Acta Crystallographica Section D
**Biological Crystallography**
ISSN 0907-4449

# Parallel, distributed and GPU computing technologies in single-particle electron microscopy

**Martin Schmeisser, Burkhard C. Heisen, Mario Luettich, Boris Busche, Florian Hauer, Tobias Koske, Karl-Heinz Knauber and Holger Stark***

Max Planck Institute for Biophysical Chemistry, Germany

Correspondence e-mail: hstark1@gwdg.de

Received 10 October 2008
Accepted 27 March 2009

Most known methods for the determination of the structure of macromolecular complexes are limited or at least restricted at some point by their computational demands. Recent developments in information technology such as multicore, parallel and GPU processing can be used to overcome these limitations. In particular, graphics processing units (GPUs), which were originally developed for rendering real-time effects in computer games, are now ubiquitous and provide unprecedented computational power for scientific applications. Each parallel-processing paradigm alone can improve overall performance; the increased computational performance obtained by combining all paradigms, unleashing the full power of today's technology, makes certain applications feasible that were previously virtually impossible. In this article, state-of-the-art paradigms are introduced, the tools and infrastructure needed to apply these paradigms are presented and a state-of-the-art infrastructure and solution strategy for moving scientific applications to the next generation of computer hardware is outlined.

## 1. Introduction

### 1.1. Need for speed

Owing to the ever-increasing speed of data collection in science, computational performance plays a central role in various disciplines in biology and physics. In biology, some of the most compute-intense areas are in the field of structural biology, but new molecular-biology techniques such as 'deep sequencing' also require large computational resources for data processing and data storage. Computational demands in the field of structural biology are especially high for high-resolution structure determination by single-particle electron cryomicroscopy because an ever-larger number of images are currently being used to overcome the resolution limits of this technique. There is certainly some linear computational speed increase in central processing unit (CPU) technology that can be expected in the future. However, most of today's speed increase is already based on multi-core CPU architecture. Certain applications, such as the alignment of large numbers of single-particle cryo-EM images, will require significantly more computational power than the current improvement in CPU technology can offer. In some areas future applications will only be possible if the computational power can be increased by at least two orders of magnitude. An increase in computational power is thus essential to keep up with modern scientific technologies.

### 1.2. How can computational speed be improved?

For many standard applications, for a very long period of time programmers did not have to worry about performance as in recent decades CPU manufacturers have improved the hardware speed sufficiently. In these cases for many years it has been a valid approach to simply wait for the hardware to become faster. Moore's Law (Moore, 1965), which states that processing power doubles every 18 months, turned out to be correct for the entire decade of the 1990s. This was the result of improvements in the gates-per-die count or transistors per area (the main attribute of CPUs that Moore based his law on), the number of instructions executed per time unit (clock speed) and the so-called instruction-level parallelism (ILP), basically meaning the possibility of performing more than just one single operation within the same clock cycle (for example, summing up two registers and copying the result to another register). Today this gratuitous 'free lunch' (Sutter, 2005; Sutter & Larus, 2005) of performance gain is over and in recent years CPU manufacturers have started selling CPUs with more computational cores instead of faster CPUs, as beginning in 2003 the laws of physics put an end to the increase in clock speed. One simple reason for this is that doubling the clock speed also means halving the distance travelled by the electrical signal per clock cycle, which requires the physical size of the CPU to be twice as small. However, reducing the physical dimensions of CPUs is limited by the diffraction limits of the lithographic methods used for chip manufacturing. There are other methods that are used to increase performance that can at least partly compensate for the limited increase in clock speed. These are, for example, sophisticated ILP schemes, speculative execution and branch prediction, which nowadays are the only remaining basis for performance improvement apart from the gate count (Schaller, 1997). These methods are what manufacturers focus on today, resulting in feature-rich CPUs that are additionally equipped with an increasing number of computational cores. While an increased clock cycle automatically speeds up an existing application, this is not the case for additional CPUs or cores. Here, the extent that the application can benefit from additional cores depends on the computational problem, the algorithm used to solve it and the application architecture. Performance improvement is then totally dependent on the programmer, who has to develop optimized code in order to obtain the maximum possible speedup. For the next decade, the limiting factor in performance will be the ability to write and rewrite applications to scale at a rate that keeps up with the rate of core count. Outlining applications for concurrency may be the 'new free lunch' (Wrinn, 2007), the new way of utilizing even higher core counts in the future without having to rewrite the code over and over again.

### 1.3. *Divide et impera*

The extent to which the speed of an algorithm can benefit from multiple CPU cores using distributed or parallel processing depends on several conditions. In the first place, it must be possible to break the problem itself down into smaller subproblems. This 'divide-and-conquer' approach is the main paradigm underlying parallel and distributed computing. The main questions are how many independent chunks of work are ready to be computed at any time and what is the number of CPUs or computational cores that can be harnessed. Usually, a problem or algorithm will consist of a fraction that is serial and cannot be processed in parallel, *i.e.* when a computation depends on a previous computational result. The class of problems termed 'embarrassingly parallel' describes the ideal case when all the computations can be performed independently.

## 2. Hardware architectures and their implications for parallel processing

The building block of all computers is the von Neumann architecture (von Neumann, 1946; see Fig. 1). The main principle is that the memory that the CPU works on contains both program instructions and data to process. This design was known from the very beginning to be theoretically limited. The connection system between the data and the program instructions, the so-called data and instruction bus, can become a communication bottleneck between the processor and the memory. No problems existed as long as processing an instruction was slower than the speed of providing data to process. Until the 1990s, the CPU was the slowest unit in the computer, while the bus speed was still sufficient. At this time, the so-called 'von Neumann bottleneck' was a mere theoretical problem. Starting in the mid-1990s, CPUs clock-speed improvement outran random-access memory (RAM) speed, making actual execution faster than the data feed. Nowadays, therefore, the combination of memory and bus forms the 'von Neumann bottleneck', which is mostly overcome or attenuated by caching, which basically means integrating a fast but small memory directly into the CPU. This is why the CPU speed usually increases in parallel with the length of the cache lines. The single-threaded processing that leads to multiple processes running through the same single-threaded pipe and accessing all their data through a single memory interface led to today's processors having more cache than they do logic. This caching is required in order to keep the high-speed sequential processors fed. Another technique to speed up computations without increasing the clock speed is so-called vector computing. While a regular instruction such as an addition or multiplication usually only affects scalars, vector registers and vector instructions were introduced that perform the same operation on multiple scalars or on vectors within one clock cycle.

### 2.1. If you cannot have a faster CPU then use more of them

Using more than one CPU within one computer is less trivial than it sounds. All the other components such as RAM and buses have to be fit for multi-CPU demands. All theoretically explored or practically established computer architectures can be classified using Flynn's taxonomy (Flynn, 1972). Here, architectures are classified by the number of

instruction streams and data streams (see Fig. 2). The classical von Neumann approach belongs to the single instruction stream, single data stream (SISD) class of architectures. All the other architectures are ways of combining many von Neumann building blocks. The first class of hardware to exploit parallelism and concurrency were hardware shared-memory machines. This means that more than one CPU resides inside the same physical machine, working on the same shared memory. Most common today are the single program, multiple data stream (SPMD) architectures, in which multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep that single instruction, multiple data (SIMD) imposes on different data. This architecture is also referred to as single process, multiple data. This is still the most common style of parallel programming, especially for shared-memory machines within the same hardware node. Another common approach is the multiple program, multiple data (MPMD) model, in which multiple autonomous processors simultaneously run independent programs. Typically, such systems pick one node to be the 'host' (the 'explicit host/node programming model') or 'manager' (the 'manager/worker' strategy) that runs one program that farms out data to all the other nodes, which all run a second program. Those other nodes then return their results directly to the manager.

**2.1.1. Shared-memory computing**. Extensive research has been performed on exploiting different computer architectures in the past. The main paradigm here is so-called shared memory (SM) computing, in which many CPUs share the same RAM. This approach can further be divided into symmetric multiprocessor (SMP) systems, in which the memory address space is the same for all the CPUs, and so-called non-uniform memory access (NUMA), in which every CPU has an own address space, resulting in local memory being accessible faster than remote memory.

**2.1.2. Farming**. The next logical step is to scale out to many physical computers: the so-called nodes. This technique is also known as farming. This is not really a stand-alone hardware architecture but a way of clustering several computers (nodes) to simulate NUMA architecture by passing messages between individual nodes across a network connection using a number of different approaches. So-called dedicated homogeneous clustering combines a fixed number of nodes with exactly identical hardware and thus computational power that are dedicated to the cluster computations. Here, every computational node can be trusted to be permanently available and to have the same processing resources and all nodes are connected *via* high-speed networking to enable point-to-point communication. The other extreme is a completely heterogeneous nondedicated environment in which no resource can be trusted to be available and communication is only possible between individual nodes and a managing server or master node (see Table 1). The most prominent examples for the latter are projects such as SETI@Home or Folding@Home using the BOINC framework (Anderson, 2004) from the realm of volunteer computing.

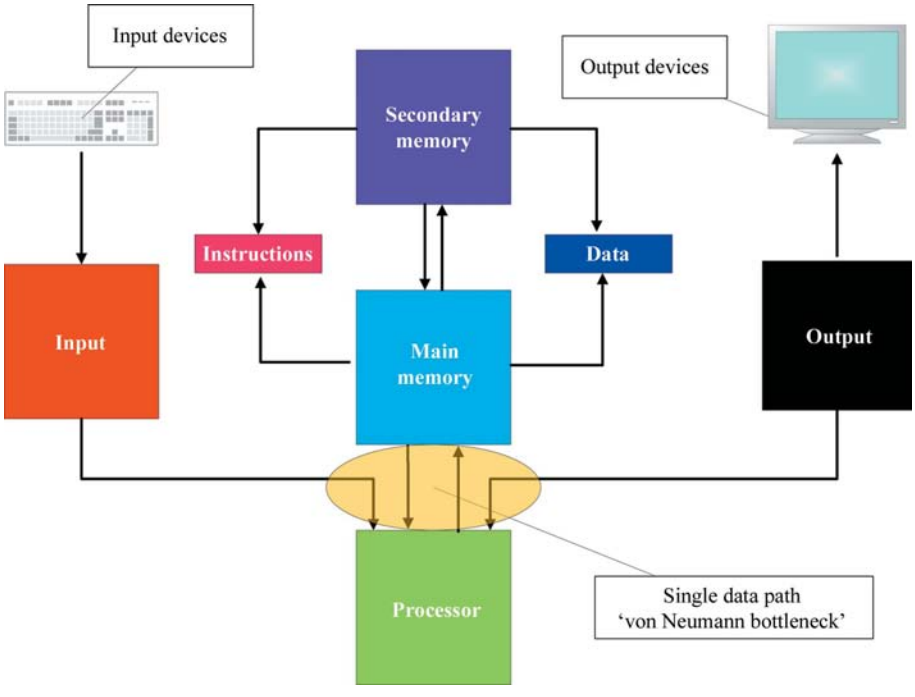


**Figure 1**
The von Neumann architecture is the reference model for programmable computing machines. The memory contains both data to process and program instructions to execute. This architecture was revolutionary in times where the program was either hard-wired or present in the form of punch cards. This architecture is the foundation of most computers today. The interconnection system is called the bus and forms the von Neumann bottleneck.

**2.1.3. GPU computing**. In addition to the multicore revolution (Herlihy & Luchangco, 2008), graphical processing units (GPUs) have recently become more than hardware for displaying graphics and have become a massive parallel processor for general-purpose computing (Buck, 2007*b*). Most observers agree that the GPU is gaining on the CPU as the single most important piece of silicon inside a PC. Moore's law, which states that computing performance doubles every 18 months, is cubed for GPUs. The scientific reason why GPUs overcome and will continue to overcome Moore's Law is the fact that CPUs, as considered by Moore, follow the von Neumann hardware model. The architecture of a single GPU computer unit has a completely different structure, which is called a 'stream processor' in supercomputing (see Fig. 3). Multiple-stream processors obtain their input from other stream processors *via* dedicated pipes. Concurrent input/output (IO) is history on the chip and there is no waiting for data to

**Table 1**
Dedicated *versus* nondedicated clustering.

| Property | Dynamic nodes | Dedicated nodes |
|---|---|---|
| Node performance | Different | Same |
| Node count | Dynamic | Fixed |
| Resources | Dynamic | Fixed |
| Communication | Node-initiated only | Dynamic, even point to point |
| Connectivity | Different (unreliable) | Same (reliable) |
| Reliability | Might never finish processing work unit | Will always finish processing work unit |
| Persistency | Master/controller persistent | Master/controller not persistent |
| Frameworks used | BOINC, SmartTray | MPI, CORBA, .net remoting |
| Application domain | Embarrassing parallel problems (other types with workarounds) | Any kind of project |
| Infrastructure needed | Transactional storage, master/controller for job status | Reliable internode communication, reliable nodes |

process. Moore's law also underestimates the speed at which silicon becomes faster by a considerable margin. With today's advanced fabrication processes, the amount of transistors on a chip doubles every 14 months. Additionally, the processing speed is doubled about every 20 months. Combined with architectural or algorithmic content, a doubling of speed occurs every six months for GPUs. The hardware architecture of the GPU is designed to eliminate the von Neumann bottleneck by devoting more transistors to data processing. Every stream processor has an individual memory interface. Memory-access latency can be further hidden by calculations. The same program can thus execute on many data elements in parallel, unhindered by a single memory interface. The GPU is especially suited for problems that can be expressed as data-parallel computations, in which the same program is executed on many data elements in parallel with a high ratio of arithmetic operations to global memory operations. Because of the parallel execution on multiple data elements, there is a low requirement for flow control. Algorithms that process large data sets which can be treated in parallel can be sped up. Algorithms that cannot be expressed in a data-parallel way, especially those that rely on sophisticated flow control, are not suitable for GPU processing.

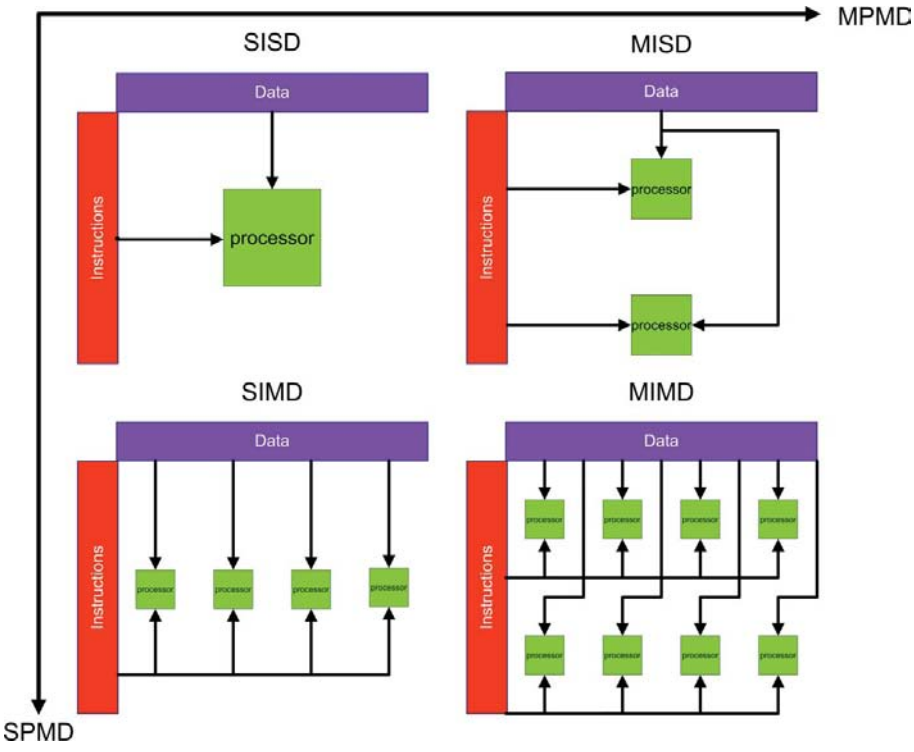


**Figure 2**
Flynn's taxonomy classifies computing architectures by the number of instruction and data streams. The number of processing elements can exceed the number shown in the figure, except for the SISD case. SISD is the classical von Neumann architecture and the classic single-processor system. SIMD computers are also known as array or vector computers, executing the same instruction on a vector of data elements. Examples of MIMD are either local shared-memory systems or distributed systems in which several processors execute different instructions on different data streams. MISD is more theoretical; it can be used for redundant calculations on more than one data stream for error detection.

## 3. Change your code

### 3.1. Software standards for shared-memory and distributed computing

None of the mentioned architectures will speed up any algorithm that is not designed to benefit from concurrent hardware. Generally speaking, extra layers of programming code are needed for the synchronization and communication of many cores or CPUs within one physical computer, the synchronization and communication of different nodes among each other and, last but not least, for using parallel co-processors such as GPUs.

In practice, a reliable locking infrastructure is necessary for any kind of shared-memory programming in order to prevent uncontrolled concurrent access to the same data. Otherwise, memory conflicts will arise that will lead to deadlocks or race conditions. If, for example, a shared counter value is increased, it has to be read first. The new now increased value has to be computed and then finally written back to the shared-memory location. If this is allowed concurrently, unpredictable results will occur, termed race condi-

tions, because the task of increasing is not 'atomic'. The task consists of three operations that must not be interrupted. Locking the counter until the whole task is completed can prevent race conditions. This may lead to deadlocks where one task waits for the other and *vice versa*. The programmer thus needs to rely on combinations of 'locks' and conditions such as 'semaphores' or 'monitors' to prevent concurrent access to the same shared data. This approach enables the developer to treat sections of code as 'atomic' uninterruptible instructions in order to simplify reasoning about instruction interactions.

One shortcoming is that the decision whether coarse-grained locking or fine-grained locking should be used is totally up to the developer. Coarse-grained locking means that a single lock protects a whole data structure. This is simple to implement but permits little or no concurrency. In contrast, fine-grained locking, in which a lock is associated with each component of the data structure, is complicated to implement and may lead to a larger parallelization overhead. The best possible solution will only be valid for a single problem and hardware, making it very difficult to write scalable and portable code at the same time. Fortunately, there are various subtle nonblocking algorithms for standard problems and data structures (Herlihy, 1991). These facilitate standard applications without the risk of deadlocks or race conditions.

Another paradigm that has gained importance in distributed computing is termed the 'transactional memory' approach (Herlihy & Moss, 1993). Here, the possible clashes of shared-memory computing are overcome by the so-called transaction paradigm, which originated in the field of database system design. It enables the developer to mark a designated region of code as a transaction that is then executed atomically by a part of the system. A distributed transaction controller is

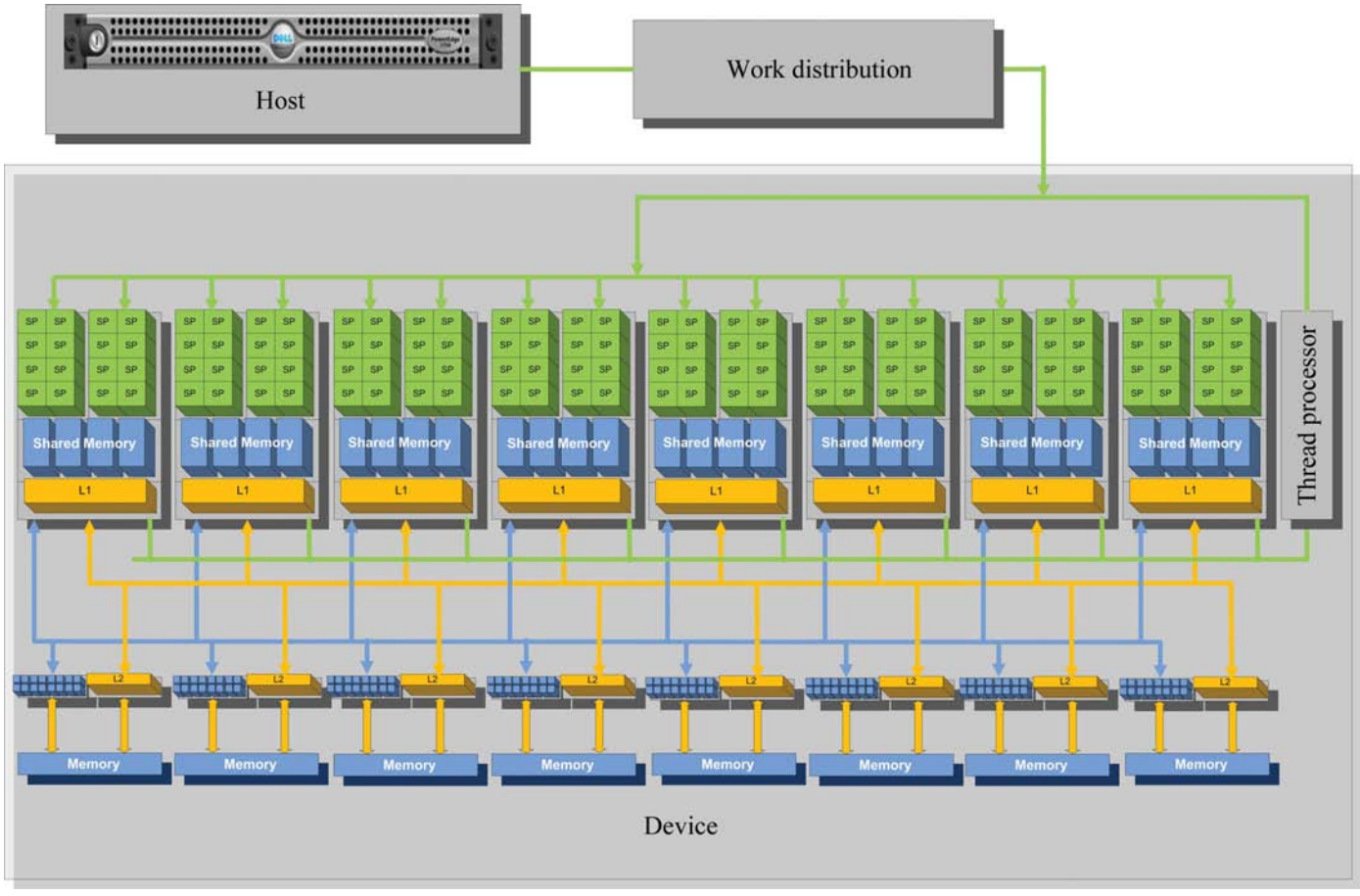


**Figure 3**
Example of modern GPU hardware architecture (modified from Lefohn *et al.*, 2008). The von Neumann bottleneck formed by a single memory interface is eliminated. Each green square represents a scalar processor grouped within an array of streaming multiprocessors. Memory is arranged in three logical levels. Global memory (the lowest level in the figure) can be accessed by all streaming multiprocessors through individual memory interfaces. Different types of memory exist representing the CUDA programming model: thread local, intra-thread block-shared and globally shared memory. This logical hierarchy is mapped to hardware design. Thread local memory is implemented in registers residing within the multiprocessors, which are mapped to individual SPs (not shown). Additionally, dynamic random-access memory (DRAM) can be allocated as private local memory per thread. Intra-thread block-shared memory is implemented as a fast parallel data cache that is integrated in the multiprocessors. Global memory is implemented as DRAM separated into read-only and read/write regions. Two levels of caching accelerate access to global memory. L1 is a read-only cache that is shared by all SPs and speeds up reads from the constant memory space L2, which is a read-only region of global device memory. The caching mechanism is implemented per multiprocessor to eliminate the von Neumann bottleneck. A hardware mechanism, the SIMT controller for the creation of threads and the context switching between threads (work distribution), makes the single instruction multiple threads (SIMT) approach feasible. Currently, up to 12 000 threads can be executed with virtually no overhead.

in control of locks, monitors and semaphores. A transaction that cannot be executed is rolled back and its effects are discarded. The underlying system will probably also use locks or nonblocking algorithms for the implementation of transactions, but the complexity is encapsulated and remains invisible to the application programmer.

Industry-standard solution libraries exist for the different levels of concurrency. For shared-memory computing within one node, today's *de facto* standard is open multi-processing or OpenMP (Dagum & Menon, 1998), which is a framework for the parallelization of C/C++ and Fortran at the compiler level. It allows the programmer to mark sections as parallelizable. The compiler and runtime will then be able to unroll a loop and distribute it across given CPUs or cores for parallel execution. There is also a *de facto* standard for message passing between nodes, which is called the message-passing interface (MPI; Park & Hariri, 1997). This is a fast method for copying data between computational nodes and aggregating results from computational nodes. MPI's predecessor with common goals was PVM: the parallel virtual machine (Beguelin *et al.*, 1991). Of course, MPI and OpenMP can be mixed to parallelize across multiple CPUs within multiple nodes (Smith & Bull, 2001). Several frameworks exist for the programming of GPUs, which differ regarding the hardware manufacturer of the GPU used.

### 3.2. Software standards for GPU programming

As mentioned previously, the GPU is now a massive multicore parallel or streaming processor and can be used for general-purpose computing. Especially in image processing and three-dimensional reconstruction, a tremendous amount of effort has been made to speed up common problems (Bilbao-Castro *et al.*, 2004) and several libraries currently exist that exploit parallel processing features for common tasks. The most prominent example is the discrete fast Fourier transformation (FFT), which can now be sped up by up to 40 times on the GPU in comparison to CPUs (Govindaraju *et al.*, 2008). An illustrative summary of performance gain on image-processing algorithms that will be especially interesting to crystallographers can be found in Castaño-Díez *et al.* (2008). In the early days, programming the GPU meant 'abusing' graphics primitives for general-purpose computing (Moreland & Angel, 2003). Since GPUs were originally peripheral devices, computations have to be initialized and controlled by the device-driver software (Mark *et al.*, 2003) and the graphics hardware producer defines the application programmer interfaces (Pharr & Fernando, 2005). Since today's graphics-card market consists of mainly two companies, ATI and NVIDIA, there are unfortunately two frameworks depending on the GPU brand used: CUDA for NVIDIA (Buck, 2007*a*) and CTM/Brook+ for ATI (Hensley, 2007). In the future, a standard interface for using the GPU for general-purpose computations will hopefully be part of the operating system or compilers will be able to generate the necessary code automatically as is performed with OpenMP language extensions. Apple has announced that its next operating system will support OpenCL, a C/C++ extension similar to CUDA, as an essential building block which facilitates execution on any given multicore CPU or GPU hardware, in 2009 (Munshi, 2008). OpenCL will also be available as an open standard for other platforms, including Linux and Windows. Additionally, Microsoft Research, also steering towards hardware-independent solutions, recently conveyed a parallel programming language called BSGP that is likely to replace graphics hardware manufacturer-dependent solutions at least for the Windows platform (Hou *et al.*, 2008). Another break from hardware dependency might involve MCUDA (Stratton *et al.*, 2008), which also enables code written for graphics hardware to be efficiently executed on any given multi-core hardware.

## 4. Historical limitations alleged by Amdahl's law

Until the 1990s, everything but the single instruction-stream approach was mostly scientific and experimental because even embarrassingly parallel application scaling capabilities were thought to be limited by Amdahl's law (Amdahl, 1967; see Fig. 4). This law seemed to be a fundamental limit on how much concurrent code can be sped up. Applied to parallel computing, this law describes the change of total speedup given different numbers of processing cores. Following Amdahl's conclusion, the theoretical maximum speedup is limited by the portion of the program that cannot be made parallel (*i.e.* the serial part). If $p$ is the amount of time spent (by a serial processor) on parts of the program that can be computed in parallel, the maximum speedup using $N$ processing cores is $\text{speedup}_{\text{Amdahl}} = 1/[s + (p/N)]$, where $s = 1 - p$, assuming a total execution time of 1. Briefly, this law renders the usage of ever-increasing numbers of processors useless as soon as $p$ is significantly smaller than 1. If, for example, $p$ is 0.95 then $s$ is 0.05 and the maximum speedup that theoretically could be achieved is limited to 20 (even with an infinite number of processors).

Fortunately, this law does not hold true in practice. This is mainly because the implicit assumption that $p$ is independent of $N$ is often violated. In practice, the problem size is often scaled with $N$. The $s$ component itself consists of multiple parts. On one hand, the amount of time spent on program startup, serial bottlenecks *etc.*, which contributes to the $s$ component, does not necessarily grow linearly with problem size. On the other hand, the communication between and synchronization of multiple processors (parallelization overhead), which also makes up part of $s$, usually does increase with $N$. Thus, increasing the problem size for a given run time effectively results in an increased value of $p$. Based on the assumptions of constant run time (not problem size) and lack of parallelization overhead, Gustafson stated that the total speedup of a parallel program is given by the Gustafson–Barsis law: $\text{scaledspeedup}_{\text{Gustafson}} = N + (1 - N)s$ (Gustafson, 1988; see Fig. 5). As $N$ increases to infinity, the total work that can be accomplished also increases to infinity. Amdahl's law can be considered as the lower speedup boundary and the Gustafson–Barsis law as the upper speedup boundary.

Depending on the problem itself, its implementation and the hardware infrastructure used, the effective speedup will lie somewhere between Amdahl's and Gustafson's predictions. The larger the size of the problem, the closer the speedup will be to the number of processors.

## 5. Transition to parallel computing: NVIDIA CUDA

Driven by developments in multicore technology in general and GPU technology in particular, a large amount of research on parallel programming techniques has taken place. We will now focus on NVIDIA's compute unified device architecture (CUDA) and outline the way in which parallel programming with CUDA differs from standard programming in C/C++.

Surprisingly, there is not a great difference. CUDA comes with a minimal extension to the C/C++ language to the programmer. In the CUDA approach, normal C code is required for a serial program that calls parallel threads to execute the parallel program parts, termed kernels. The serial portion of the code executes on the host CPU and the parallel kernels execute as parallel threads in thread blocks on the GPU, which in this context is called a 'device'. A parallel kernel may be a single simple function or a complete program, but many threads in parallel, which the programmer has to keep in mind, will execute it.

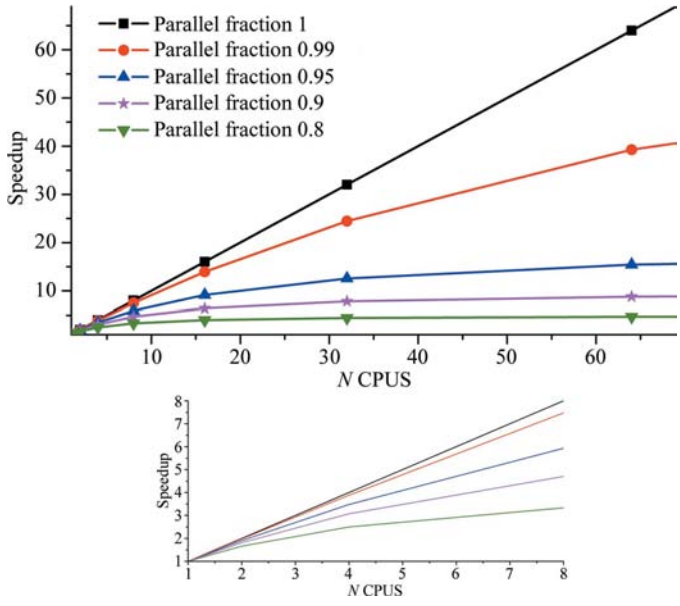


**Figure 4**
Amdahl's law (established in 1967) appeared to be a fundamental limit on the maximum speedup or performance gain achievable by parallelization of an algorithm to multiple CPUs. The fraction of parallelizable code drives this law. The law was formulated under the assumption of a fixed problem size. This assumption means that the ratio between the serial and parallel code fraction is constant. Under this assumption, the efficiency of parallelization is constantly decreasing with the growing number of CPUs. Until recently, this was still accepted in the range between two and eight CPUs. However, when increasing the problem size while not linearly increasing the serial fraction of the algorithm, Amdahl's law has to be re-evaluated and a much higher scalable speedup or performance gain can be achieved.

Threads are organized in thread blocks, which are arrays of threads. Coarse-grained or task parallelism can be realised *via* thread blocks. Fine-grained data parallelism such as in a vector machine can be controlled *via* single threads. In contrast to even the most sophisticated CPU threading models, CUDA threads are extremely lightweight. This property means that the overhead for the creation of threads and context switching between threads is reduced to virtually one GPU clock cycle, in comparison to several hundred CPU clock cycles for CPU threads such as boost threads or p-threads (Nickolls *et al.*, 2008). This technology makes one thread per data element feasible.

Initializing parallel thread blocks, the CUDA runtime and compiler will schedule the execution of up to 12 000 threads simultaneously. The main job of the programmer becomes the parallel decomposition of the problem, analysis of the level of parallelism and distribution to coarse-grained task and fine-grained data-parallel processing. Threads can cooperate, synchronize and share data through shared memory. Thread blocks are organized in grids of blocks with one to many thread blocks per grid. Sequential grids can be used for sequential processing. The developer has to specify the grid and block dimensions and the thread in block dimensions. Currently, a single thread block is limited to 512 threads, which is a hardwired limit and is related to the zero-overhead approach for context switching and thread creation. Synchronization commands also exist at the thread and block level. The command '_synchthreads()' applies a barrier synchronization of all the threads within one block. Addi-

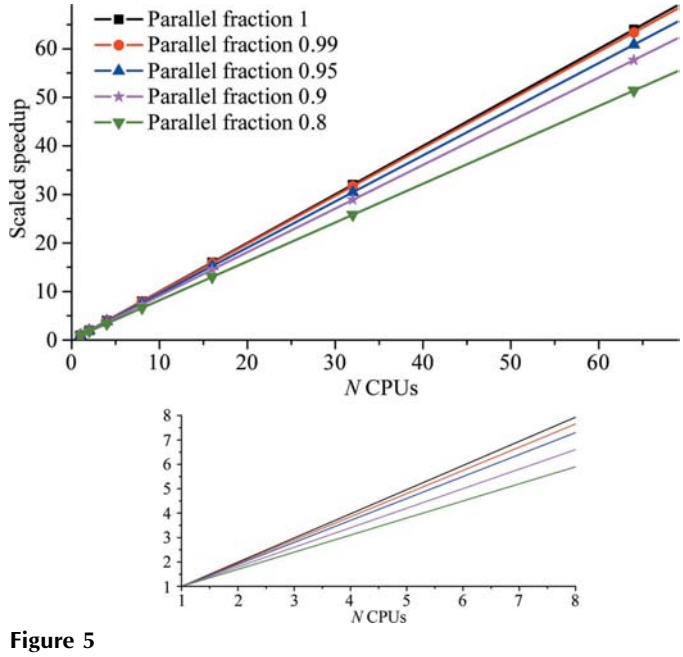


**Figure 5**
Scaled speedup predicted by the Gustafson–Barsis law. Gustafson argued that the sequential portion of a problem is not fixed and does not necessarily grow with problem size. For example, if the serial phase is only an initialization phase and the main calculations can run independently in parallel, then by increasing the problem size the sequential fraction can effectively be reduced to obtain a much larger speedup than that predicted by Amdahl's law.

tionally, there are atomic operations that can be trusted not to be interrupted by other threads.

In addition to the programming model, a new processing model that NVIDIA calls single instruction, multiple thread (SIMT; Lindholm *et al.*, 2008) is implemented on GPUs. Since a GPU is a massively multi-threaded processor array, a new architecture is needed to map thousands of threads running several programs to the streaming multiprocessors (SM). The SM maps each thread to one scalar core (SP) and each scalar thread executes independently with its own instruction addresses and register state. The SM SIMT unit creates, manages, schedules and executes threads in groups of parallel threads termed warps. The threads in a warp start together but are free to branch and execute independently. Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. Full efficiency is realised when all threads of a warp agree on their execution path. Threads that are not on the same execution path will be disabled until they converge back to the main execution path. SIMT architecture is quite similar to SIMD vector organization, as a single instruction controls multiple processing elements. The main difference is that in SIMD the width is exposed to the software, while SIMT behaves like a single thread. SIMT enables programmers to write thread-level parallel code for independent scalar threads as well as data-parallel code for coordinated threads. On the hardware side, the execution of the threads or SIMT (Lindholm *et al.*, 2008) instructions is also very interesting. All 32 threads form a warp of threads that executes one SIMT instruction in parallel. In summary, SIMT encompasses threads while SIMD encompasses data. SIMT offers single thread scalar programming with SIMD efficiency (Nickolls *et al.*, 2008).

## 6. Data-parallel programming

Traditional parallel programming methods often have limited scaling capabilities because the serialization and de-serialization phases required produce a linear growth in serial code if each core has to synchronize with a single core. For point-to-point synchronization between multiple cores, the overhead can even increase as a combinatorial of core count. A paradigm which scales with core count without the need to restructure existing code would be very useful to prevent future core counts from exceeding the level of parallelism an application can scale to. A finer level of granularity has to be identified for parallelism in order to prevent the presence of more processing cores than threads. In the near future, the only type of element that will be available in the same number as cores will be data elements (Boyd, 2008). Therefore, the data-parallel approach looks for fine-grained inner loops within each computational step and parallelizes those with the goal of having one logical core for processing each data element. Ideally, this should take place automatically by the combination of the compiler and the runtime system. There are several programming languages or language extensions to perform this such as High Performance Fortran or Co-Array, but these hide unique aspects of data-parallel programming or the processor hardware. Array-based languages such as Microsoft Research Accelerator (Tarditi *et al.*, 2005, 2006) need a very high level of abstraction and are rarely portable or even available for more than one specific platform. The graphics hardware manufacturers involved are developing new data-parallel languages for their individual hardware: CAL and Brook+ for ATI (Hensley, 2007) and CUDA for NVIDIA (Nickolls *et al.*, 2008). The syntax of these languages provides a direct mapping to hardware that enables specific optimizations and access to hardware features that no other approach allows. The main disadvantage is once again the dependency on the (graphics) platform until hardware and operating-system manufacturers eventually agree on a unified application programmer's interface (API) for GPUs, which might even become available within the near future. For Microsoft Windows systems, proprietary arises from the introduction of DirectX 11 featuring compute-shaders (an API for GPU computing independent of the hardware manufacturer) and additionally a completely new C/C++ extension called BSGP that has just recently been released (Hou *et al.*, 2008). These systems can create and execute parallel kernels graphics hardware independently. For all operating systems and hardware this will soon be available royalty-free owing to OpenCL (Munshi, 2008). Another option might be MCUDA, a CUDA extension that can efficiently run CUDA kernels on any given multicore hardware (Stratton *et al.*, 2008).

## 7. The SmartTray

We will now introduce our contributions to parallel computing and the reasons why we chose various different strategies. We started in 2004 with a set of mostly embarrassingly parallel Fortran applications that were parallelized using MPI. The first thing we noticed was that there were certain scheduling limitations of the MPI approach *via* a 'machines file' in which all the nodes included in the calculations need to be listed. Furthermore, one of the drawbacks of MPI was the insufficient failover model, which is a basic requirement within our department where several people share the same computing resources with very long-running calculations that should not stop in the case of a single node's failure. Thus, there was a need for more dynamic resource allocation than that provided by MPI. Some of the most desired features were (i) the possibility of pausing a long-running job for the benefit of a smaller but highly urgent job, (ii) a mechanism for dynamic resource allocation, namely reassigning nodes to already running jobs, (iii) adding nodes to a running calculation and (iv) a failover mechanism that enables a node to automatically rejoin calculations after solving/encountering a hardware problem. Research on middleware implementing these exact features as an industry standard is currently ongoing (Wang *et al.*, 2008), but was not available in 2004. Another highly desired feature was to include the increasing computational power of standard workstations available locally in our calculations. For example, a secretary's workstation that was

usually utilized only during daytime hours had more computational power than some of our dedicated cluster nodes. To implement the above-mentioned features and to also make use of the nondedicated computational resources, we have developed a new framework which we call 'SmartTray' (Fig. 6). It mainly consists of a self-updating SmartClient running in the system tray of the Windows operating system. The SmartTray Framework, which is still experimental, is based on the master worker approach in combination with transactional memory. A job consists of work packs, which are administered by a master server. Since this master server is implemented as a web service using Microsoft SQL Server 2005, which is a transactional database for storage, the transactional capabilities of the database can also be used to emulate transactional memory by means of high-performance database mechanisms. In contrast to the standard approach, in which the master server initiates communication to the computational nodes, owing to inversion of control from master to node the whole system is kept as dynamic as possible. Initiating the communication on the node rather than on the master server is thus the key issue for inverting the control. Each single node polls work packs from the master server, executes them and returns the results to the master server. In case of node failure, a work pack will not be completely finished in time. The same work pack will then simply be executed again by another node. A recovered node can also restart, contributing to the overall calculations, simply by polling a new work pack. A major advantage of the SmartTray approach is that it is programmed to run hardware detection on each node before entering computations. Based

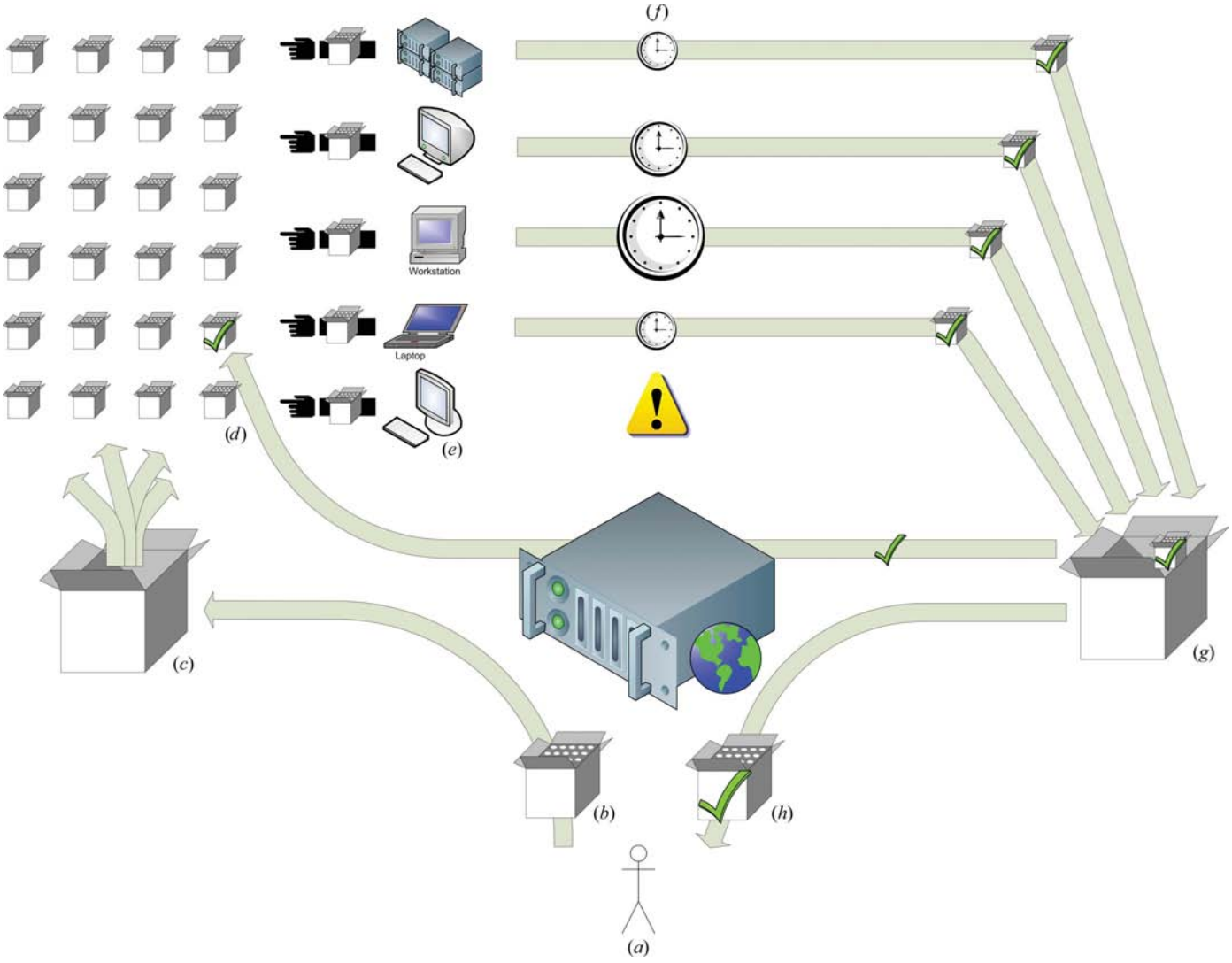


**Figure 6**
The workflow of our SmartTray Framework. A computation is a job (*b*) that consists of multiple work packs (*d*) that can be independently processed in parallel. A work pack contains both the data to process and the computation instructions. The job is created by the user (*a*) and hosted by a master server [represented logically by (*c*) and (*g*)]. All computational nodes, dedicated cluster nodes or volunteer computers (*e*) poll work packs (*d*) from the master server (*c*), execute the calculation (*f*) and send the result to the master server (*g*). When a work pack's result is not returned within a given time-out period (or nodes are idle otherwise), the work pack will be re-dispensed as a failover measure. As a side effect, the master server holds a constantly updated list of nodes that are willing to participate in calculations that includes their hardware features.

on the hardware, the software version of the main logic will then choose the correct mode of execution. In case of a multicore or a multi-CPU node, the OpenMP version will be executed.

For GPU computing, we focused on NVIDIA as a hardware manufacturer and NVIDIA CUDA as a programming language. However, our first experience with GPU programming started before CUDA became available. At that time, programming the GPU was performed by 'abusing' the GPU graphics primitives for general-purpose computations. Here, we started with OpenGL Shading Language (GLSL) because we did not want to restrict ourselves to a specific graphics hardware manufacturer. However, it became evident that hardware independence also leads to a loss of overall performance. For this reason, we subsequently focused on NVIDIA shader language, NVIDIA Cg (Mark *et al.*, 2003), a C extension originally designed for programming visual effects. NVIDIA Cg and Brook (Buck *et al.*, 2004) were thus the first languages for general-purpose computations on graphics cards. While Brook is a platform-independent streaming language with an OpenMP CPU, OpenGL, DirectX 9 and now AMD CTM backend, NVIDIA Cg is proprietary for NVIDIA hardware only. Brook is now called Brook+ and became ATI's CUDA equivalent, lacking an NVIDIA backend. Currently, a hardware manufacturer has to be chosen owing to incompatibilities between the graphics cards currently available on the market. With the choice of manufacturer made, the first applications we ported were FFT calculations and three-dimensional reconstruction techniques such as SIRT and ART. Immediately obtaining a speedup factor of 30 in our three-dimensional reconstruction algorithms on the graphics card motivated us to incorporate the GPU version into our SmartTray approach. SmartTray is now able to detect the graphics card and can then automatically make use of the faster GPU software. We were amongst the first early adaptors after CUDA was launched together with the NVIDIA Tesla unified graphics and computing architecture (Buck, 2007*a*). The Tesla board is the NVIDIA compute card. However, CUDA code also runs on NVIDIA GeForce 8800 consumer graphics boards and we actually originally started software development on the GeForce boards while waiting for the Tesla boards to be delivered. Currently, we have two systems running in our GPU cluster. One system is the NVIDIA Tesla compute server. This server is a rack-mountable one height unit (HU) computer containing four Tesla GPUs. Each Tesla compute server needs another CPU server with two fast PCI express connections. A Tesla compute server and a CPU server are thus mounted alternatively into a 19-inch rack. Apart from this commercial solution, we have also developed a homemade GPU compute server (see Fig. 7). This system features three GPUs, eight CPU cores, six hard disks and memory on two HUs, while the NVIDIA prefab version comes with four GPUSs, no CPU cores, no hard disks and no memory at all on one HU and additionally needs the above-mentioned extra CPU server to run.

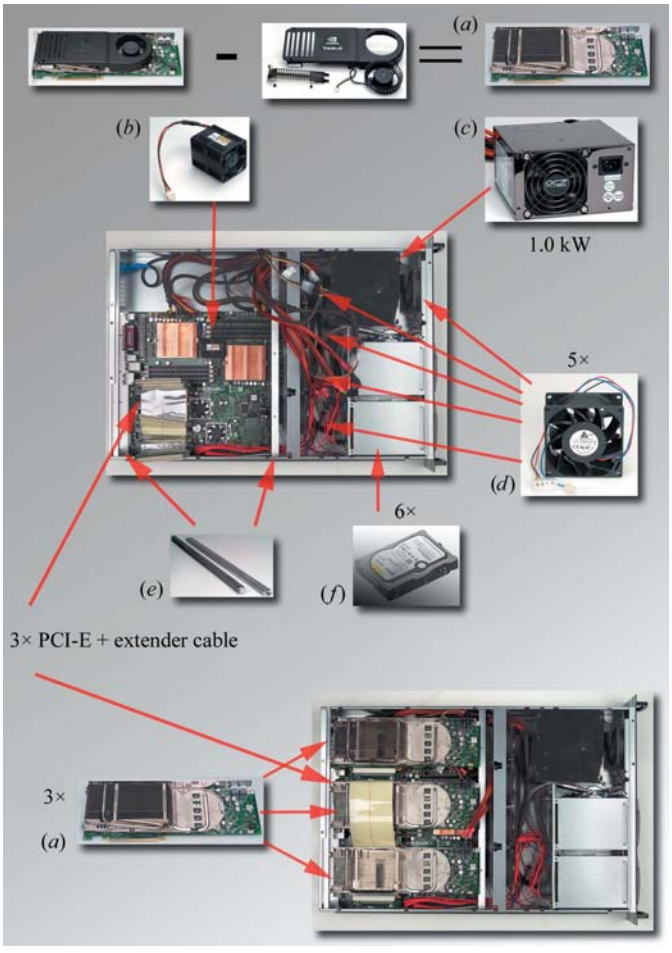


**Figure 7**
The blueprint of our homemade GPU server featuring two CPUs, three GPUs, six hard disks (*f*) and memory on two height units (HU). The overall layout is a sandwich design in a standard 19-inch two HU case. The main board's three PCI-Express slots are extended using extender cables. For cooling purposes a battery of four fans (*d*) is installed between the main board compartment and the hard disk and power supply area. The power supply fan's built-in direction must be inverted and another fan (*d*) must be installed to additionally cool the 1 kW power supply. Additionally, another fan (*b*) is installed to ensure CPU cooling. On top of this, two custom rods (*e*) are mounted to support the GPUs (*a*). For space and thermal reasons, the GPUs used must be stripped from their fans and casings, resulting in the slimmer bare-bones version (*a*). Finally, the GPUs are mounted on the supporting rods and connected to the PCI-E slots *via* the extender cables.

## 8. Getting started with data-parallel processing

Another important factor to consider is the optimized usage of the computational resources. Will there be concurrent tasks that have to be scheduled? What should happen in the case of hardware failure? Will a single node failure affect the whole computation? Should the solution be optimized for performance of an individual problem or for throughput if many people run several computations? These considerations depend strongly on the individual demands, needs and amounts budgeted. The hardware chosen should of course match the software architecture and *vice versa*. Once the hardware has been chosen, the hierarchy of the problem has to

be mapped to the given hardware. Parallelism and its granularity have to be identified and distributed amongst the layers of the application and thus the tiers of the hardware. For every layer or tier the decision has to be made whether standard patterns/libraries which might not give the maximum possible performance gain should be used (this consideration is generally strongly recommended for maintenance and portability reasons) or whether the potential scaling merits hand coding and optimization.

To start initial parallel programming on graphics cards is actually very easy. Along with the software-development kits (SDKs) and toolkits that can be downloaded from the internet for the different graphics cards, it is also possible to download large sets of examples, documentation and even GPU accelerated maths and performance libraries. Obviously, there are some infrastructure requirements for the setup of a large-scale GPU compute cluster for high-performance computing. The main limiting points at the level of GPU clusters are heat generation and power consumption, which make a sophisticated cooling infrastructure indispensable. The computing power per volume and thus the heat generated is maximized to levels far above blade server systems. In our system, we thus use Rittal's liquid-cooling packages (LCP) combined with an in-house water-cooling system to operate the cluster at constant temperature. This system relies on water cooling to cool the air in the otherwise sealed server racks. Heat exchangers are thus required as basic parts of the infrastructure.

From our experiences, the initial development of GPU-exploiting code can best be implemented on a local machine featuring at least one CUDA-enabled device using the CUDA SDK and toolkit. Once the implementation of an algorithm for data-parallel processing, the actual parallel decomposition, is complete, further distribution to several GPUs can be tackled. The approach chosen for this distribution strongly depends on the available resources. If an infrastructure exists where several GPUs are available within one machine, the work distribution to these is best realised using Boost or p-threads. If several machines featuring GPUs are available, the distribution to these is best realised using MPI or a SmartTray-like framework.

## 9. Discussion and outlook

Currently, the preferred trend of increasing CPU core count over execution speed is very likely to continue in the future. Therefore, the next generation of CPUs will again not significantly speed up existing applications if they are not prepared to harness the power of these additional cores. Furthermore, commodity graphics hardware is evolving to become a new generation of massively parallel streaming processors for general-purpose computing. Scientists and developers are challenged to port existing applications to the new data-parallel computing paradigms and to introduce as much concurrency in the algorithms as possible (if possible, using one thread per data element). This strategy guarantees optimum scaling with core count. Additionally, hardware such as GPUs can and should be included at the level of desktop computers and even up to dedicated GPU cluster solutions. Unfortunately, today these new paradigms are still hardware-dependent, but there are initial indications that this will change within the near future (Hou *et al.*, 2008; Munshi, 2008; Stratton *et al.*, 2008). Currently, one still needs to be prepared for the possibility that entering GPU computing may require rewriting the code partially for the next hardware generation. Nevertheless, this can be expected to be rather trivial because the data-parallel approach is not likely to change dramatically over time. The technology exists to harness today's state-of-the-art system architecture for at least one full compute-cluster duty cycle from commissioning to the end of financial amortization. Approaches such as OpenCL, MCUDA or BSGP target these architectures and are also prepared for Intel's and AMD's GPU/CPU developments of hybrid or extreme multicore architectures that will soon enter the market. Larrabee for Intel (Seiler *et al.*, 2008) and Fusion from ATI are targeted to recapture market shares from GPU manufacturers and in visual computing. No matter how powerful GPUs may be for general-purpose computations, they are still a workaround and no panacea. Especially when irregular data structures are needed or scatter–gather operations cannot be avoided, GPU programming is suboptimal because of the fixed function logic. Although a lot of high-performance and high-throughput calculations can be sped up significantly using GPUs, we are currently far from a hardware- and platform-independent technology. The only hope is that hardware manufacturers and/or operating-system manufacturers will agree on a common API that does not depend on the hardware used. In the long run, it is very likely that companies such as Intel or AMD will (re-)enter the data-parallel market. Intel have already released a new compiler generation called High Performance Parallel Optimizer that is capable of auto-vectorizing and auto-parallelization, which utilizes loop unrolling and peeling. A great deal of workloads have been analyzed by Intel, including well known HPC kernels such as 3D-FFT and BLAS3, converging towards a common core of computing (Chen *et al.*, 2008). This approach shows great potential for a common run-time, a common programming model and common compute kernels, data structures and math functions. Once these steps are complete, the code generation for a specific platform being CUDA/CTM or Larrabee/Fusion is a political and marketing issue but not an academic problem. In summary, it is still too early to tell which API or programming model will succeed in the long run, but today's technology can already be utilized. The trend is set: get ready for the data-parallel multicore age (Che *et al.*, 2008) and exploit as much concurrency in algorithms as possible.

Single-particle cryo-EM is certainly one field in which GPU computing currently pays off. Nowadays, electron microscopes allow data collection to be completely automated. In combination with fast and large detectors, these instruments can produce massive amounts of data within a very short amount of time. In the best case, one can expect such microscopes to collect up to several million particle images within a week. The collection of this huge amount of data is only reasonable if

the computer infrastructure needed to process the data is powerful enough to keep up with data collection. In the cryo-EM field it is thus almost impossible to rely only on CPUs. In contrast, the massive speedup that can be achieved on GPUs for most of the embarrassingly parallel applications needed in single-particle cryo-EM image processing makes the computational analysis of millions of images possible on a medium-sized GPU cluster. For compute-intense applications such as single-particle cryo-EM, GPUs are thus the only option simply because of the lack of any other affordable alternatives. In single-particle cryo-EM the most compute-intense applications are image alignment of a large number of randomly oriented macromolecules imaged in the electron microscope and three-dimensional reconstruction using these aligned images. We implemented the alignment of images to reference templates and the subsequent three-dimensional reconstruction in CUDA and ran these on a dedicated GPU Cluster featuring 320 NVIDIA Tesla boards using MPI for the work distribution amongst cluster nodes. Additionally, we ran a dynamic cluster using SmartTray, featuring up to 120 compute nodes with up to 360 CPUs that are partly equipped with CUDA-enabled GPUs. Performance measurements on the dedicated GPU cluster show a speedup of three-dimensional reconstruction using the simultaneous iterative reconstruction technique (SIRT) algorithm by a factor of $\sim$65 comparing one CPU to one GPU using a pixel frame of $64 \times 64$. The speedup of the image alignment is more difficult to measure because the actual alignment algorithms implemented on CPU and GPU are also significantly different. Additionally, there is a significant dependence of the measured speedup on the image size. Using small images (such as $64 \times 64$ pixels) the speedup on the GPU compared with the CPU is almost negligible. However, for high-resolution three-dimensional structure determination high image statistics need to be combined with small pixel sampling. For most macromolecules we thus expect a pixel frame size to be in the range of $256 \times 256$ pixels and $1024 \times 1024$ pixels. Using these larger images the measured speedup of GPU computing is in the range of 10–100 times, allowing the determination of high-resolution three-dimensional structures by single-particle cryo-EM in an acceptable time frame.

The authors would like to thank all members of the Stark laboratory that volunteered to test our software. Additionally, we would like to thank everyone who runs the SmartTray on their computer. Special thanks go to Irene Boettcher-Gajewski and Peter Goldmann for invaluable Photoshop experience and photo shooting.

## References

Amdahl, G. M. (1967). *AFIPS Conf. Proc.* **30**, 483–485.
Anderson, D. P. (2004). *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pp. 4–10. Washington: IEEE Computer Society.
Beguelin, A., Dongarra, J., Geist, A., Manchek, R. & Sunderam, V. (1991). *A User's Guide to PVM Parallel Virtual Machine.* University of Tennessee.
Bilbao-Castro, J. R., Carazo, J. M., Fernández, J. J. & García, I. (2004). *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pp. 96–102. Washington: IEEE Computer Society.
Boyd, C. (2008). *Queue*, **6**, 30–39.
Buck, I. (2007*a*). *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2007 Courses*, course 24, article 6. New York: ACM.
Buck, I. (2007*b*). *Proceedings of the International Symposium on Code Generation and Optimization*, p. 17. Washington: IEEE Computer Society.
Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. & Hanrahan, P. (2004). *ACM Trans. Graph.* **23**, 777–786.
Castaño-Díez, D., Moser, D., Schoenegger, A., Pruggnaller, S. & Frangakis, A. S. (2008). *J. Struct. Biol.* **164**, 153–160.
Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W. & Skadron, K. (2008). *J. Parallel Distrib. Comput.* **68**, 1370–1380.
Chen, Y.-K., Chhugani, J., Dubey, P., Hughes, C. J., Daehyun, K., Kumar, S., Lee, V. W., Nguyen, A. D. & Smelyanskiy, M. (2008). *Proc. IEEE*, **96**, 790–807.
Dagum, L. & Menon, R. (1998). *IEEE Comput. Sci. Eng.* **5**, 46–55.
Flynn, M. J. (1972). *IEEE Trans. Comput.* **21**, 948–960.
Govindaraju, N. K., Lloyd, B., Dotsenko, Y., Smith, B. & Manferdelli, J. (2008). *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, article 2. Piscataway: IEEE Press.
Gustafson, J. (1988). *Commun. ACM*, **31**, 532–533.
Hensley, J. (2007). *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2007 Courses*, course 24, article 7. New York: ACM.
Herlihy, M. (1991). *ACM Trans. Program. Lang. Syst.* **13**, 124–149.
Herlihy, M. & Luchangco, V. (2008). *SIGACT News*, **39**, 62–72.
Herlihy, M. & Moss, J. E. B. (1993). *ACM SIGARCH Comput. Archit. News*, **21**, 289–300.
Hou, Q., Zhou, K. & Guo, B. (2008). *ACM Trans. Graph.* **27**, 1–12.
Lefohn, A., Houston, M., Luebke, D., Olick, J. & Pellacini, F. (2008). *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2008 Classes*, article 18. New York: ACM.
Lindholm, E., Nickolls, J., Oberman, S. & Montrym, J. (2008). *Micro, IEEE*, **28**, 39–55.
Mark, W. R., Glanville, R. S., Akeley, K. & Kilgard, M. J. (2003). *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2003 Papers*, pp. 896–907. New York: ACM.
Moore, G. E. (1965). *Electron. Mag.* **38**, 114–117.
Moreland, K. & Angel, E. (2003). *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 112–119. Aire-la-Ville, Switzerland: Eurographics Association.
Munshi, A. (2008). *OpenCL Parallel Computing on the GPU and CPU.* http://s08.idav.ucdavis.edu/munshi-opencl.pdf.
Neumann, J. von (1946). Reprinted in *IEEE Ann. Hist. Comput.* **3**, 263–273.
Nickolls, J., Buck, I., Garland, M. & Skadron, K. (2008). *Queue*, **6**, 40–53.
Park, S.-Y. & Hariri, S. (1997). *J. Supercomput.* **11**, 159–180.
Pharr, M. & Fernando, R. (2005). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Upper Saddle River: Addison–Wesley Professional.
Schaller, R. R. (1997). *Spectrum, IEEE*, **34**, 52–59.
Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T. & Hanrahan, P. (2008). *ACM Trans. Graph.* **27**, article 18.
Smith, L. & Bull, M. (2001). *Sci. Program.* **9**, 83–98.
Stratton, J. A., Stone, S. S. & Hwu, W. W. (2008). *21st Annual Workshop on Languages and Compilers for Parallel Computing (LCPC'2008).* http://www.gigascale.org/pubs/1328.html.

Sutter, H. (2005). *Dr Dobb's J.* **30**.
Sutter, H. & Larus, J. (2005). *Queue*, **3**, 54–62.
Tarditi, D., Puri, S. & Oglesby, J. (2005). *Accelerator: Simplified Programming of Graphics-Processing Units for General-Purpose Uses via Data-Parallelism.* Technical Report MSR-TR-2004-184. Microsoft Corporation.
Tarditi, D., Puri, S. & Oglesby, J. (2006). *ACM SIGARCH Comput. Archit. News*, **34**, 325–335.
Wang, C., Mueller, F., Engelmann, C. & Scott, S. L. (2008). *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, article 64. Piscataway: IEEE Press.
Wrinn, M. (2007). *Is the Free Lunch Really Over? Scalability in Many-core Systems.* Intel White Paper. http://software.intel.com/file/7354.