 Caches
- More Shared Memory
- Faster Context Switching
- Faster Atomic Operations

1.7.3. Limitations and best practices

Since this architecture is really different compared to an x86 processor, there are some rules which need to be followed in order to reach a sustainable performance:

- Optimize algorithms for the GPU:
  - First of all it is important to maximize independent parallelism, i.e. use as many threads as possible and avoid communication between threads as much as possible.
  - Sometimes it is better to re-compute than to cache because GPU spends its transistors on Arithmetic and Logic Units (ALUs) and not on memory.
  - One of the most important points is to do as much computation as possible on the GPU to avoid costly data transfers. Even low parallelism computations can sometimes be faster than transferring back and forth to host.
  - Optimize memory access and take advantage of shared memory which is hundreds of times faster than global memory. Threads can cooperate via shared memory. On the new Fermi architecture, it is also possible to configure the size of L1 and L2 caches. There is an L1 cache per streaming processor and an L2 cache for global operations. The per-SM L1 cache is configurable to support both shared memory and caching of local and global memory operations. The 64 KB memory can be configured as either 48 KB of Shared memory with 16 KB of L1 cache, or 16 KB of Shared memory with 48 KB of L1
cache. Depending on the simulation/program running on the card, taking advantage of either the shared memory or the cache can improve significantly the performance, reducing accesses to the DRAM. This is also valid for common processors. Managing algorithms/programs so that data can fit to the cache (as much as possible) can considerably reduce the number of data cache misses and therefore reduce the time necessary to compute the next iteration of a simulation.

- Use load balancing: partition the computation to keep the GPU streaming multiprocessors equally busy and keep resource usage low enough to support multiple active thread blocks per multiprocessor. During the execution, thread instructions are executed sequentially. Thread blocks are divided into warps of 32 threads. The warp is the fundamental unit of dispatch. Therefore executing other warps when one warp is paused is the only way to hide latencies and keep the hardware busy.

CUDA programs running on GPU can achieve great performance if one can maximize parallel execution, maximize memory bandwidth and maximize instruction throughput.

### 1.8. Arithmetic Intensity

Arithmetic efficiency is a useful construct to measure the potential performance of a tightly executed kernel – which depends upon the hardware in runs. It has been used to evaluate the behaviour of stencil operators for conventional grids, but is equally applicable to a Lagrangian formulation. The arithmetic efficiency is defined as the ratio of computation to bandwidth.

\[
\text{arithmetic intensity} = \frac{\text{operations}}{\text{data transfers}}
\]

**High arithmetic intensity: processor bound**

**Low arithmetic intensity: memory bound**

Consider for example (from [11]) the case of a 3D (7 point stencil) Finite Volume solver which sums contributions from neighbour cells to produce a new value at \( T^{(n+1)} \) from the cell value, plus those from 6 neighbour cells at \( T^{(n)} \), using a total of 9 coefficients (\( C_{ijk} \)).

\[
T^{(n+1)}_{ijk} = (0) C_{ijk} \left( T^{(n)}_{ijk} \right)^2 + (1) C_{ijk} T^{(n)}_{ijk} + (2) C_{ijk} + \\
(3) C_{ijk} T^{(n)}_{i+1,j,k} + (4) C_{ijk} T^{(n)}_{i-1,j,k} + \\
(5) C_{ijk} T^{(n)}_{i,j+1,k} + (6) C_{ijk} T^{(n)}_{i,j-1,k} + \\
(7) C_{ijk} T^{(n)}_{i,j,k+1} + (8) C_{ijk} T^{(n)}_{i,j,k-1}
\]

The data lies in a regular slab and the computation requires a memory load to bring in the solution values for \( T^{(n)} \), another to write the result, and 9 for the coefficients (which in this case are spatially variable and not available from the cache). The kernel implementation would be as follows (A, B being processor local memory buffers for result data).
NextMuSE

Next generation Multi-mechanics Simulation Environment
a Future and Emerging Technologies FP7-ICT European project

```c
void stencil3D (float A[], float B[], float c[9][9]) {
    for (t=0; t < t_final; t++) {
        for all grid indices in x-dim {
            for all grid indices in y-dim {
                for all grid indices in z-dim {
                    B[i] = A[i]*(c[0][i]*A[i] + c[1][i]) + c[2][i] +
                }
            }
        }
    }
}
```

//double buffering swap A and B

tmp = B; B = A; A = tmp;

Without specific memory preloading of data values, the simplest traversal of the grid to perform the calculation requires 7+9 memory loads and one result write. The implementation requires 16 Flops and therefore has an arithmetic intensity of 16/(17*4).

The example given in [11] runs on a Cell BE processor and which has a theoretical maximum flop rate per SPU of 25.6 GFlops/s, and a memory bandwidth of 25.6GB/s – therefore for a single precision execution of the stencil, the maximum practical performance given would be

\[
\frac{25.6 \times 16}{(2+7+1) \times 4} = 6.02 \text{ GFlops}
\]

The implementation is bandwidth limited, and simply cannot keep the processor fully occupied due to the reliance on memory fetches. By preloading 3 planes of the data into processor local memory (be it L1 cache memory of specific local memory on a Cell or GPU architecture) making use of double buffering and treating the solution as a series of (overlapping) sub-blocks, the memory accesses may be reduced to 9+1+1, which improves performance to

\[
\frac{25.6 \times 16}{(1+9+1) \times 4} = 9.31 \text{ GFlops}
\]

It is clear from this example that sub blocks must overlap by one boundary cell since the kernel required the neighbours from one cell along each edge. (Note that in this example, the coefficients are in main memory and it is not possible to load planes of these as well as solution data due to memory restrictions. If this is not the case, then naturally the numbers improve accordingly).

The problem of improving the performance beyond this point has been treated in both [12] and [11] by adding the temporal domain into the mixture. It can be observed that for the stencil example given, the values computed will propagate one cell in each direction on each timestep (diffusion), by unrolling the calculation of \( T(n+1), T(n+2), T(n+3) \) ...and having blocks which overlap by additional cells (up to the maximum available local memory), one is able to re-use the same data items in memory without additional load/store operations. In effect, multiple time steps are computed as sub iterations of the main solution loop. Additional ‘wasted’ computations must be made to cater for the block overlap regions where intermediate calculations must be done twice in subsequent blocks, but since computations on locally available data occur at the processor clock speed – which is an order of magnitude faster than an external memory fetch, the performance of the overall code was improved to
20 GFlops. This illustrates a general trend in modern programming, that redundant computation which is cheaply available due to the large number of cores or streaming processors available may be beneficial if it avoids memory accesses. Careful optimization of code to match the architecture is of paramount importance.

1.8.1. Application to SPH

For the example stencil given above, it is clear that careful arrangement of memory, algorithm and time-dependent solution unrolling can lead to dramatic improvements in performance. It is not so clear that SPH algorithms can be trivially adjusted in the same way. In general SPH is arithmetically more intensive than a grid based code, it makes use of distance computations between particles, and a kernel evaluation which is computationally expensive for every interaction, therefore, the formulation is by definition more compute bound than for a grid calculation. It is also the case that the particle based nature of SPH and the reliance on neighbour searches for computations makes the kind of optimizations illustrated above more difficult. Very careful consideration of the layout of particle data in memory is necessary, and how neighbour lists are traversed in order to reduce the load/store of particle data from non-local memory. Since particles move over time, the amount of temporal unrolling that is possible with an SPH solver is necessarily going to be less. Other sections of this document describe some of the ongoing attempts to improve the performance on CPUs and GPUs.

1.9. x86 Architectures and Cache Management

We see that with all architectures, the management of memory is of paramount importance if the processors are to be kept busy. With GPU and Cell type stream processors, the memory management has, to some extent, been placed under programmer control, whereas the x86 architecture (for historical reasons as much as anything) allows the processor logic to make decisions about memory transfers. This does not prevent the user from making the most of the hardware The architecture is referred to as ccNUMA (Cache coherent Non-Uniform Memory Access) – implying that memory fetches are not constant time, but depend upon where the particular address is retrieved from, and that cache coherency must be maintained across processors/cores (dirty cache lines must be written to shared memory before other cores read them). The x86 architecture consists of processors from AMD and Intel. For the purposes of this discussion, the AMD processors are used for illustration. Intel CPUs have a different cache architecture and memory transport etc. but the essential arguments about cache optimizations are valid across processor families. The top end AMD processors are currently the hex core (8/12 core becoming available) Istanbul range with the following properties

A dedicated ‘per core’ 64 KB L1 (data) cache (+64kB instruction cache) with

- 2 way associativity
- 2 64-bit loads per cycle (3 cycle latency)

A dedicated ‘per core’ 512 KB L2 cache with

- 16 way associativity (9 cycle latency)
- Holds data evicted from L1 cache
- A Shared 6MB L3 Cache
  - 3248 way associativity (45 cycle latency)
  - Holds data evicted from L2 caches

Note 1: Main memory accesses may require >200 cycles,
Note 2: A register access is 1 cycle (free).
Note 3: Cache lines are 64 bytes wide (8 doubles, or 16 ints).

What are the implications for a programmer trying to optimize code when a multi-level cache is in use?

Random access to data becomes slower as the data block size grows, see Figure 8 which shows timing for random access to L1/L2 caches. The L1 Cache is $2^{12}$ bytes, whilst the L3 is $2^{20}$ bytes.

![Figure 8: An example of memory access times for a L1/L2 cache system (from [15])]()
which are larger than the cache size will all be making requests from main memory for data. Where one core may run at a given speed, N cores may cause a significant drop in performance caused by contention for memory bandwidth triggered by N cores which individually would be satisfied, but together swamp the bus.

### 1.9.1. Iterating over lists

When iterating over a list of items, the striding of data can play an important role in the overall speed of access. For consecutive elements of an array, (depending upon the item size), each memory access will trigger a cache miss and fetch each time a cache line is exhausted. For an element size of 4 bytes, a 64 byte line allows 16 elements to be used before the next line is requested. As the list is traversed, lines are pulled in and evicted as necessary, prefetching of lines reduces the total access time once a pattern is detected. As the working set grows in size, beyond L1/L2/L3 capacity the total access shows a trend similar to Figure 8.

If the data is not consecutive and subsequent elements are separated by steps smaller than a cache line, then to some extent, prefetching of data will ameliorate the access time, N elements will be used prior to each cache line fetch, and the next line will be prefetched whilst this is going one. However, once the stride between elements exceeds the cache line, then every access triggers a cache fetch (since in effect a prefetch is required every iteration and is therefore not really a prefetch any more). With a working set larger than the L3 cache size, the cost of memory accesses equates to that of the main memory access.

SPH codes must pay careful attention to how particle data is stored. In particular, the cell based access of neighbour lists is much more likely to produce data access patterns which reuse cache lines than an explicit neighbour list which is stored along with the particle – providing the particles themselves are stored in a spatially corresponding way. If particles are stored in memory without regard to their spatial arrangement in terms of neighbourhood, then random access patterns of particle interactions will result in negligible cache reuse when the number of particles * particle data size in the system exceeds the cache size.

From this discussion, it is clear that for a given list of particles, and particle interactions, the calculations which are made on them, should use as many available data items as possible before the particles are dropped and the next ones fetched. The inner iteration of the SPH kernel, must try to compute all interactions possible for a given list of particles which are currently in memory, rather than performing searches of neighbours repeatedly. When possible particle lists should be traversed in regular order to avoid random access patterns of cache requests, in this way prefetching of items can improve performance – also, prefetching of items can waste memory access if the order is randomized. The hardware makes redundant requests for items which actually reduces performance.

Storage of particle data as arrays of `x[num_particles]`, `y[num_particles]`, rather than as `struct particle {x,y,z etc}` should be used only if the number of particles in the process memory is manageable – or if the particles are sorted spatially (ie lists are traversed in some sequential order). Otherwise the random interactions of particles as they move around will produce many cache misses as most cache lines will hold only one element of usable data, and the remainder will correspond to particles which are out of the current calculation. Conversely, structures of particles are more likely to be reused again and again if interactions are computed based on spatially coherent traversal of the particle lists (and by implication neighbour lists). If particle structures are used, then all items in the structure (pressure, velocity, vorticity etc) should be used at the same time if possible, rather than loading and unloading the particles multiple times to compute different parameters. The
striding of particle data should also be noted, if a particle structure can fit into a single cache line (and be stored at integral cache boundaries), then it may be much more efficient than one occupying a cache line plus delta.

The reader is referred to [15] for an excellent discussion of all cache and memory related issues on conventional processors.

### 1.9.2. Associativity

The associativity of the cache is the number of cache lines to which a memory address may map. In the case of one-way or direct associative cache, a given memory location may only appear at one place in the cache, this makes the lookup of a given address very fast, but since the cache has a limited size, multiple memory locations will map to the same address. This results in many cache misses taking places as certain memory access patterns will repeatedly evict certain lines as they compete for the same slot. As associativity increases, the number of places that a given address can be placed increases, but the cost of retrieving it does too. This results in a trade off from L1-L2-L3 in increasing associativity, but decreasing cache misses. Increasing the number of cores sharing a cache effectively reduces the associativity by the same amount, so L3 caches need correspondingly higher associativity on multi-core designs.

The effect of associativity is subtle, but worth mentioning. With a 2-way associative cache a given memory address can only enter the cache in 2 places, the same is true for other memory addresses, and it is a simple function of the memory address to compute the cache line entry to which it maps. Since memory is restricted in where it reside in cache, one can imagine strided data access patterns which repeatedly hit the same cache line by walking through arrays with a (single byte) stride matched to the cache size/associativity – or in other words, if many data items fall into the same “set” of the cache. For SPH data where the access pattern is likely to be randomized, this may not present a serious problem, however, padding of structures such that they do not lie on exact multiple of the stride is used in matrix/vector type codes to ensure that conflict misses are avoided.

### 1.10. Monitoring Hardware Counters: PAPI & TAU

If optimization of both cache accesses and other architectural features are to be performed, then it is necessary to gain access to the hardware performance counters provided on chip. In general, a small number of registers are available for counting events, such as cache hits/misses, cycles stalled, instruction branches, floating point operations etc, and these may be setup by the programmer to count the events that are of interest (meaning not all events can be monitored all the time). The Performance API (PAPI) project is an open source toolset which the programmer may use to instrument his code in order to monitor these events. The library is written in such a way that it “will count similar and possibly comparable events when run on different platforms” [13]. Figure 9 shows an example of the kind of diagnostics that are possible using the tool.

Vendors such as Cray, also make available hardware monitoring software libraries, such as CrayPat (performance analysis tool), which are capable of collecting hardware information and integrating with debugging tools to make the programmers job easier.
For a higher level of diagnostics, TAU (Tuning and Analysis Utilities) [14] provides a framework or integrated analysis environment for parallel high level applications. The library supports even high level C++ template functions and allows precise timing information to be generated for a wide variety of problems.

Figure 9: Example of performance counters in action from the PAPI web pages.
2. Specificities of the SPH codes for their parallelization

This section first presents the specificities of the parallelization of an SPH code in terms of overall structure of the code to consider, in particular highlighting the advantages that this method may offer for multi-threaded applications. A bibliographic review is then proposed to present the parallelization strategies adopted by different actors of the SPH community. Finally, the parallelization features of two existing codes within the NextMuSE project (SPH-flow and Asphodel) are presented and discussed.

2.1. Specificities of the SPH formalism

The SPH method presents various peculiarities concerning its formalism, which naturally affects the parallelization strategy to adopt in order to obtain good scalability properties.

We call here “parallelization” the fact of distributing the overall calculation to perform onto several tasks. Two great choices are possible: the domain decomposition and / or the data decomposition. In both cases, the various tasks that make up the calculation must share fluid problem data. Such data is carried by particles.

The domain decomposition consists in geometrically splitting the overall fluid domain into as many tasks as desired, and attributing each sub-domain created to a task. The main job of the parallel software is then to meet these areas to get the desired overall result.

The data decomposition consists in distributing the fluid problem data without specially worrying about their spatial arrangement. Typically, this type of decomposition focuses on the loops, and the realization of simultaneous procedures (synchronization).

At this stage, both data and domain decomposition strategies may be envisaged to achieve a parallelization of a SPH code.
2.1.1. Lagrangian nature of SPH

As opposed to mesh-based methods for which a fixed grid is employed, the SPH method relies on moving particles that are advected by the calculated fluid flow. Depending on the problem to solve, large particle displacements may occur. The parallel scheme should necessarily take this specificity into account.

This Lagrangian nature of the method also implies a major disorder of the particles, and therefore of scattered data in memory. Note that for mesh-based methods, the connectivities are fixed during the calculation: the neighbourhood of a given cell of interest remains the same throughout the calculation. Thus, a recurrent pattern of interpolation can be used, simplifying the parallelization scheme. Unfortunately, such a simplification is not possible in the SPH formalism. Indeed, the neighbourhoods of SPH particles are constantly changing during the calculation, requiring a special procedure for updating neighbourhoods. The parallelization is thereby affected. More precisely, whatever the strategy adopted for the parallelization (data decomposition as well as domain decomposition), the core of the parallel algorithmic structure naturally revolves around a neighbour search procedure. Note that in the particular case of a domain decomposition parallelization, this Lagrangian aspect involves an additional constraint, since particles tend naturally to transit from one area to another (i.e. from one process to another). Specific procedures aiming at frequently updating particle data in relation to geometric areas should be developed.

2.1.2. Explicit nature of SPH

For the general case of implicit schemes, a linear system involving the whole data forming the fluid domain must be solved at each time step. Thus, very strong data inter-dependence exists for implicit schemes, and these data are spread over the various tasks involved in the calculation. Consequently, the speedup of linear system parallel solvers has some limitations, resulting in difficulties to get a good parallelization efficiency of the overall implicit code.

We stand here in the context of compressible SPH method, for which the formalism used is fully explicit. This particularity is a real advantage of SPH for its parallelization. Indeed, at a given instant of calculation, all the fluid variables of the overall problem are known and are independently marched in time, so that the distribution of data does not affect the calculation speedup (provided that parallelization is achieved with care).

In practice, it is always possible to sequence appropriately different parts of the calculation, so that no latency or redundant tasks appear. The explicit nature of the SPH method therefore largely facilitates its parallelization, so that it is theoretically possible to obtain a linear speedup.

2.1.3. Acoustic problem and CFL condition

The explicit nature of the compressible SPH method implies the imposition of a relatively restrictive CFL condition. The time step size imposed is particularly low, usually resulting in high computational costs. Nevertheless, this feature is particularly conducive to tend to the ideal speedup, especially in the context of quasi-incompressible flows. Indeed, for this particular case, particle neighbourhoods present very slow variations during several time
Efficient parallelization strategy for the SPH method

steps. It is thus possible to take advantage of this feature by performing a single
neighbourhood search that remains valid for several time steps. As mentioned above, the
core of the parallel algorithmic structure revolves around this specific procedure. Thus the
parallel structure of the code can take advantage of this aspect of non-systematic program
execution, allowing a drastic reduction of CPU times on procedures related to the intrinsic
parallelism.

2.2. Review of existing parallel SPH codes

Modern hardware architectures allow different types of SPH codes parallelisation. Today,
among the various HPC accelerations typically encountered in the SPH literature, one can
cite distributed memory oriented algorithms (based on the standard MPI), but also new
attempts of GPU implementations. Note that to our knowledge, shared memory parallel
programming (OpenMP directives for instance) are still relatively rare in the SPH related
literature.

2.2.1. Parallel SPH codes based on MPI library

The parallelization based on the use of MPI library appears as one of the practices most
often encountered in the literature related to particle HPC computing. We describe here only
a few SPH models to discuss briefly the main technical choices in the existing literature on
this specific topic.

Among the various existing MPI-based SPH models, Gadget-2 [6] appears as one of the
most advanced model. This open source code was written by Volker Springel, and its first
version appeared in 2000. Originally designed for astrophysical purposes, some adaptations
to fluid mechanics or other transverse specialties have been tried (see for example Ulrich et
al. [8]). In this software, the domain decomposition is achieved through the use of Peano-
Hilbert space filling curves proposed by Warren & Salmon [10], together with a neighbor
search procedure based on an oct-tree algorithm. This peculiarity ensures a correct load-
balance together with a sorting of particles that enables induced cache optimization
properties. One of the peculiarities of Gadget-2 is its ability to deal with sub-domains of any
prismatic shape. Particle information from other sub-domains is also detected using the oct-
tree walk, before its MPI transmission. The use of an arbitrary number of processors is
possible. However, due to the domain decomposition strategy adopted, this number is
restricted to powers of two. Among all its interesting properties, Gadget-2 supports parallel
I/O and various output formats, including the HDF5 format.

Other techniques can be found in the literature, as for the example of Fleissner et al. [2]. In
their model, the domain decomposition is also dynamically adapted. However, it is based on
the use of an “Orthogonal Recursive Bisection” [10] which consists in successive bisections
of the initial domain, to finally obtain the desired number of sub-domains to be addressed to
each MPI process. Here again, the link between neighbor search procedure and outer
particle point-to-point communications is clearly outlined. Nevertheless, they adopted a
manager-worker model where the manager coordinates the work distribution and ensures
the current load-balance. One of the peculiarities of their code also relies in the use of a
proportional-integral controller using the measure of current CPU times as load-balance
needs detection. The final load-balance correction is achieved by moving the sub-domain
boundaries, so that each local workload is modified. This technique allows additional
flexibility compared to parallel SPH model usually encountered, by allowing the parallelization to fully adapt to the machine, especially when it presents large hardware heterogeneities (presence of different kind of CPUs in the calculation network for instance).

### 2.2.2. GPU SPH codes

Some preliminary tests of GPU porting of SPH codes [4] have been conducted in the last two years, mainly within the SPHERIC community, showing very promising results in terms of expected speedup. These first attempts clearly highlight the need to modify the algorithmic structure of the code, in order to fully adapt it to this new architecture.

### 2.3. SPH-flow parallelization features

SPH-Flow (developed by ECN and HO) was designed with the general purpose of targeting hundred to thousand cores involved in the calculation, so that distributed memory architecture has essentially to be considered. Thus, the parallelization strategy of SPH-flow exposed hereinafter is expressed in the context of a distributed memory architecture, for which memory data must be sent from one task to another through the calculation network, namely by the MPI (Message Passing Interface) library [5][6].

The parallelization of SPH-Flow is based on a domain decomposition strategy. Thus, the global fluid domain to study is first divided geometrically into a number of sub-domains equal to the number of processes involved. Each of the sub-domains created is then assigned to a process. In order to obtain a code as simple as possible, a rectangular shape in 2D and a parallelepiped shape in 3D is adopted for the sub-domains.

The following discussion is based on the point of view of a single process (that we call here “process of interest”) in connection with its neighbour processes. The discussion is conducted on the 2D case for easier understanding, but the principle remains the same for the 3D case.

#### 2.3.1. Overall parallel algorithm of SPH-Flow

Assuming the creation of sub-domains and their linking to each process are carried out, consider a process of interest. It contains a number of particles, for which it is capable of performing the particle-to-particle calculations (discrete convolutions) everywhere except in areas close to the limits of its assigned area. Its neighbour processes hold the particles needed to complete its calculation (which we call here “outer particles”), and must therefore transmit the data specific to these particles (velocity, pressure, position... etc) to complete the discrete convolutions necessary to the time integration of data of its own particles (here called "inner particles"). The process of interest should first communicate areas containing the needed particles to its neighbour processes (dotted areas in the figure below) to allow its neighbours to then send the outer particles included in these areas.
Efficient parallelization strategy for the SPH method

Figure 10: Process of interest surrounded by its neighboring processes.

The boundaries of these areas (dashed lines) are determined from the local particles "seen" by the particles belonging to the process of interest (figure below).

Figure 11: Determination of outer particle areas (left), example of variable-H particle initialization (right).

Note that SPH-Flow is designed to deal with cases involving variable space discretization (variable-H). Thus, identification of particles to be transmitted should take into account the variation of radius, and the procedure described above must take into account this subtlety.

2.3.2. Non blocking MPI point-to-point communications

Various types of communications are possible using the MPI library: "blocking" and "non-blocking" communications. In the case of blocking communications, the sender process waits for completeness of data transmission before continuing its operations, and similarly, the receiving process waits for the reception completeness before continuing. These latencies are likely to affect the overall parallelization efficiency of the code. On the contrary, non-blocking communications do not require any sending and reception waiting. Sender and receiver processes can continue to run their own operations during a pending
communication. The code presented here takes advantage of this benefit, since communications can then be carried out during necessary (and as costly as possible) operations, that is to say during operations that do not require knowledge of outer particles. Covering the communication latencies using necessary operation allows a large performance increase.

### 2.3.3. Particle domain updating

Because of their motion (Lagrangian nature of SPH) some particles exit from the process of interest, while others enter its assigned area. In this peculiar condition, the procedure discussed above provides an additional advantage in terms of flexibility, since nothing prevents the process of interest to properly manage the inner particles placed outside its boundaries, and to access data related to outer particles having penetrated its limits, this case is illustrated in the figure below (exaggerated circumstances).

![Figure 12: Particle sub-domain changes (blacks dots : inner particles, red circles: outer particles).](image)

However, this situation tends to undermine the parallelization efficiency of the code in terms of computation time and communication latencies since a large number of outer particles is transmitted, while few of them actually interact with the inner particles. In the case of large deformations of the fluid domain, the process of interest could thus have to manage particles all placed outside its limits, resulting in very poor parallelization performances.

For these reasons, a specific procedure has been developed. It is designed to periodically update the inner particles to each process of interest, simply by transforming outer particles included within the limits of a process into its inner particles, and conversely by giving its inner particles placed outside its boundaries to its appropriate neighbour process. A call to this procedure is performed whenever the number of particles outside the limits of the process of interest exceeds a certain fixed percentage of the number of total inner particles.
2.3.4. Load-balance control

To obtain good parallelization properties, homogeneous workload should be imposed to each processor (as much as possible). This workload is a priori directly related to the amount of particle-to-particle interactions to achieve, themselves being dependent on the number of particles to be treated. Thus, all processes must have the same (or at least a rather identical) number of inner particles. This problem does not arise for a calculation in which the fluid occupies the entire computational domain. However, if the space occupied by the fluid varies greatly -as frequently encountered in problems involving a free surface- the number of inner particles can vary widely (following successive calls to the particle domain updating procedures above mentioned), resulting in strong unequal workloads. In this case, the processes the least loaded "wait" for the more loaded, lowering the overall code performances.

To ensure correct parallelization efficiency, some tests are regularly made during the calculation in order to estimate the gap between the amounts of inner particles of each process. When the gap becomes too large, a specific procedure is called, aiming at re-splitting the global fluid domain into sub-domains having identical amounts of inner particles, thus restoring a correct load-balancing of the parallel calculation.

2.3.5. Scalability study and MPI profiling

During the development of the parallel structure of SPH-Flow, and in order to improve its performance in a massive HPC context, various scalability studies were conducted in 2007 and 2008, in collaboration with the Ecole Polytechnique Federale of Lausanne (EPFL), on an IBM Blue Gene /L architecture [5][6].

Unfortunately, this study was limited by the limited available memory intrinsic to the architecture of the Blue Gene /L. This study allowed testing the code on a range of 16 to 256 cores, highlighting important limitations in terms of speedup and memory requirements of the code.

Following this study, various works dedicated to improve the speedup of SPH-flow and to reduce its memory requirements have been completed, allowing large improvements of these critical points.

Figure 13: Performances obtained in 2007-2008 on the IBM Blue Gene /L of the EPFL (left: speedup, right: efficiency).

Unfortunately, this study was limited by the limited available memory intrinsic to the architecture of the Blue Gene /L. This study allowed testing the code on a range of 16 to 256 cores, highlighting important limitations in terms of speedup and memory requirements of the code.

Following this study, various works dedicated to improve the speedup of SPH-flow and to reduce its memory requirements have been completed, allowing large improvements of these critical points.
The figure below shows a speedup obtained during a recent scalability study within a range of 8 to 64 cores, and involving $12 \times 10^6$ particles.

![Figure 14: Speedup and efficiency obtained for a calculation involving $12,10^6$ particles, from 8 to 64 cores](image)

A speedup of 43 is observed when 64 cores are involved in the calculation, showing rather satisfying parallelization efficiency.

In order to complete these tests, the various MPI calls made by SPH-flow during a calculation have been profiled, on an IBM 1600 cluster (Power 4 processors, “Colony” switch). This MPI profiling has been obtained through the option –lmpitrace, based on an intern specific IBM library. Figure 15, shows an example of such a MPI profiling.

While more than 3000 MPI_WAIT calls (dedicated to wait for a send or receive request) occur in the calculation, we observe that latencies are only of about 2 seconds. Moreover, a total communication time of about 5 seconds is observed for a total calculation time of 295 seconds. Communication latencies are thus rather negligible in this parallelization, by communication overlapping. As a result, the efficiency degradation observed in Figure 14 is not attributable to communication latencies.

Thus, a much improved speedup is certainly possible for SPH-flow, by improving parallel-dedicated subroutines (responsible for additional delays), as well as by improving the load-balance of the code presently based on the number of particles only. A key improvement shall be obtained from optimizing memory organization of intrinsically scattered date carried by SPH particles (see section 3). Such improvements will be tried during this project.
### Efficient parallelization strategy for the SPH method

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>1</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>1</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Isend</td>
<td>2994</td>
<td>127132.4</td>
<td>0.212</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>3020</td>
<td>199804.6</td>
<td>0.099</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>3021</td>
<td>0.0</td>
<td>2.062</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>5</td>
<td>0.0</td>
<td>1.272</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>385</td>
<td>24.0</td>
<td>0.551</td>
</tr>
</tbody>
</table>

**total communication time** = 4.982 seconds.
**total elapsed time** = 295.146 seconds.
**user cpu time** = 192.453 seconds.
**system time** = 1.115 seconds.
**maximum memory size** = 263932 KBytes.

<table>
<thead>
<tr>
<th>Message size distributions:</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Isend</td>
<td>3</td>
<td>4.0</td>
<td>0.000</td>
</tr>
<tr>
<td>40</td>
<td>6.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>15.0</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td>40.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>2856.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>5865.1</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>15160.5</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>261</td>
<td>25964.3</td>
<td>0.016</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>43063.7</td>
<td>0.037</td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>115720.0</td>
<td>0.094</td>
<td></td>
</tr>
<tr>
<td>320</td>
<td>19822.0</td>
<td>0.054</td>
<td></td>
</tr>
<tr>
<td>278</td>
<td>358024.5</td>
<td>0.080</td>
<td></td>
</tr>
<tr>
<td>83</td>
<td>551795.8</td>
<td>0.17</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>1900160.0</td>
<td>0.099</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4260060.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>19</td>
<td>4.0</td>
<td>0.000</td>
</tr>
<tr>
<td>40</td>
<td>6.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>15.0</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td>40.0</td>
<td>0.006</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1440.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td>3955.6</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4800.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>88</td>
<td>11454.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>259</td>
<td>24863.5</td>
<td>0.003</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>46167.1</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>91966.1</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>360</td>
<td>189428.0</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>319</td>
<td>347700.0</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>91904.8</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>1295926.9</td>
<td>0.017</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>9914000.0</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>1</td>
<td>4.0</td>
<td>0.314</td>
</tr>
<tr>
<td>689</td>
<td>24.0</td>
<td>0.531</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1072.0</td>
<td>0.001</td>
<td></td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>1</td>
<td>4.0</td>
<td>0.532</td>
</tr>
<tr>
<td>689</td>
<td>24.0</td>
<td>0.019</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 15:** MPI profiling result.
2.4. ASPHODEL parallelization features

The ASPHODEL flow solver is parallelized using Message Passing Interface (MPI) to be run on clusters and super-computers. Two versions of the code should be distinguished.

The first version uses a naïve and only partial domain decomposition. The particle dataset is not split on subsets attributed to processors, but all processors have in their local memory all data related to all calculation points. Only the search for neighbours and the computation of interactions between neighbouring particles are parallelized. In order to implement an efficient parallel neighbour search the domain decomposition uses the simple space filling curve in Figure 10. Grid cells are indexed dimension by dimension and the particles inside are also indexed in the same order. Sub-domains are then created as sets of successive cells according to this indexing pattern, starting with the unattributed grid cell of lowest index and adding new cells until a given number of particles inside the sub-domain has been reached. By this mean all particles inside a cell belong to the same sub-domain. Figure 10 illustrates this procedure to divide the computational domain onto sub-domains. As an example the speed-up obtained with up to 16 processors on a case involving 200’000 particles is shown on Figure 10.

![Figure 10](image)

<table>
<thead>
<tr>
<th>Np</th>
<th>CPU time [s]</th>
<th>Speed-up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4332</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>2160</td>
<td>2.01</td>
<td>1.005</td>
</tr>
<tr>
<td>4</td>
<td>1116</td>
<td>3.88</td>
<td>0.97</td>
</tr>
<tr>
<td>8</td>
<td>576</td>
<td>7.52</td>
<td>0.94</td>
</tr>
<tr>
<td>16</td>
<td>322</td>
<td>13.45</td>
<td>0.84</td>
</tr>
</tbody>
</table>

Figure 10: Right: simple domain decomposition in ASPHODEL_V1, grid cells are described dimension by dimension. Left: Measure of speed-up and efficiency for ASPHODEL_V1.

A quick analysis of this strategy shows some conceptual errors. Domain decomposition is very simple, and the particles themselves are not scattered among processors. Instead all processors keep an up-to-date array of particles, which means that memory requirement per processor increases linearly with the problem size. Another drawback is that global MPI communications are used, which tend to saturate the bandwidth.

The second version of ASPHODEL tries to compensate for the identified shortcomings of version 1. A real domain decomposition is applied, aiming at optimizing memory occupancy on each computing node. Computational domain is first divided onto smaller subsets which are attributed to processes. These processes are made independent and interact only at boundaries, information being transferred across an overlapping region. In practice particles lying inside the overlapping region are communicated to the neighbour process where they are treated as “virtual” particles. That means that they give contributions to the equation of motion of local particles but are not updated. These communications are repeated at each time step, using MPI non-blocking point-to-point communications. The packing and unpacking of messages has to be carefully conducted, the complexity of the communication scheme being O(Np), Np being the number of processes. In order to simplify the connectivity between processes, sub-domains have a rectangular shape and are obtained by recursive dichotomy (see Figure 11). Compared to version one, this domain decomposition is applied earlier in the succession of tasks, prior to the construction of the regular grid.
Because of their motion, particles can change sub-domain and this is also explicitly managed by packing all these particles and using MPI non-blocking point-to-point communications. After an evolution period the evolution of the numbers of particles in sub-domains lead to a load imbalance among processes. A given criterion then triggers a new domain decomposition.

*Figure 11: decomposition in ASPHODEL_V2 by recursive dichotomy.*
3. Efficient parallelization of SPH codes on modern architectures

After the hardware/software links analyzed in Section 1 and the SPH formalism specificities for parallelization analyzed in Section 2, this third section is dedicated to providing general recommendations concerning highly efficient designs of parallel SPH models.

3.1. Efficient inter-node communications on CPU clusters

As introduced in Section 2, key aspects have to be considered to obtain the best performance on a CPU network where communications are made by using the MPI library.

3.1.1. Domain decomposition and load-balance control

The SPH method being of explicit resolution, the domain decomposition strategy is best suited to achieve efficient inter-node parallelization based on MPI. Two types of sub-domain creation exist in literature. The first one, most commonly used, is based on recursive cuts of the sub-domains (ORB: orthogonal recursive bisection). The second is based on the use of an oct-tree structure, coupled with the Peano-Hilbert space filling curve method (as for Gadget-2). The ORB method already exists in both parallel SPH codes of NextMuSE (SPH-flow and Asphodel). However, further investigations will be conducted during the project to estimate the potential gains to expect from oct-tree and/or space filling curves approach.

The domain decomposition aims at distributing workloads on a processor network. However, in addition to this procedure a real-time load-balance control is required given the Lagrangian motion of SPH particles through the sub-domains generated. Thus, particular attention must be paid to the monitoring and adjustment of the load-balance. In this respect, a reliable measure of the CPU times of each process is necessary. Presently, this time is estimated indirectly through the number of calculation points each processor deals with, but in the general case (variable size of calculation points, shape of the domain of the considered process, etc.), this is not sufficient to ensure an accurate load-balance. The CPU-time measure must thus be embedded in the code in a reliable way. Further, to preserve a homogeneous workload repartition across the processor network in time, specific procedures
aiming at moving the boundaries of the sub-domains for them to follow the calculation domain evolution will have to be developed. Presently, a redistribution procedure is used periodically (same as the one done initially) which cost is not very high but still not negligible. An enhanced procedure could be widely inspired from the works of Fleisser et al. [2] which where adaptive procedures are proposed. This could then permit to optimally address heterogeneous architectures (networks composed of different processor performances).

3.1.2. Importance/efficiency influence of masking MPI communication latencies

Section 2 highlighted the fact that SPH codes based on MPI standard usually deal with point-to-point communications. Collective communications are rather unusual. Given the SPH formalism, these communications typically transmit very large amounts of data. Note that data size strongly affects communication delays. Furthermore, communication latencies are generally responsible for very large efficiency degradations of the overall parallel code.

Thus, in addition to domain decomposition and load-balance control, a correct MPI parallelization is obtained only if a special care is paid to cancel (wherever possible) or lower communication latencies. The SPH formalism presents a large number of costly particle-to-particle interactions, combined with an explicit resolution. The experience of NextMuSE partners suggests that communication overlapping is thus relatively straightforward by masking the communications behind these costly calculations, provided that the code structure is thought in this way. Depending on the numerical choices made for the code, loops may be arranged so as to allow such overlapping MPI communications. It depends however on the exact nature of the code, itself depending on the equation solved and the SPH integration scheme chosen. During the project, a better assessment of the efficiency of the present SPH codes in masking these communication latencies will be achieved.

The conclusion regarding the efficiency of inter-node communications on CPU clusters is thus that present implementations are rather good in terms of load-balance and communication latencies, even though a better assessment has to be achieved and enhancements to be realized. This conclusion is valid for rather heterogeneous architectures composed of CPUs with few cores. However, the present architectures are more and more multi-threading oriented, and this will strongly influence the parallelization efficiency, requiring additional work.

3.2. Multi-threading and efficient SPH parallelization

This section is devoted to a discussion of various pre-requisites necessary for the parallelization of multi-threaded SPH codes (OpenMP directives and GPU based implementation). Different aspects intrinsic to the SPH algorithms are highlighted here, in connection with the management of the memory. Indeed, among all the precautions necessary to obtain a good speedup on a multi-threaded SPH application, cache management aspects are particularly important.
3.2.1. Memory access tests and new developments

A special attention must be paid to how data are stored in memory, and generally speaking, one must keep in mind that when a datum is requested, it is always preferable to make this datum already present in the cache (cache hits). The SPH method is based on a Lagrangian particle formalism which need to deal with possibly strongly disordered particles, that is to say to deal with very scattered data in the memory bank. Actually, even if initial particles arrangement in memory corresponds to their physical neighbourhood, this will no longer be true after some evolution time. This scattered data in memory quickly results in cache misses which seriously affect the overall code efficiency (even in the case of a sequential code). To avoid this, the code should be designed so that data in memory is organized as judiciously as possible.

Some preliminary tests were achieved by ECN and HO during this project. These tests used a simplified version of a SPH code (reduced to particle-to-particle interaction calculations), and aimed at disposing the particle data in different ways in the memory bank (multiple or single particle data arrays, various data order in memory), this without changing the code itself. Large differences were observed. In particular, these tests revealed that the use of a single array seems preferable, by drastically decreasing the computational time.

However, given the large amount of data assigned to each particle, it is difficult to place a large number of particles in the cache, resulting in additional memory access delays. A solution to this problem could consist for instance in cutting the particle list into several different arrays, and then in breaking each subroutine accessing these data into several parts, in order to optimize each cache access. Complementary tests will confirm or infirm these ideas.

Concerning memory data contiguity, tests were made in which the particles were re-ordered periodically in memory following their physical neighborhood, as illustrated in the figure below. Again, gains of several tens of % were obtained. A way to optimize this is to use a particle sorting algorithm based on Hilbert space filling curve method. This will be tested as well.

3.2.2. Recommendations regarding OpenMP algorithms

OpenMP is a directive based language for expressing parallelism on shared memory architectures. F77, F90, C and C++ are all supported by this formalism. It is based on the principle of fork and join: the program starts as a sequential one, but can fork into a desired number of threads, and then join to become a master task again, and so on as necessary.
This language is adapted to modern architectures composed of multi-core nodes (and even “manycores” soon), each thread being addressed to a core. As a consequence, when a fork appears, each thread has an access to the shared memory bank, so that great care must be paid to the memory accesses occurring along the calculation.

In addition to the problem of scattered data/caches misses above mentioned, the use of multi-threading imply the possibility of accessing the same data by several threads. In the best cases, this “concurrence” between threads will result in poor efficiency by occurrence of data access conflict. In the worst case, it will result in computational errors. Thus, adapting a code to multi-threading implies its adaptation in such a way that data in memory could not be accessed by several threads at the same time.

A classical practice in sequential SPH codes consists in computing the reciprocal contributions of a particle-to-particle interaction in a single way. This practice is not possible anymore in the context of an OpenMP multi-threading, since it allows several threads to access the same data in the memory bank during this reciprocal interaction calculation, for example:

```
Particle-to-particle       Thread       Particle-to-particle       Thread
1 < -- > 2                     2 < -- > 3
1 < -- > 3                     2 < -- > 4
1 < -- > 4

Thread 1

Thread 2
```

In the above case, both threads 1 and 2 could modify particles 2, 3 and/or 4 data at the same time, which is to be avoided. Thus, a possible strategy may consist in addressing the following way:

```
Particle-to-particle       Thread       Particle-to-particle       Thread
1 < --  2                           2 < --  3
1 < --  3                         2 < --  4
1 < --  4                         2 < --  5

...etc
```

With such an algorithm, each thread changes one particle data at the time. When a thread has finished computing its particle, a new particle (that was not computed by another thread) is addressed to it, and so on until all particles have been treated.

Other possibilities exist for thread/particle addressing, and various solutions will be tested during the project to find the best compromise between the CPU time required to sort particle interactions and efficiency loss due to wrong particle per thread repartitions.

Note that implementing such algorithms has large consequences on the overall SPH code structure, resulting in quite a large amount of work.

### 3.2.3. Recommendations regarding GPU algorithms

The GPU implementation of a SPH code generally requires the same constraints as those mentioned above for the OpenMP implementation, since the GPU applications behaves as multi-threaded ones. The main difference relies in the total number of threads involved.
Indeed, a classical number of threads on GPU devices is 240, whereas OpenMP threads are generally in the range of 8 to 16. Thus, the main difficulty compared to OpenMP implementations relies in the management (without as low efficiency decrease as possible) of such a number of threads. Additional difficulty arise when one wants to do calculations not purely on a GPU, but on more cores requiring communications through CPU since GPU are not directly interconnectable up to now, cf. §3.3.

3.2.3.1. First attempts of SPH-flow GPU porting

CAPS Entreprise is a French company developing the “Hybrid Multicore Parallel Programming” (HMPP) technology. This programming environment is a directive-based compiler dedicated to build parallel GPU accelerated applications. It targets NVIDIA as well as AMD/ATI GPUs.

To make a first attempt of GPU porting, HydrOcean applied to the call made by GENCI (National French HPC infrastructures) and CAPS to port innovative software on GPU. The application was selected and the porting started in December 2009. In order to match with the requirements to achieve an efficient GPU porting of SPH-Flow, some changes were made within SPH-Flow. Actually, while this software was based on pointer arrays, all variables are now stored into classical arrays, allowing their optimized treatment by GPU devices.

A CPU-GPU version of SPH-flow based on the technology developed by CAPS Entreprise is under development. Preliminary results are encouraging, showing interesting levels of speedup on parts of the code (of the order of 10). First results of porting of the full code will be available in the next weeks.

3.2.3.2. First attempts of Asphodel GPU porting

The second version of ASPHODEL has been used to test a GPU implementation of the SPH algorithm using NVIDIA CUDA. This was exploratory work to assess the benefit of using this architecture and to measure the amount of work that should be dedicated to this porting.

The work focused on the porting of the piece of the algorithm that computes the point-to-point interactions and summation, which is known to be the most compute intense part. In the retained approach, each thread on the GPU is responsible for the computation of all interactions of one particle, and all the threads within a block are linked to particles of the same cell. Data related to these particles are copied in shared memory, so that a fairly good reuse of shared memory is achieved as particles lying in the same cell have a high probability to be neighbours.

A simplified neighbour search is performed on the CPU. The first reason is that the transfer of the complete list of connectivity to the device would saturate device memory and PCI bus bandwidth. The second reason is that the implementation of the neighbour search on GPU is delicate. The simplified procedure consists in sorting particles according to the grid cell they belong to. Then while computing point to point interactions the thread tests all particles lying in surrounding cells. If multiple loops over neighbouring particles are required (depending on the numerical and physical treatments implemented) then particles lying outside the kernel support are uselessly tested many times. However some tests showed that the high computing power of the GPU and the delicate memory management on the device make the simplified neighbour search competitive and sufficient in a first approach.
A lot of work on data structures has been performed in order to improve the data transfer to and from the device. On the CPU side data are arranged in a structure of arrays and on the GPU side they are arranged on arrays.

In the end the speed-up obtained on this piece of the global algorithm is between five and six. This shows that a GPU implementation of ASPHODEL is of high value and that great speedups can be obtained with GPUs. But optimization of the implementation has not been made completely and has revealed cumbersome. In particular designing optimized thread blocks is a compromise between shared memory, device occupancy and number of registers, and it’s hard to find general rules that are effective on any implementation and any simulation case. An important point is also that as much as possible of the algorithm should be ported on the device in order to reduce data transfers which are a real bottleneck. In particular special attention should be paid to an efficient neighbour search implementation on the GPU.

These first actions towards SPH implementations ready for multi-threading thus showed that large modifications have to be achieved within the SPH codes to permit to keep the same level of efficiency as for standard few-core CPU cluster architectures. The memory must be at the centre of the thinking of such implementations, both in terms of code organization and in terms of cache-miss optimization which becomes crucial to code efficiency. The developments started by the partners in these directions already show interesting results. The strategy will be adapted depending on the scalability results.

3.3. Efficient parallelization on hybrid CPU/GPU architectures

This last subsection gives some considerations regarding the optimal parallelization of a SPH code on modern hardware architecture. Thus, a hybrid CPU/GPU architecture has to be considered, CPUs being composed of several computing nodes in the calculation network (i.e. distributed memory), themselves composed of several cores having access to a common memory (i.e. shared memory). Here, the discussion thus naturally revolves around a parallel hybrid distributed and multi-threaded CPU/GPU algorithm.

3.3.1. Hybrid OpenMP/MPI

Current architectures are clearly oriented towards node networks machines (through Infiniband switches or Gigabit Ethernet network), each node being composed of several cores (currently 4 to 8 cores). To fully exploit this type of architecture, applications should ideally be based on shared memory OpenMP directives for multi-cores, in conjunction of MPI communication between nodes.

Thus, provided that the previous recommendations for OpenMP parallelization are taken into account, its implementation in a code previously parallelized using MPI is relatively straightforward. Indeed, the OpenMP formalism is relatively independent of the rest of the implementation, from the point of view of each MPI process.
3.3.2. Recommendations regarding SPH parallelization on hybrid architectures

Another peculiarity of GPU cards is that this device has its own memory, which is separated from the CPU RAM. The only way currently available to allow transmissions from the CPU RAM memory to the GPU relies in the PCI Express bus. This bus displays two main latencies: the first one is based on the transmission initialization (latency independent of the quantity and nature of the data to be communicated). The second one is related to the size of data to be transmitted.

Note that, on the other hand, current architectures do not allow direct communications between multiple GPU cards. However, starting from a MPI parallel code, it is possible to envisage a multi-node hybrid architecture connected by Infiniband switches (using the MPI library), each node being linked with a GPU card via a PCI-express BUS.

This strategy can theoretically allow large calculation speedup. However, two limits exist to such implementations. The first one concerns the CPU-GPU communication latencies that have to be minimized in order to obtain a correct overall speedup. The second one relies in the Amdahl’s law, linking the proportion of code that is accelerated through GPUs with the maximum speedup expected by these devices.

Indeed, this strategy prevents a complete GPU translation of the code. Thus, noting $P$ the proportion of GPU translated code, and $S$ the GPU expected speedup (in the best conditions and taking CPU performances as the reference), the overall expected speedup can be expressed as

$$O_{sp} = \frac{1}{(1-P) + \frac{P}{S}}$$

If we consider for instance that 60% of the code will be addressed to the GPU cards ($P=0.6$), and the expected speedup of the GPU is $S=1000$, this law states that the overall maximum speedup to expect is only 2.49. Figure 12 shows the global trend of the overall speedup versus the expected proportion $P$.
As a result, this law clearly states that a very high percentage of code must be addressed to the GPU cards to obtain a correct overall speedup. Every parallel development should thus take this law into consideration.

To conclude, present SPH implementations relying on optimized MPI communications exhibit a good efficiency on standard CPU clusters with few cores per CPU. This is the case of the parallel codes of the partners of the project. Communication latencies can be masked in an efficient way and are not affecting much the overall performance. These SPH codes have now to be adapted to modern hybrid CPU/GPU architectures with several cores in CPUs and many in GPUs. In these architectures distributed and shared memory are mixed, and efficient parallelization must thus be based on a mixed MPI multi-threaded strategy. The way the code addresses the memory becomes crucial to preserve efficiency; this leads to deep reorganization of the implementation. The GPU portable part must also be large enough to get interesting speedups. All these aspects are presently addressed by NextMuSE partners whose codes are being adapted to these modern architectures following the conclusions drawn from the analysis made in the present document.
4. References


[14] TAU web site http://www.cs.uoregon.edu/research/tau, Department of Computer and Information Science, University of Oregon