J. Parallel Distrib. Comput. 73 (2013) 1613–1626

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

A block-asynchronous relaxation method for graphics processing units

Hartwig Anzt [a,*], Stanimire Tomov [b], Jack Dongarra [b,c,d], Vincent Heuveline [a]

[a] Karlsruhe Institute of Technology, Germany
[b] University of Tennessee Knoxville, USA
[c] Oak Ridge National Laboratory, USA
[d] University of Manchester, UK

HIGHLIGHTS

- Block-asynchronous relaxation on GPU-accelerated systems.
- Method's high iteration rate compensates a low convergence rate (competitive to CG).
- GPU thread-block scheduling reduces non-deterministic behavior and stochastic noise.
- Analysis of different communication strategies for multi-GPU usage.
- Block-asynchronous relaxation inherently tolerant to hardware error.

ARTICLE INFO

Article history:
Received 14 October 2012
Received in revised form
29 April 2013
Accepted 21 May 2013
Available online 6 June 2013

Keywords:
Asynchronous relaxation
Chaotic iteration
Graphics processing units (GPUs)
Jacobi method

ABSTRACT

In this paper, we analyze the potential of asynchronous relaxation methods on Graphics Processing Units (GPUs). We develop asynchronous iteration algorithms in CUDA and compare them with parallel implementations of synchronous relaxation methods on CPU- or GPU-based systems. For a set of test matrices from UFMC we investigate convergence behavior, performance and tolerance to hardware failure. We observe that even for our most basic asynchronous relaxation scheme, the method can efficiently leverage the GPUs computing power and is, despite its lower convergence rate compared to the Gauss–Seidel relaxation, still able to provide solution approximations of certain accuracy in considerably shorter time than Gauss–Seidel running on CPUs- or GPU-based Jacobi. Hence, it overcompensates for the slower convergence by exploiting the scalability and the good fit of the asynchronous schemes for the highly parallel GPU architectures. Further, enhancing the most basic asynchronous approach with hybrid schemes–using multiple iterations within the "subdomain" handled by a GPU thread block–we manage to not only recover the loss of global convergence but often accelerate convergence of up to two times, while keeping the execution time of a global iteration practically the same. The combination with the advantageous properties of asynchronous iteration methods with respect to hardware failure identifies the high potential of the asynchronous methods for Exascale computing.

1. Introduction

The latest developments in hardware architectures show an enormous increase in the number of processing units (computing cores) that form one processor. The reason for this varies from various physical limitations to energy minimization considerations that are at odds with further scaling up of processor' frequencies—the basic acceleration method used in the architecture designs for the last decades [15]. Only by merging multiple processing units into one processor does further acceleration seem to be possible. One example where this core gathering is carried to extremes is the GPU. The current high-end products of the leading GPU providers consist of 2880 CUDA cores for the NVIDIA Tesla K20 and 3072 stream processors for the Northern Islands generation from ATI. While the original purpose of GPUs was graphics processing, their enormous computing power also suggests the usage as accelerators when performing parallel computations. Yet, the design and characteristics of these devices pose some challenges for their efficient use. In particular, since the synchronization between the individual processing units usually triggers considerable overhead and limits the performance to the slowest component, it is attractive to employ algorithms that have a high degree of parallelism and only very few synchronization points.

* Corresponding author.
E-mail addresses: hanzt@icl.utk.edu, hartwig.anzt@googlemail.com (H. Anzt).

0743-7315/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jpdc.2013.05.008

On the other hand, numerical algorithms usually require this synchronization. For example, when solving linear systems of equations with iterative methods like the Conjugate Gradient or GMRES, the parallelism is usually limited to the matrix–vector and the vector–vector operations (with synchronization required between them) [12,33,34]. Also, methods that are based on component-wise updates like Jacobi or Gauss–Seidel have synchronization between the iteration steps [26,13]: no component is updated twice (or more) before all other components are updated. Still, it is possible to ignore these synchronization steps, which will result in a chaotic or asynchronous iteration process. Despite the fact that the numerical robustness and convergence properties severely suffer from this chaotic behavior, they may be interesting for specific applications, since the absence of synchronization points make them perfect candidates for highly parallel hardware platforms. The result is a trade-off: while the algorithm's convergence may suffer from the asynchronism, the performance can benefit from the superior scalability. Additionally, the asynchronous nature implies a high tolerance to hardware failure, which is one of the key properties required for numerics suitable for future hardware.

In this paper, we want to analyze the potential of employing asynchronous iteration methods on GPUs by analyzing convergence behavior and time-to-solution when iteratively solving linear systems of equations. We split this paper into the following parts. First, we will shortly recall the mathematical idea of the Jacobi iteration method and derive the component-wise iteration algorithm. Then the idea of an asynchronous relaxation method is derived, and some basic characteristics concerning the convergence demands are summarized. The section about the experiment framework will first provide information about the linear systems of equations we target. The matrices affiliated with the systems are taken from the University of Florida matrix collection. Then we describe the asynchronous iteration method for GPUs and multi-GPUs that we designed. In the following section we analyze the experiment results with focus on the convergence behavior, iteration times and fault-tolerance for the different matrix systems. In Section 5 we summarize the results and provide an outlook about future work in this field.

2. Mathematical background

2.1. Jacobi method

The Jacobi method is an iterative algorithm for finding the approximate solution for a linear system of equations

$$Ax = b, \tag{1}$$

where A is strictly or irreducibly diagonally dominant. One can rewrite the system as $(L + D + U)x = b$ where D denotes the diagonal entries of A while L and U denote the lower and upper triangular part of A, respectively. Using the form $Dx = b-(L+U)x$, the Jacobi method is derived as an iterative scheme

$$x^{k+1} = D^{-1}(b - (L + U)x^k).$$

Denoting the error at iteration $k + 1$ by $e^{k+1} \equiv x^{k+1} - x^*$ (x^* may denote the exact solution of (1)), this scheme can also be rewritten as $e^{k+1} = (I - D^{-1}A)e^k$. The matrix $B \equiv I - D^{-1}A$ is often referred to as *iteration matrix*. The Jacobi method provides a sequence of solution approximations with increasing accuracy when the spectral radius of the iteration matrix B is less than one (i.e., $\rho(B) < 1$) [3].

The Jacobi method can also be rewritten in the following component-wise form:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j\neq i} a_{ij}x_j^k\right). \tag{2}$$

2.2. Asynchronous iteration methods

For computing the next iteration in a relaxation method, one usually requires the latest values of all components. For some algorithms, e.g., Gauss–Seidel [26], even the already computed values of the current iteration step are used. This requires a strict order of the component updates, limiting the parallelization potential to a stage, where no component can be updated several times before all the other components are updated.

The question of interest that we want to investigate is, what happens if this order is not adhered. Since in this case, the individual components are updated independently and without consideration of the current state of the other components, the resulting algorithm is called *chaotic or asynchronous iteration method*. Back in the 1970s Chazan and Miranker analyzed some basic properties of these methods and established convergence theory [11]. In the last 30 years, these algorithms were subject of dedicated research activities [8,18,19,37,39,38]. However, they did not play a significant role in high-performance computing, due to the superior convergence properties of synchronized iteration methods. Today, due to the complexity of heterogeneous hardware platforms and the high number of computing units in parallel devices like GPUs, these schemes may become interesting again: they do not require explicit synchronization between the computing cores, probably even located in distinct hardware devices. Since the synchronization usually thwarts the overall performance, it may be true that the asynchronous iteration schemes overcompensate the inferior convergence behavior by superior scalability.

The chaotic-, or asynchronous-relaxation scheme defined by Chazan and Miranker [11] can be characterized by two functions, an update function $u(\cdot)$ and a shift function $s(\cdot, \cdot)$. For each non-negative integer k, the component of the solution approximation x that is updated at step k is given by $u(k)$. For the update at step k, the component m used in this step is $s(k, m)$ steps back. All the other components are kept. This can be expressed as

$$x_i^{k+1} = \begin{cases} \sum_{j=1}^{N} b_{i,j}x_j^{k-s(k,j)} + d_i & \text{if } i = u(k) \\ x_i^k & \text{if } i \neq u(k). \end{cases} \tag{3}$$

Furthermore, the following conditions can be defined to guarantee the well-posedness of the algorithm [35]:

1. The update function $u(\cdot)$ takes each of the values i for $1 \leq i \leq N$ infinitely often.
2. The shift function $s(\cdot, \cdot)$ is bounded by some $\bar{s}$ such that $0 \leq s(k, i) \leq \bar{s}\ \forall\ k \in \{1, 2, \ldots\}, \forall i \in \{1, 2, \ldots, n\}$. For the initial step, we additionally require $s(k, i) \leq k$.

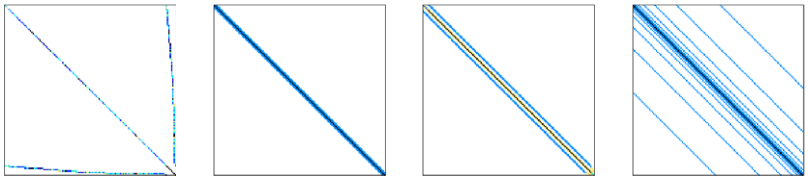
If these conditions are satisfied and $\rho(|B|) < 1$ (i.e., the spectral radius of the iteration matrix, taking the absolute values for its elements, to be smaller than one), the convergence of the asynchronous method is fulfilled [35].

3. Experiment framework

3.1. Linear systems of equations

In our experiments, we search for the approximate solutions of linear system of equations, where the respective matrices are taken from the University of Florida Matrix Collection.[1] Due to the convergence properties of the iterative methods considered the

[1] UFMC; see http://www.cise.ufl.edu/research/sparse/matrices/.



(a) CHEM 97ZTZ. (b) FV1, FV2, FV3. (c) S 1RMT 3M 1. (d) TREFETHEN.

Fig. 1. Sparsity plots of the SPD test matrices.

experiment matrices must be properly chosen. While for the Jacobi method a sufficient condition for convergence is clearly $\rho(B) = \rho(I - D^{-1}A) < 1$ (i.e., the spectral radius of the iteration matrix B to be smaller than one), the convergence theory for asynchronous iteration methods is more involved (and is not the subject of this paper). In [35] John C. Strikwerda has shown that a sufficient condition for the asynchronous iteration to converge for all update and shift functions satisfying conditions (1) and (2) in Section 2.2 is the condition $\rho(|B|) < 1$, where $|B|$ is derived from B by replacing its elements by their corresponding absolute values.

Due to these considerations, we choose symmetric, positive definite (SPD) systems for the experiments. The properties of the test matrices used and their corresponding iteration matrices are summarized in Table 1; structure plots can be found in Fig. 1.

We furthermore take the right-hand sides equal to one for all linear systems.

3.2. Hardware and software issues

The experiments were conducted on a heterogeneous GPU-accelerated multicore system [36] located at the Engineering Mathematics and Computing Lab (EMCL) at the Karlsruhe Institute of Technology, Germany. The system is equipped with two Intel XEON E5540 @ 2.53 GHz and 4 Fermi C2070 (14 Multiprocessors × 32 CUDA cores @1.15 GHz, 6 GB memory [31]). The GPUs are connected to the host through a PCI-e ×16. In the synchronous implementation of Gauss–Seidel on the CPU, 4 cores are used for the matrix–vector operations that can be parallelized. The Intel compiler version 11.1.069 [14] is used with optimization flag "–O3". The GPU implementations of block-asynchronous iteration and Jacobi are based on CUDA [30] in version 4.0.17 [29] using a block-size of 448. The kernels are then launched through different streams. The thread block size and the number of streams, along with other parameters, were determined through empirically based tuning. The CG solver is based on CUBLAS routines provided by NVIDIA.

3.3. An asynchronous iteration method for GPUs

The asynchronous iteration method for GPUs that we propose is split into two levels. This is due to the design of graphics processing units and the CUDA programming language.

The linear system of equations is decomposed into blocks of rows, and the computations for each block is assigned to one thread block on the GPU. Between these thread blocks, an asynchronous iteration method is used, while on each thread block, a Jacobi-like iteration method is performed. We denote this algorithm by *async-(1)*.

Further, we extended this basic algorithm to a version where the threads in a thread block perform multiple Jacobi iterations (e.g., 5) within the block. During the local iterations the x values used from outside the block are kept constant (equal to their values at the beginning of the local iterations). After the local iterations, the updated values are communicated. This approach was also analyzed in [4] and is inspired by the well known hybrid relaxation schemes [5,6], and therefore we denote it as *block-asynchronous* (see Algorithm 1).

Algorithm 1 Block-Asynchronous Iteration.

```
for all (J_l ∈ {J_1 ... J_q}) do {asynchronous outer loop}
  read x from global memory
  s := d_i + Σ_{j∉J_l} b_{i,j} x_j (off-block part)
  for all (i ∈ J_l) do {synchronous Jacobi updates on subdomain}
    for (k = 0; k < iter; k++) do {equal local stopping criterion}
      x_i := s + Σ_{j∈J_l} b_{i,j} x_j^{local}
    end for
  end for
end for
```

In other words, using domain-decomposition terminology, our blocks would correspond to subdomains and thus we additionally iterate locally on every subdomain. We denote this scheme by *async-(local_iters)*, where the index *local_iters* indicates the number of Jacobi updates on the subdomains. A possible motivation for this comes is the hardware. As the subdomains are relatively small and the data needed largely fits into the multiprocessor's cache, the additional iterations almost come for free. The obtained algorithm, visualized in Fig. 2, can be written as component-wise update of the solution approximation:

$$x_i^{(k+1)} += \frac{1}{a_{i,i}} \left(b_i - \underbrace{\sum_{j=1}^{T_S-1} a_{i,j} x_j^{(k-s(k+1,j))}}_{\text{global part}} \right.$$
$$\left. \times \underbrace{- \sum_{j=T_S}^{T_E} a_{i,j} x_j^{(k)}}_{\text{local part}} \underbrace{- \sum_{j=T_E+1}^{n} a_{i,j} x_j^{(m-s(k+1,j))}}_{\text{global part}} \right),$$

where T_S and T_E denote the starting and the ending indices of the matrix/vector part in the thread block. Furthermore, for the local components, the always antecedent values are used, while for the global part, the values from the beginning of the iteration are kept. The shift function $s(k+1, j)$ denotes the iteration shift for the component j—this can be positive or negative, depending on whether the respective other thread block already has conducted more or less iterations. Note that this gives a block Gauss–Seidel flavor to the updates. It should also be mentioned that the shift function may not be the same for components located in different thread blocks.

3.4. Block-asynchronous iteration on multi-GPU systems

Using multiple GPUs, an additional layer of asynchronism is added to the block-asynchronous iteration method. Now we may split the original system into blocks that are addressed to the different devices, whereas the local blocks of rows are again split into

Table 1
Dimension and characteristics of the SPD test matrices and their corresponding iteration matrices.

Matrix name	Description	#n	#nnz	cond(A)	cond($D^{-1}A$)	$\rho(M)$
CHEM 97ZTZ	Statistical problem	2,541	7,361	1.3e+03	7.2e+03	0.7889
FV1	2D/3D problem	9,604	85,264	9.3e+04	12.76	0.8541
FV2	2D/3D problem	9,801	87,025	9.5e+04	12.76	0.8541
FV3	2D/3D problem	9,801	87,025	3.6e+07	4.4e+03	0.9993
S 1RMT 3M 1	Structural problem	5,489	262,411	2.2e+06	7.2e+06	2.65
TREFETHEN _2000	Combinatorial prob.	2,000	41,906	5.1e+04	6.1579	0.8601
TREFETHEN _20 000	Combinatorial prob.	20,000	554,466	5.1e+04	6.1579	0.8601

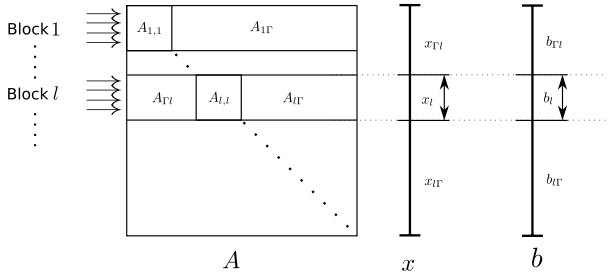


Fig. 2. Visualizing the asynchronous iteration in block description used for the GPU implementation.

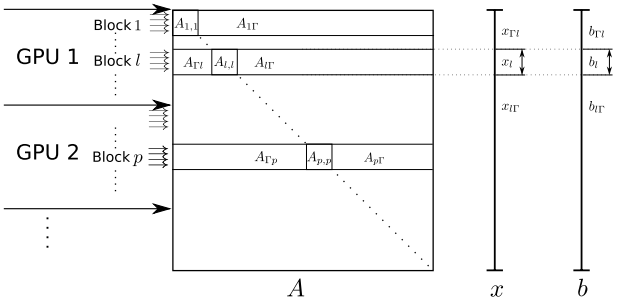


Fig. 3. Visualizing the block-asynchronous iteration for multi-GPU implementation.

even smaller blocks of rows that are then assigned to the different thread blocks on the respective GPUs. In between the GPUs as well as in between the different thread blocks, an asynchronous iteration method is conducted, while on each thread block, multiple Jacobi-like iteration steps can be applied (see Fig. 3). For this reason, one could consider the multi-GPU implementation of the block-asynchronous iteration also as three-stage iteration, but as both outer (block) updates are asynchronous, there is no algorithmic difference to the two-stage iteration. (The inter-GPU and inter-block iterations are considered as one asynchronous block-component update.) For implementing the asynchronous iteration on multi-GPU systems, the central question is how to communicate the updated values efficiently. In the classical approach to multi-GPU implementations, the main memory serves as communication facility, synchronizing the operations and transferring data to the different devices. This usually triggers a high CPU workload and a bottleneck in the algorithm. CUDA 4.0 offers a whole set of possibilities, replacing the former slow communication via the host [29]. While the asynchronous multi-copy technique allows for copying data from the host to several devices and vice versa simultaneously, GPU-direct introduces the ability to transfer data between the memory of distinct GPUs without even involving the CPU. It also features the ability to access GPU memory located in a different device inside a kernel call.

Despite the fact that more implementations can be realized, we want to limit our analysis to the following possibilities.

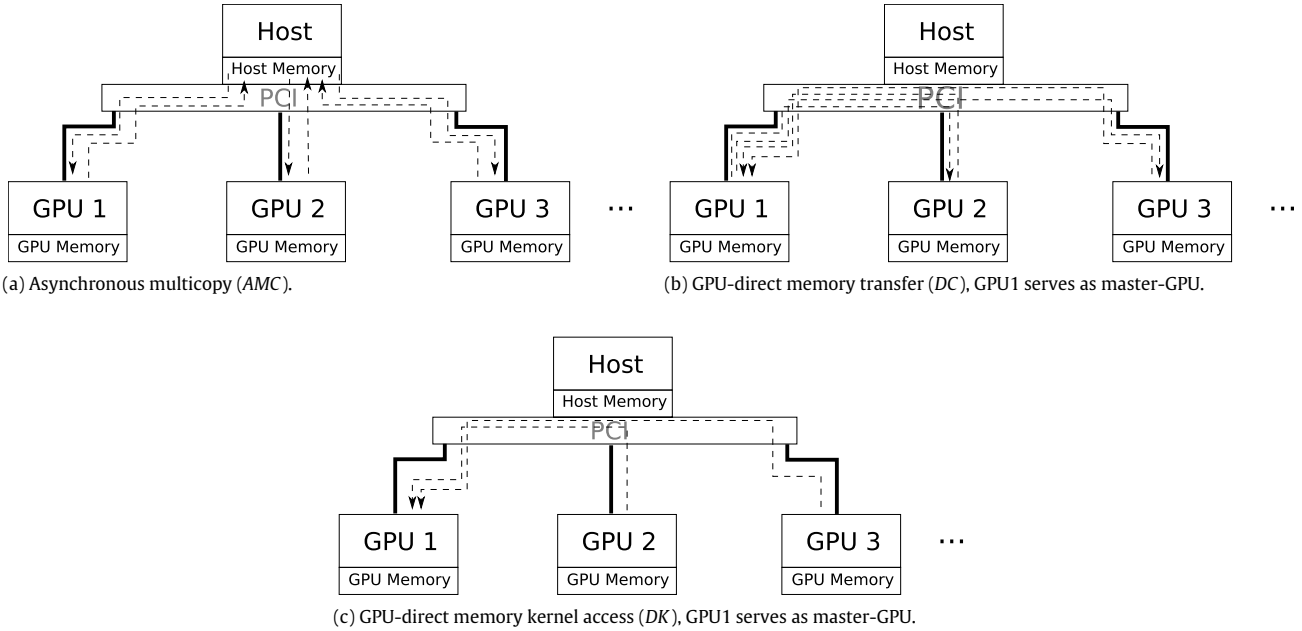
1. *Asynchronous multicopy (AMC).* Like in the classical approach, the host memory serves as communication facility. The performance is improved by allowing for asynchronous data transfers from and to the distinct graphics processing units. The data is put in streams that are then handled independently and without synchronization. This allows the simultaneous data transfer between host and multiple devices. While for computing the next iteration all components of the iteration vector are required, every GPU sends only the new values of the respectively updated components back to the host. We want to stress that this approach is only possible due to the asynchronism in the iteration method; otherwise the performance would suffer from synchronization barriers managing the update order. This approach, illustrated in Fig. 4(a), may especially be interesting for architectures where different GPUs are accessed with different bandwidths, or the distinct GPUs feature different clock rates.
2. *GPU-direct memory transfer (DC).* One feature of CUDA 4.0 is the GPU-direct, allowing the data transfer between different GPUs via PCI without involving the host. If the complete iteration vector is stored on one GPU, say the master-GPU, it can be transferred to the other devices without using the main memory of the host. After every GPU has completed the kernels, the updated components are copied back to the memory of the master-GPU, serving as central storage; see Fig. 4(b). While the connection to the master-GPU now becomes the bottleneck, the CPU of the host system is not needed and can be used for other tasks.
3. *GPU-direct memory kernel access (DK).* The possibility to access data located in another device inside a kernel allows even another implementation, avoiding any explicit data transfer in the code. In this algorithm, the components of the iteration vector are exclusively stored in the memory of the master-GPU, and the kernels handled by the different devices directly access the data in the master-GPU; see Fig. 4(c). Like in the GPU-direct memory transfer implementation, the CPU of the host system is not needed and can be used for other tasks.

4. Numerical experiments

4.1. Stochastic impact of chaotic behavior of asynchronous iteration methods

At this point it should be mentioned that only the synchronous Gauss–Seidel and Jacobi methods are deterministic. While synchronous relaxation algorithms always provide the same solution approximations for each solver run, the unique pattern concerning the distinct component updates in the asynchronous iteration method generates a sequence of iteration approximations that can usually only be reproduced by choosing exactly the same update order. This also implies that variations in the convergence rate may occur for the individual solver runs. If the scheduling for the individual component updates is based on a recurring pattern, these variations in the convergence behavior may increase with the number of iterations since the component update pattern may multiply its influence for higher iteration counts. This may especially become a critical issue when targeting linear systems with considerable off-diagonal parts.

To investigate the issue of the non-deterministic behavior, we conduct multiple solver runs using the same experiment



(a) Asynchronous multicopy (*AMC*).

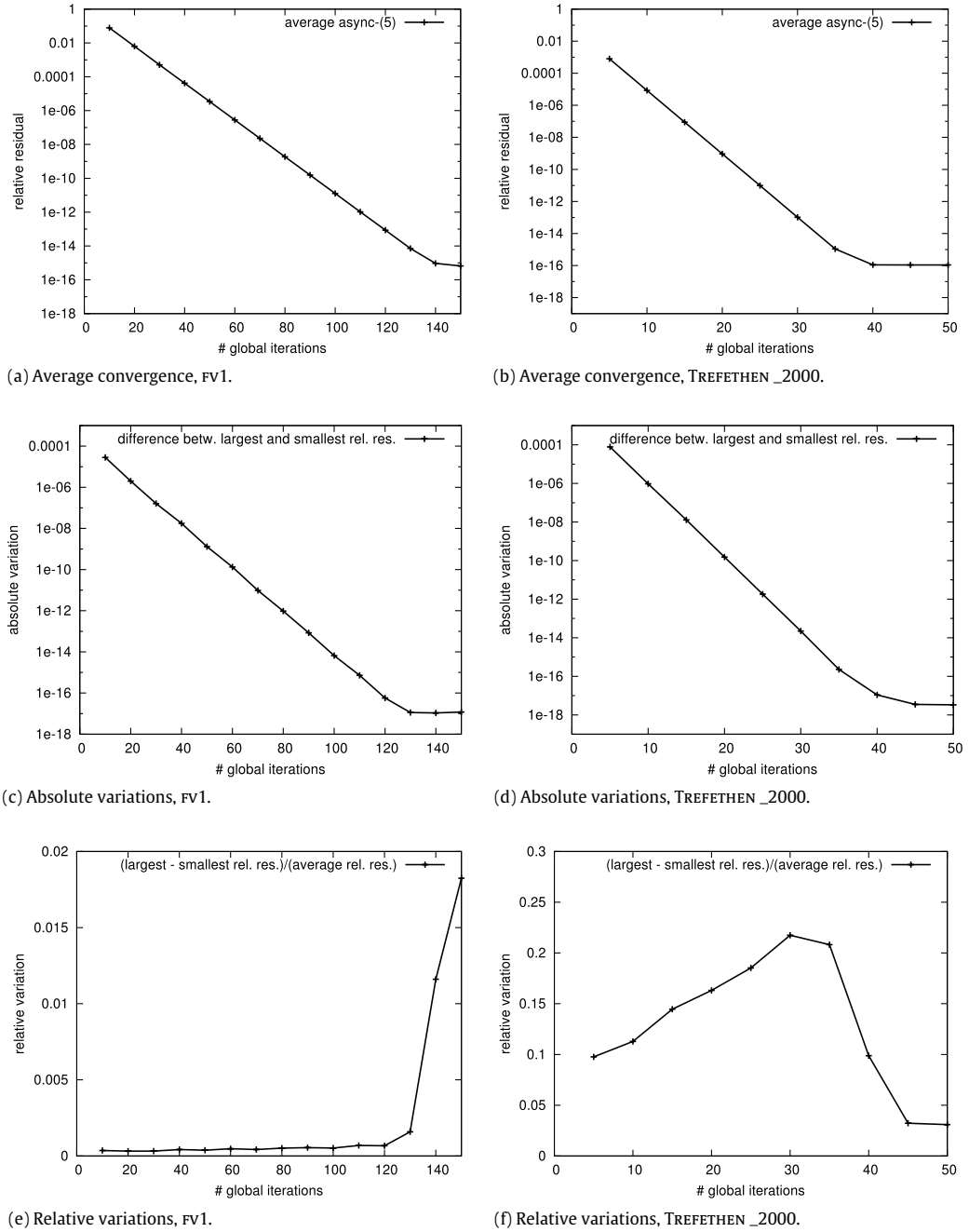
(b) GPU-direct memory transfer (*DC*), GPU1 serves as master-GPU.

(c) GPU-direct memory kernel access (*DK*), GPU1 serves as master-GPU.

Fig. 4. Visualizing the different multi-GPU memory handling. The dashed lines indicate the explicit data transfers.

setup and monitor the relative residual behavior for the different approximation sequences. As the main focus of this chapter is on the block-asynchronous method, we apply the async-(5) algorithm based on a moderate block size of 128, which allows for a strong influence of the non-deterministic GPU-internal scheduling of the threads [30]. Targeting the matrices FV1 and TREFETHEN _2000 ensures that we include results for very different matrix structures, i.e. these systems have very different diagonal dominance (see Section 3.1). All tests on the non-deterministic behavior are based on 1000 solver runs using the same hardware and software configuration on the GPU-accelerated Supermicro system (see Section 3.1 for details about the hardware). For each matrix system, we report in Fig. 5 the variations in convergence as the difference between the largest and smallest relative residual in absolute and relative values (relative to the average residual). Note that the residuals in all reported data of this and the following sections are always in the l^2-norm ($\|r\|_2$). From Fig. 5(a) we can deduce that the async-(5) method using the given experiment setup converges within about 130 global iterations for matrix FV1. Having achieved convergence, also the absolute variations in between the fastest and slowest convergence approach a limit; see Fig. 5(c). While the absolute values provide a general idea about the magnitude of the variations, investigating the relative variances in Fig. 5(e) allows for a more efficient quantification of the differences. Obviously, there exist some variations, but they are very small and may even be neglected. The large relative variations for more than 130 iterations can be explained by rounding effects when computing the relative variation using the very small values (see Table 3) on a system with limited accuracy. The overall small variations are due to the design and sparsity pattern of the matrix: FV1 is symmetric and almost all elements are gathered on the diagonal blocks and therefore are accounted for in the local iterations. Similar tests on diagonal dominant systems with the same sparsity pattern but higher condition number reveal that the latter one has only small impact on the variations between the individual solver runs; see [1]. For the system TREFETHEN _2000 async-(5) converges within about 40 iterations (see Fig. 5(b)). Only very few solver runs have not reached convergence after 45 iterations. Like in the FV1 case, we observe in Fig. 5(d) the expected exponential decrease of the absolute variation. Concerning the relative variations, the results look quite different. All values for the relative variations are significantly higher than for the test matrix FV1. Close to convergence, the relative difference between largest and smallest relative residual approximates 20%. This confirms the expectation that the larger off-block parts in TREFETHEN _2000 emphasize the non-deterministic GPU-internal scheduling since they are not accounted for in the local iteration on the subdomains (see Section 3.3 for the algorithm design). Furthermore, we can identify a dependency between iteration count and the relative variation. The component update orders multiply their impact when conducting a high number of iterations, and the relative difference between the individual approximations rises (see Fig. 5(f)). The linear growth of the relative variations immediately suggests the existence of a recurring pattern in the GPU-internal scheduling, which amplifies the variations in the convergence of the different solver runs.

The exact numbers for these plots can be found in Tables 2 and 3, respectively, where we additionally provide information about the statistical parameters variance, standard error and standard deviation [20]. Summarizing, we can conclude that the scheduling of the threads has influence on the convergence behavior of the block-asynchronous iteration. Especially for systems with significant off-diagonal parts, the effects should be considered when using the method in scientific computing, while for diagonal dominant systems, the variations may be negligible. Furthermore, it seems that the GPU-internal scheduling is based on a recurring pattern which ensures that at some point of the iteration run, all components have been updated similarly often. Due to this pattern, it may be useful to apply larger block-sizes that allow for less possibilities in the scheduling. Larger block-sizes usually come along with the advantage of accounting for more elements in the local iterations, and therewith faster convergence. (Only for very specific matrix properties it may be reasonable to choose small block sizes, i.e. if the off-diagonal parts can be reduced or the matrix size is a multitude of the smaller block size.) The naturally induced upper bound for the block size is the physical core number of the GPU. However, the main consequence of the results is a convention for the rest of this paper. Although we will not stress it explicitly every time, all further results should be considered as average using several solver runs. It should be kept in mind that a different solution update pattern may lead to a slightly faster or slower convergence rate.



(a) Average convergence, FV1.
(b) Average convergence, TREFETHEN _2000.
(c) Absolute variations, FV1.
(d) Absolute variations, TREFETHEN _2000.
(e) Relative variations, FV1.
(f) Relative variations, TREFETHEN _2000.

Fig. 5. Visualizing the average convergence, the absolute respectively relative variations in the convergence behavior of async-(5) depending on the number of conducted global iterations.

4.2. Convergence rate of asynchronous iteration

In the next experiment, we analyze the convergence behavior of the asynchronous iteration method (async-(1)) and compare it with the convergence rate of the Gauss–Seidel and Jacobi method.

The experiment results, summarized in Fig. 6, show that for test systems CHEM 97ZTZ, FV1, FV2, FV3 and TREFETHEN _2000 the synchronous Gauss–Seidel algorithm converges in considerably less iterations than the asynchronous iteration. This superior convergence behavior is intuitively expected, since the synchronization after each component update allows for the usage of the updated components immediately for the next update in the same global iteration. For the Jacobi implementation, the synchronization after each iteration still ensures the usage of all updated components in the next iteration. Since this is not true for the asynchronous iteration, the convergence depends on the problem and the update order. While the usage of updated components implies the potential of a Gauss–Seidel convergence rate, the chaotic properties may trigger convergence slower than Jacobi. Still, we observe for all test cases convergence rates similar to the synchronized Jacobi iteration, which is still low compared to Gauss–Seidel.

The results for test matrix s 1RMT 3M 1 (Fig. 6(e)) show an example where neither of the methods is suitable for direct use. The reason is $\rho(B) > 1$ (in particular, $\rho(B) \approx 2.65$; see Table 1). Nevertheless, note that this matrix is SPD and Jacobi-based methods still can be used after a proper scaling is added, e.g., taking

Table 2
Variations and statistics in the convergence behavior of async-(5) for 1000 solver runs on FV1.

No. of global iters	Averg. res.	Max. res.	Min. res.	Abs. var.	Rel. var.	Variance	Standard deviation	Standard error
10	7.8304e−02	7.8313e−02	7.8285e−02	2.8111e−05	3.5885e−04	1.6334e−11	4.0415e−06	2.0233e−07
20	6.3243e−03	6.3251e−03	6.3231e−03	2.0125e−06	3.1782e−04	1.1244e−13	3.3532e−07	1.6787e−08
30	5.1392e−04	5.1400e−04	5.1383e−04	1.6310e−07	3.1716e−04	7.4425e−16	2.7281e−08	1.3657e−09
40	4.1902e−05	4.1912e−05	4.1895e−05	1.7607e−08	4.2002e−04	6.7713e−18	2.6021e−09	1.3027e−10
50	3.4238e−06	3.4244e−06	3.4231e−06	1.3232e−09	3.7969e−04	4.9401e−20	2.2226e−10	1.1127e−11
60	2.8019e−07	2.8026e−07	2.8012e−07	1.3318e−10	4.7467e−04	4.1092e−22	2.0271e−11	1.0148e−12
70	2.2956e−08	2.2961e−08	2.2951e−08	9.7834e−12	4.2254e−04	2.9781e−24	1.7257e−12	8.6395e−14
80	1.8825e−09	1.8829e−09	1.8820e−09	9.6127e−13	5.0995e−04	2.6892e−26	1.6398e−13	8.2096e−15
90	1.5449e−10	1.5453e−10	1.5445e−10	8.4035e−14	5.4372e−04	1.9368e−28	1.3916e−14	6.9672e−16
100	1.2685e−11	1.2689e−11	1.2682e−11	6.6530e−15	5.2025e−04	1.4737e−30	1.2139e−15	6.0775e−17
110	1.0422e−12	1.0426e−12	1.0418e−12	7.2436e−16	6.9082e−04	1.2855e−32	1.1338e−16	5.6761e−18
120	8.5725e−14	8.5753e−14	8.5695e−14	5.7262e−17	6.6724e−04	8.8288e−35	9.3962e−18	4.7039e−19
130	7.1399e−15	7.1458e−15	7.1345e−15	1.1334e−17	1.5823e−03	4.6302e−36	2.1518e−18	1.0772e−19
140	9.2748e−16	9.3342e−16	9.2265e−16	1.0767e−17	1.1608e−02	3.1601e−36	1.7776e−18	8.8994e−20
150	6.5579e−16	6.6219e−16	6.5022e−16	1.1964e−17	1.8243e−02	3.9031e−36	1.9756e−18	9.8905e−20

Table 3
Variations and statistics in the convergence behavior of async-(5) for 1000 solver runs on TREFETHEN _2000.

No. of global iters	Averg. res.	Max. res	Min. res.	Abs. var.	Rel. var.	Variance	Standard deviation	Standard error
5	8.0190e−04	8.1516e−04	7.3689e−04	7.8277e−05	9.7614e−03	1.6769e−10	1.2949e−05	4.0970e−07
10	8.4330e−06	8.6821e−06	7.7307e−06	9.5147e−07	1.1282e−01	2.8159e−14	1.6780e−07	5.3091e−09
15	8.8600e−08	9.2472e−08	7.9658e−08	1.2813e−08	1.4462e−01	4.8964e−18	2.2127e−09	7.0009e−11
20	9.3022e−10	9.8491e−10	8.3319e−10	1.5171e−10	1.6309e−01	7.0058e−22	2.6468e−11	8.3742e−13
25	9.7817e−12	1.0427e−11	8.6158e−12	1.8120e−12	1.8524e−01	9.4666e−26	3.0767e−13	9.7345e−15
30	1.0260e−13	1.1038e−13	8.8065e−14	2.2314e−14	2.1747e−01	1.2095e−29	3.4778e−15	1.1003e−16
35	1.0906e−15	1.1960e−15	9.6899e−16	2.2705e−16	2.0818e−01	1.5321e−33	3.9142e−17	1.2384e−18
40	1.1012e−16	1.1843e−16	1.0755e−16	1.0881e−17	0.9880e−01	2.4862e−36	1.5767e−18	4.9887e−20
45	1.0811e−16	1.1041e−16	1.0692e−16	3.4880e−18	3.2261e−02	2.2543e−37	4.7479e−19	1.5021e−20
50	1.0811e−16	1.1057e−16	1.0723e−16	3.3381e−18	3.0875e−02	2.2468e−37	4.7400e−19	1.4996e−20

$B = I - \tau D^{-1}A$ with $\tau = \frac{2}{\lambda_1+\lambda_n}$, where λ_1 and λ_n approximate the smallest and largest eigenvalue of $D^{-1}A$.

4.3. Convergence rate of block-asynchronous iteration

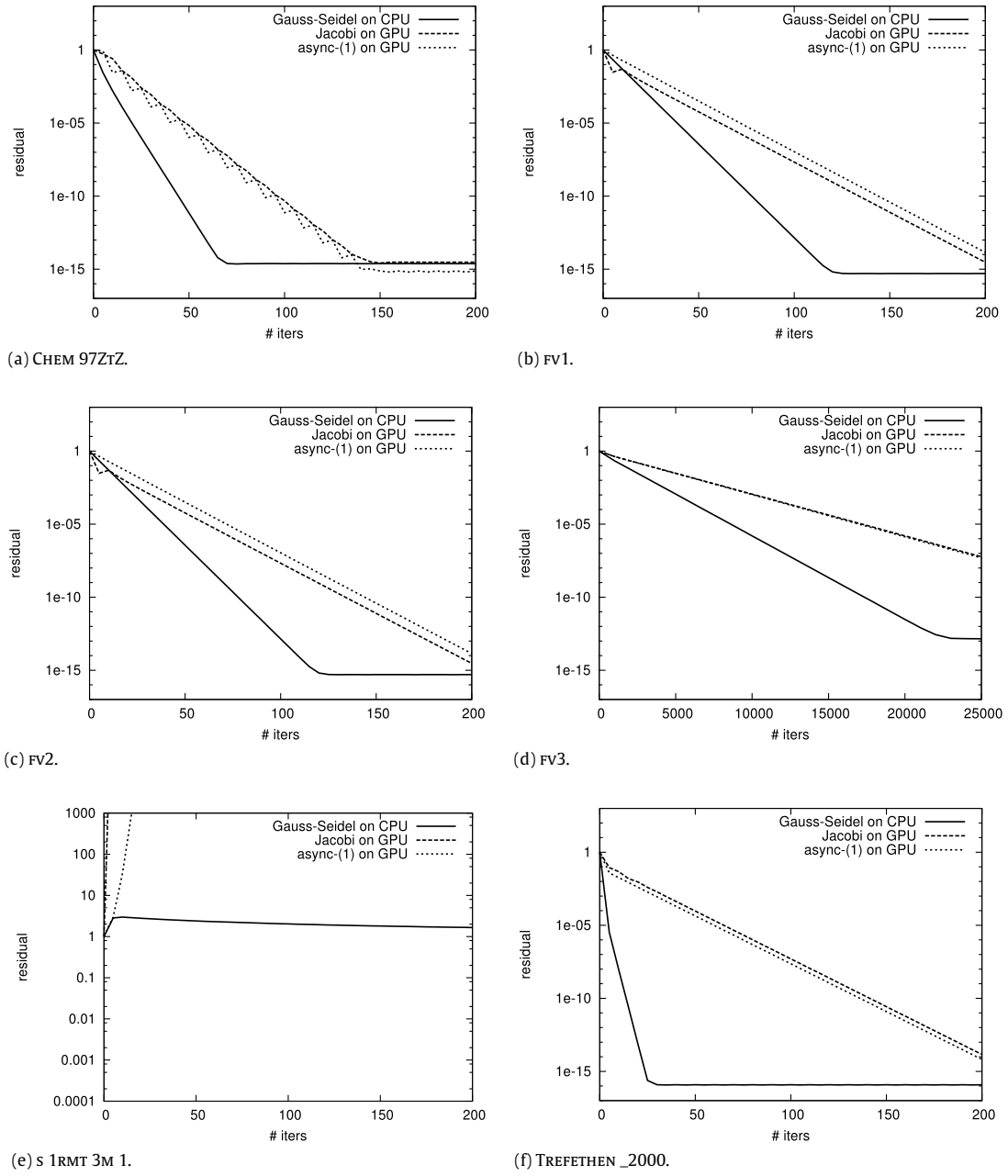
We now consider the block-asynchronous iteration method which additionally performs a few Jacobi-like iterations on every subdomain; see Section 3.3. In Table 4 we report the overhead triggered by the additional local iterations conducted on the subdomains. Switching from async-(1) to async-(2) affects the total computation time by less than 5%, independent of the total number of global iterations. At the same time, the resulting algorithm updates every component twice as often. Even if we iterate every component locally by 9 Jacobi iterations, the overhead is less than 35%, while the total updates for every component is increased by a factor of 9 [1].

There exists a critical point, where adding more local iterations does not improve the overall performance. It is difficult to analyze the trade-off between local and global iterations [28], and we refrain from giving a general statement for the optimal choice of local iterations. This is due to the fact that the choice depends not only on the characteristics of the linear problem, but also on the iteration status of the thread block and the local components (as related to the asynchronism), subdomain sizes, and other parameters. Based on empirical tuning and practical experience (trying to match the convergence of the new method to that of a Gauss–Seidel iteration) we set the number of local Jacobi-like updates to five. Therefore we choose async-(5) for all subsequent analysis of the block-asynchronous iteration method in this chapter. The additional local iterations in asynchronous methods provide less contribution to the iteration process than global ones in synchronized algorithms, as they do not take into account off-block entries. We note that as a consequence, the number of iterations in asynchronous algorithms cannot directly be compared with the number of iterations in a synchronized algorithm. To account for this mismatch and the fact that the local iterations almost come for free in terms of computational effort we from now on use the convention of counting only the number of global iterations, where every single component is updated five times as often by iterating locally. Under this convention we now compare in Fig. 7 the convergence rate of async-(5) with the Gauss–Seidel convergence rate.

Table 4
Overhead to total execution time by adding local iterations, matrix FV3.

Method	Computation time for # global iterations				
	100	200	300	400	500
Async-(1)	1.376425	2.437521	3.501462	4.563519	5.624792
Async-(2)	1.431110	2.546361	3.660030	4.773864	5.891870
Async-(3)	1.482574	2.654470	3.819478	4.987472	6.156434
Async-(4)	1.532940	2.749808	3.972644	5.191812	6.410378
Async-(5)	1.577105	2.838185	4.099068	5.363081	6.655686
Async-(6)	1.629628	2.938897	4.255335	5.569045	6.879329
Async-(7)	1.680975	3.044979	4.412199	5.778823	7.144304
Async-(8)	1.736295	3.148895	4.571684	5.990520	7.409536
Async-(9)	1.786658	3.259132	4.730689	6.202893	7.676786

As theoretically expected, synchronous relaxation as well as the block-asynchronous async-(5) is not directly suitable to use for the S 1RMT 3M 1 matrix. Besides this case, the async-(5) improves the convergence rate of async-(1) for all other matrices. While, depending on the matrix structure, we may expect an improvement factor of up to five, in the experiments we observe improvements of up to four. The rule of thumb expectation for the convergence rate of the async-(5) algorithm is based on the rate with which values are updated and the rate of propagation for the updates. For example, this is the observation that Gauss–Seidel often converges about twice as fast as Jacobi [25]. In other words, four Jacobi iterations would be expected (in general) to provide residual reduction approximating two Gauss–Seidel



(a) CHEM 97ZTZ. (b) FV1.

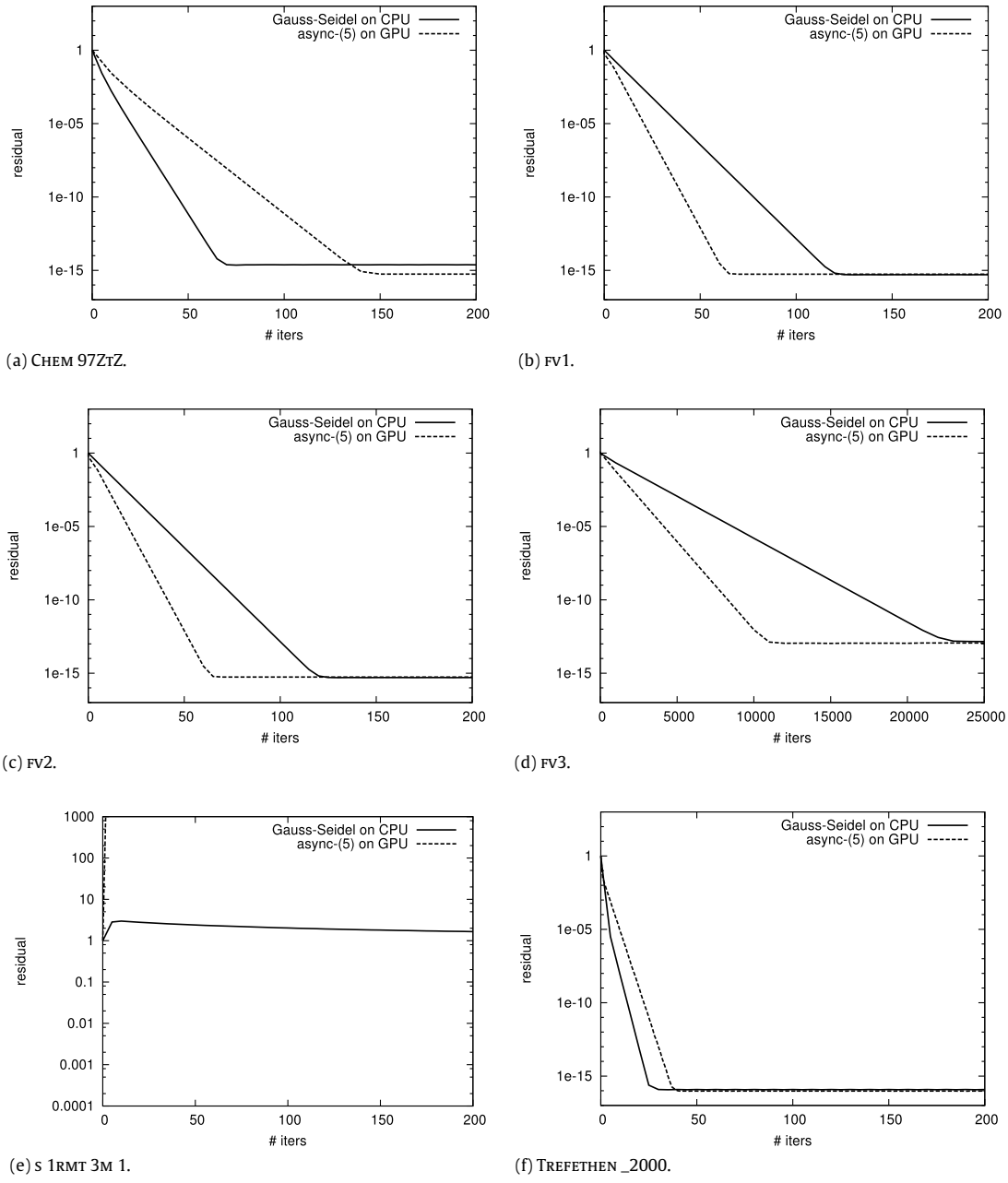
(c) FV2. (d) FV3.

(e) S 1RMT 3M 1. (f) TREFETHEN _2000.

Fig. 6. Convergence behavior for different test matrices.

iterations. The experiments show that the convergence of async-(5) for CHEM 97ZTZ is characteristic for the convergence of the synchronous Jacobi iteration. This can be explained by the fact that the local matrices for CHEM 97ZTZ are diagonal and therefore it does not matter how many local iterations are performed. An improvement for this case could potentially be obtained by reordering. The results for TREFETHEN _2000 are similar; although there is improvement compared to Jacobi, the rate of convergence for async-(5) is not twice as fast as Gauss–Seidel, and the reason is again the structure of the local matrices (see Fig. 1 for the structure of the matrices and Fig. 7(a) and (f) for the convergence results). Considering the remaining linear systems of equations FV1, FV2 and FV3, we obtain approximately twice as fast convergence when replacing Gauss–Seidel by the async-(5) algorithm (see Fig. 7(b)–(d)). Since for these matrices most of the relevant matrix entries are gathered on or near the diagonal and therefore are taken into account in the local iterations on the subdomains, we observe significant convergence gain when iterating locally. Hence, as long as the asynchronous method converges and the off-block entries are "small", adding local iterations may be used to not only compensate for the convergence loss due to the chaotic behavior, but moreover to gain significant overall convergence improvements [1].

But the convergence rate alone does not determine whether an iterative method is efficient or not. The second important metric that we must consider is the time needed to process one iteration on the respective hardware platform. While this time can easily be measured for the synchronous iteration methods, the nature of asynchronous relaxation schemes does not allow the straight forward determination of the time needed per iteration, since not all components are updated at the same time. Especially, it may happen that some components/blocks were already iterated several times, while other blocks were not processed at all. For this reason, only an average time per global iteration can be computed as the ratio of the total time and the total number of iterations. For uniformity we also use an average time for the



(a) Chem 97ZtZ. (b) fv1.

(c) fv2. (d) fv3.

(e) s 1rmt 3m 1. (f) Trefethen _2000.

Fig. 7. Convergence rate of block-asynchronous iteration.

CPU implementation. It should also be mentioned that, while the average timings for one iteration on the CPU are almost constant, for the GPU implementations the iteration time differs considerably. This stems from the fact, that all timings include the data transfers between host and GPU and the kernel launch. Hence, we have considerable communication overhead when performing only a small number of iterations, while the average computation time per iteration decreases significantly for cases where a large number of iterations are conducted. This behavior is shown in Fig. 8, where the average iteration timings for the test matrix fv3 are reported. For the other test matrices, the average timings for the Gauss–Seidel implementation on the CPU and the Jacobi and async-(5) iteration on the GPU are shown in Table 5 where we took the average of the cases when conducting 10, 20, 30, . . . , 200 iterations for the GPU implementations. Note that the iteration time for Jacobi is due to the synchronization after each iteration higher than the time for async-(5), despite the five local updates in the asynchronous method.

Table 5
Average iteration timings in seconds per global iteration.

Matrix name	G.-S. (CPU)	Jacobi (GPU)	Async-(5) (GPU)
Chem 97ZtZ	0.008448	0.002051	0.001742
fv1	0.120191	0.019449	0.012964
fv2	0.125572	0.020997	0.014729
fv3	0.125577	0.021009	0.014737
s 1rmt 3m 1	0.039530	0.006442	0.004967
Trefethen _2000	0.007603	0.001494	0.001305

Overall, we observe that the average iteration time for the async-(5) method using the GPU is only a fraction of the time needed to conduct one iteration of the synchronous Gauss–Seidel on the CPU. While for small iteration numbers and problem sizes we have a factor of around 5, it rises to over 10 for

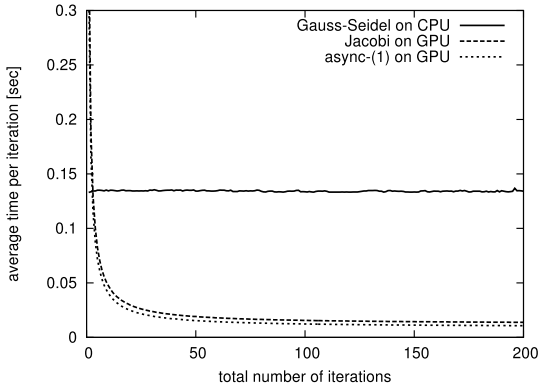


Fig. 8. Average iteration timings of CPU/GPU implementations depending on the total number of iterations, test matrix FV3.

large systems and high total iteration numbers. The question is, whether the faster component updates can compensate for the slower convergence rate when targeting the matrices CHEM 97ZTZ and TREFETHEN _2000. In this case, the block-asynchronous method using the GPU as accelerator would still outperform the synchronous Gauss–Seidel on the CPU.

4.4. Performance of block-asynchronous iteration

In [1] the block asynchronous iteration performance is compared to the performance of Gauss–Seidel and Jacobi. Now, we extend the analysis by comparing also to a highly tuned GPU implementation of the CG solver [33]. Also, instead of reporting only time-to-accuracy, we show the residual decrease over the computing time. This enables us to determine the optimal solver also for cases where low accuracy approximations are sufficient. Such scenarios occur for example in the solution process of nonlinear equations where only coarse solutions for the first linearization steps are required [22]. Due to the results in Fig. 7 it is reasonable to limit the performance analysis to the matrices CHEM 97ZTZ, FV1, FV3 and TREFETHEN _2000. The matrix characteristics and convergence results for FV2 are very similar to FV1, and for the S 1RMT 3M 1 problem we observed that neither Gauss–Seidel nor Jacobi or block-asynchronous iteration are suitable methods. In Fig. 9 the time-dependent residuals for the different solver implementations are reported (relative residuals in l^2-norm).

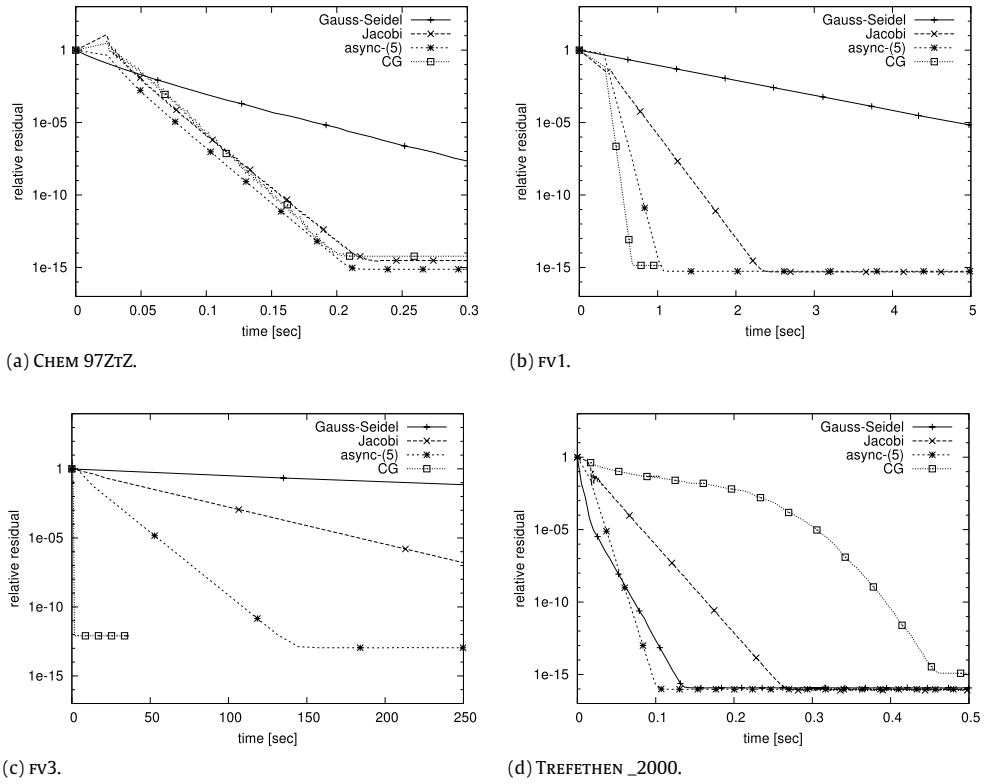
For the very diagonally dominant systems FV1 and FV3 the performance improvement when switching from Jacobi to block-asynchronous iteration is significant. The async-(5) method converges (with respect to computation time) almost twice as fast as Jacobi. At the same time, both methods outperform the sequential (CPU-based) Gauss–Seidel solver by orders of magnitude. Still, the performance of the CG method cannot be achieved. This was expected since the CG method belongs to the most efficient iterative solvers for symmetric positive definite systems. In Fig. 9(b) async-(5) converges about twice as fast as Jacobi, significantly faster than Gauss–Seidel, but the CG method still shows about one-third higher performance. For the system FV3 with considerably higher condition number, the differences are even more significant: the time to solution needed by the synchronized CG is only a fraction of the computation time of Jacobi, async-(5) or Gauss–Seidel (see Fig. 9(c)). Only if the approximation accuracy is relevant, applying async-(5) could be considered to post-iterate the solution approximation. For strongly coupled problems, which results in matrix systems containing large off-diagonal parts, the performance differences between Jacobi and block-asynchronous iteration decrease. This stems from the fact that the entries located outside the subdomains are not taken into account for the local iterations in async-(5) [1]. Thus, it is expected that for the problem CHEM 97ZTZ the performance results for Jacobi and block-asynchronous iteration are very similar. They are also almost equal to the performance of the, algorithmically very different, CG solver. Only the CPU-based Gauss–Seidel converges slower with respect to computation time (see Fig. 9(a)). Concerning the three superior methods, the block-asynchronous iteration outperforms not only the Jacobi method, but even the highly optimized CG solver. Probably the most interesting results occur for the TREFETHEN _2000 problem (see Fig. 9(d)). The async-(5) method using the GPU does not reach the Gauss–Seidel performance on the CPU for small iteration numbers, which may be caused by the characteristics of the problem. As the linear system combines small dimension with a low condition number, both enabling fast convergence, the overhead triggered by the GPU kernel calls is crucial for small iteration numbers [1]. Going to higher iteration numbers, the async-(5) outperforms the CPU implementation of Gauss–Seidel also for this matrix. Compared to CG and Jacobi, the async-(5) method is superior for any approximation accuracy.

4.5. Fault-tolerance of block-asynchronous iteration

This section is dedicated to the analysis of the block-asynchronous iteration introduced in Section 3.3 with respect to error resilience. Particularly, we want to analyze how hardware failure impacts the method's convergence and performance characteristics. The topic of fault resilient algorithms is of significant importance, as we expect that the high complexity in future hardware systems comes along with a high failure rate [10,32].

Current high performance systems consist of about one million processing elements, but aiming for exascale computing, this number must be increased by about three orders of magnitude [2,7]. This increase in the component number is expected to be faster than the increase in component reliability, with projections in the minutes or seconds for exascale systems. From the current knowledge and observations of existing large systems, it is anticipated that especially a rise of the *silent errors* may take place. These are errors that get detected after long time having caused serious damage to the algorithm, or never get detected at all [2].

For most synchronized iterative solvers hardware failure is crucial, resulting in the breakdown of the algorithm. For implementations that do not feature checkpointing or recovery techniques [9,16,17,21,27], the complete solution process must be restarted. While checkpointing strategies are widely used in today's implementations, algorithms will no longer be able to rely on checkpointing to cope with faults in the Exascale era. This stems from the fact that the time for checkpointing and restarting will exceed the mean time of failure of the full system [2]. Hence, a new fault will occur before the application could be restarted, causing the application to get stuck in a state of constantly being restarted. The nature of asynchronous methods removes the need for checkpointing: the high tolerance to update order and communication delay implies that, as long as all components are updated at some point, they are resilient to hardware failure. The inherent reliability with respect to detectable hardware failure makes asynchronous methods suitable candidates for exascale systems based on a high number of processing elements. In case of silent errors that do not get detected, also the asynchronous methods will usually not converge to the solution. This implies that for problems where convergence is expected, a convergence delay or non-converging sequence of solution approximations indicates that a silent error has occurred. Especially a delay in the expected convergence may portend that a temporal hardware failure has taken place, e.g., the power-off of one component due to overheating. Hence, additionally to the fault-tolerance with respect to detectable hardware errors, it is also possible to imagine scenarios where asynchronous methods can be used to detect silent errors.



(a) Chem 97ZtZ. (b) fv1.

(c) fv3. (d) Trefethen _2000.

Fig. 9. Relative residual behavior with respect to solver runtime.

To investigate the issue of fault tolerance experimentally, we consider the following scenario. Block-asynchronous iteration is used on a system with a high core number to solve a linear system of equations. At some point, a certain number of the cores iterating the distinct components break down. Within a certain time frame, the operating system detects the hardware failure and may reconfigure the algorithm during runtime by assigning the respective components to other (e.g., additional) cores or fix the corrupted ones. This setup is realistic as it may occur due to several reasons. One example is the failure of computing units of the different layers (cores, CPUs, nodes), a common *hardware error* scenario [10]. Operating system software may detect hardware breakdown after some time, and it is a question of the implementation how these hard errors are handled.

In our experiments we simulate the introduced scenario the following way. At a certain time t_0, a preset number of randomly chosen components is no longer considered in the iteration process. We report the residual behavior of different implementations that either detect the failure and reassign the components to other cores after a certain recovery time t_r, or do not recover at all. While we expect a delay in the convergence for the recovery case, the algorithms not reassigning the components handled by broken cores may generate a solution approximation with a significant residual error. We consider the test matrices fv1 and Trefethen _2000 that are suitable candidates due to their very different characteristics (see Section 3.1). For all experiments, we simulate the hardware breakdown of 25% of the computing cores after about $t_0 = 10$ global iterations. This implies that one fourth of the components (randomly chosen) is no longer updated. We implement different versions, where the algorithm detects and reassigns the components after $t_r = 10, 20, 30$ global iterations or does not recover at all. In Fig. 10 we report the relative residual behavior for the different implementations and observe very similar results for both matrix systems. In the non-recovering case, the algorithm generates a solution approximation that differs significantly from the exact solution. Especially, continuing the iteration process for the remaining components (handled by the working cores) has no influence on the generated values for the symmetric positive definite matrices. As soon as the iteration process recovers by assigning the workload of the broken cores to other hardware components, the convergence is retrieved. With some problem specific delay (see Table 6), the same solution approximation is generated like for the case of no error. This reveals the high resilience of block-asynchronous iteration with respect to hardware failure and the high reliability of the methods.

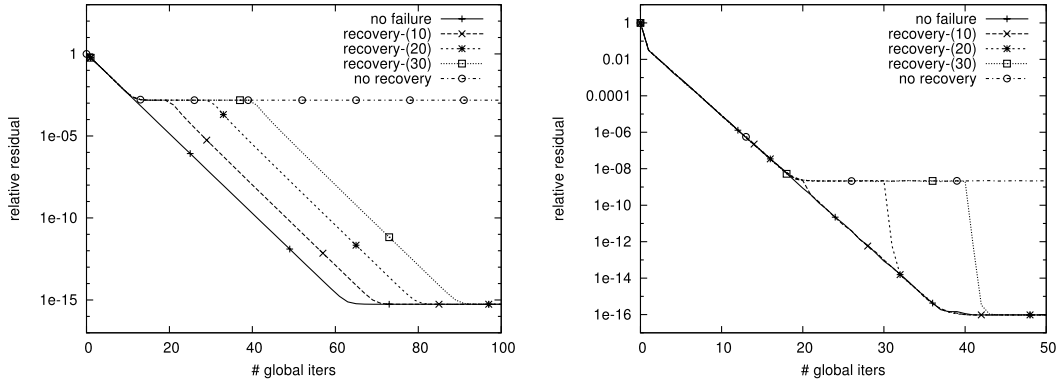
Table 6
Additional computation time in % needed for the async-(5) featuring different recovery times t_r to provide the solution approximation.

Recovering async-(5)	Recover-(10)	Recover-(20)	Recover-(30)
fv1	8.16	19.50	31.66
Trefethen _2000	8.16	11.45	16.61

4.6. Experiments on block-asynchronous iteration on multi-GPU systems

Finally, we want to include numerical experiments on the different approaches to block-asynchronous iteration using multi-GPU systems (see Section 3.4). Prior to that, we provide in Table 7 the bandwidth rates we obtained for the test system.

Note that the host\host bandwidth describes the inter-CPU bandwidth of the Dual-IOH machine using QPI. The GPU\GPU bandwidth is either the GPU-direct bandwidth (if both GPUs are connected to the same CPU), or using the main memory as transfer storage. Analyzing the data, we observe, that the GPU-direct is not able to achieve the same memory bandwidth like the GPU–host communication. At this point we want to mention that the results agree to investigations conducted by Saeed Iqbal and Shawn Gao [24,23]. Furthermore, we may conclude that GPU0 and



(a) FV1. (b) TREFETHEN _2000.

Fig. 10. Convergence of async-(5) for hardware failure. The implementations either recover by reassigning the components to other cores after the recovery time t_r (denoted by recover-(t_r)), or generate a solution approximation with a significant residual error.

Table 7
Inter-device memory bandwidth (GB/s).

From	To				
	Host	GPU0	GPU1	GPU2	GPU3
Host	16.0	5.94	5.93	5.91	5.91
GPU0	6.42	–	4.91	3.18	3.18
GPU1	6.41	4.91	–	3.18	3.19
GPU2	6.39	3.18	3.20	–	4.91
GPU3	6.39	3.19	3.20	4.91	–

GPU1, respectively GPU2 and GPU3 are connected to the same CPU controller.

We now apply the different multi-GPU block asynchronous implementations to the test matrix TREFETHEN_20 000 (see Section 3.1), which is suitable for the experiment due to its size and structure. Since we can assume that for all implementations, the solution approximation accuracy almost linearly depends on the run-time, for comparing the performance of the different implementations it is sufficient to report the time-to-convergence. In Fig. 11 we subtracted the respective initialization overhead. For the case of using only one GPU, the DC and DK approaches are slightly faster than the asynchronous multicopy since the iteration vector resides in the GPU memory and is not transferred back and forth between CPU and GPU. Other than that, the algorithms of asynchronous multicopy, GPU-direct memory transfer, and GPU-direct memory kernel access do not differ when running on only one GPU. As soon as we include a second GPU, the asynchronous multicopy performs considerably faster. The total run-time is almost cut in half, while for the GPU-direct based implementations, only small improvements can be observed. The reason for the excellent speedup of the AMC stems from the fact that each GPU only handles half the components, and due to the asynchronous data transfer using different PCI controller and different PCI connections, the two devices can add their performance. Leveraging the complete bandwidth of both PCI connections is essential since the application is memory bound. Using the memory located in one of the GPUs, like we do in the GPU-direct approaches, puts a lot of pressure on the PCI connection of this GPU: all data must be transferred via this connection. Unfortunately, for the specific architecture of the Supermicro system we target in the experiments the CUDA in version 4.0.17 does not yet support GPU-direct between more than two GPUs as CUDA's GPU–GPU communication is only supported for GPUs connected to the same CPU [29]. But also for the AMC implementation, adding more GPUs is not beneficial for this test case. Using three GPUs we iterate still faster than when using only one GPU, we are about 20% slower than for the dual-GPU case. This stems from the fact that using more than two GPUs, the architecture implies data transfers via the QPI connection between

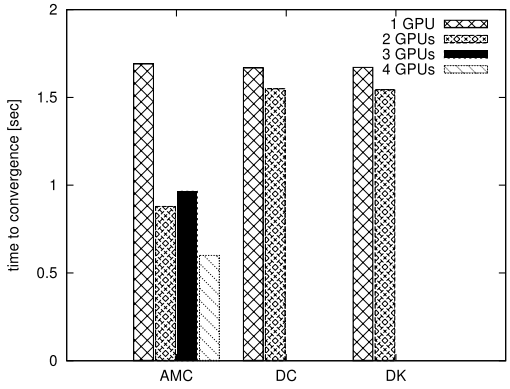


Fig. 11. Time-to-solution for the different multi-GPU implementations for test matrix TREFETHEN _20 000.

the two CPUs thwarting the performance [36]. Increasing the GPU number from three to four, we again benefit from the reduced component number handled per device as the configuration includes the communication via the QPI anyway. Although this configuration outperforms the dual-GPU performance, the speedup is, due to the additional communication overhead, considerably smaller than the factor of two.

5. Summary and future work

Synchronizations are becoming increasingly hazardous for high performance computing, especially with the ever-increasing parallelism in today's computer architectures. In effect, the use of new algorithms and optimization techniques that reduce synchronizations are becoming essential for obtaining high efficiency, high performance, and scalability. We demonstrated and quantified this for relaxation methods on current GPUs (and multicore clusters with GPUs). In particular, we developed asynchronous relaxation methods and showed that the absence of synchronization points enables not only excellent scalability, but also a high tolerance to hardware failure. Further, the efficient hardware usage enabled by the asynchronicity overcompensated the inferior convergence properties of these algorithms, e.g., when compared to the Gauss–Seidel relaxation, to still provide solution approximations of prescribed accuracy in considerably shorter time. Targeting multi-GPU platforms, we have shown that the derived algorithms allow for the efficient usage of asynchronous data transfers, while for memory bound problems the performance suffers significantly from the low throughput rates of the inter-device connection.

Finally, the results bolster an overall claim that asynchronous methods, and in general techniques that trade extra flops (that can come for free on new hardware) for reduced synchronization, would be becoming increasingly relevant in future architectures, as parallelism, and therefore the overhead of synchronization is compelled to escalate.

Along with the benefits of the asynchronous methods discussed, there are currently some limitations, overcoming which opens up some further research directions. In particular, as the numerical properties of asynchronous iteration pose some restrictions on the usage, future research should investigate whether block-asynchronous iteration is suitable for a specific partial differential equation discretization by analyzing the problem and method characteristics. Another research field is related to the widespread use of the component-wise relaxation method as preconditioner or smoother in the multigrid. However it is still a subject of further research how to determine the optimal number for the various parameters arising in the asynchronous methods, such as the number of local iterations, subdomain sizes, scaling parameters, etc., with respect to the problem. This optimization may not only be dependent on the problem, but also on the parameters in the multigrid framework like prolongation operator and number of pre- and post-smoothing steps, and the used hardware system. Another limitation is the lack of bit-wise reproducibility, a desirable requirement toward numerical libraries. Finally, asynchronous iterations when used as preconditioners would not preserve the symmetry of a possibly symmetric operator.

Acknowledgments

The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and the MathWorks for supporting this research effort. We would also like to thank the Karlsruhe House of Young Scientists (KHYS) for supporting the research cooperation.

References

[1] H. Anzt, S. Tomov, J. Dongarra, V. Heuveline, A block-asynchronous relaxation method for graphics processing units, in: IPDPS Workshops, 2012.
[2] S. Ashby, et al., The opportunities and challenges of exascale computing, in: Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, 2010.
[3] R. Bagnara, A unified proof for the convergence of Jacobi and Gauss–Seidel methods, SIAM Review 37 (1995) 93–97.
[4] Z.-Z. Bai, V. Migallón, J. Penadés, D.B. Szyld, Block and asynchronous two-stage methods for mildly nonlinear systems, Numerische Mathematik 82 (1999) 1–20.
[5] A.H. Baker, R.D. Falgout, T. Gamblin, T.V. Kolev, S. Martin, U. Meier Yang, Scaling algebraic multigrid solvers: on the road to exascale, in: Proceedings of Competence in High Performance Computing, CiHPC.
[6] A.H. Baker, R.D. Falgout, T.V. Kolev, U. Meier Yang, Multigrid smoothers for ultra-parallel computing, lLNL-JRNL-435315, 2011.
[7] K. Bergman, et al. Exascale computing study: technology challenges in achieving exascale systems, 2008.
[8] D.P. Bertsekas, J. Eckstein, Distributed asynchronous relaxation methods for linear network flow problems, in: Proceedings of IFAC'87.
[9] P.G. Bridges, M. Hoemmen, K.B. Ferreira, M.A. Heroux, P. Soltero, R. Brightwell, Cooperative application/OS DRAM fault recovery, in: Proceedings of the 2011 International Conference on Parallel Processing—Volume 2, Euro-Par'11, Springer-Verlag, Berlin, Heidelberg, 2012.
[10] F. Cappello, A. Geist, B. Gropp, L. Kale, B. Kramer, M. Snir, Toward exascale resilience, International Journal of High Performance Computing Applications 23 (4) (2009) 374–388.
[11] D. Chazan, W. Miranker, Chaotic relaxation, Linear Algebra and its Applications 2 (7) (1969) 199–222.
[12] A.T. Chronopoulos, A.B. Kucherov, A parallel Krylov-type method for nonsymmetric linear systems, 2001, pp. 104–114.
[13] J.A.H. Courtecuisse, Parallel dense Gauss–Seidel algorithm on many-core processors.
[14] I. Corporation, Intel C++ compiler options, document Number: 307776-002US.
[15] J. Dongarra, et al., The international ExaScale software project roadmap, International Journal of High Performance Computing Applications 25 (1) (2011).
[16] P. Du, A. Bouteiller, G. Bosilca, T. Herault, J. Dongarra, Algorithm-based fault tolerance for dense matrix factorizations, Tech. Rep., Innovative Computing Laboratory, University of Tennessee, 2011.
[17] P. Du, P. Luszczek, S. Tomov, J. Dongarra, Soft error resilient QR factorization for hybrid system with GPGPU, Journal of Computational Science (2013).
[18] A. Frommer, H. Schwandt, D.B. Szyld, Asynchronous weighted additive Schwarz methods, Electronic Transactions on Numerical Analysis 5 (1997) 48–61.
[19] A. Frommer, D.B. Szyld, On asynchronous iterations, Journal of Computational and Applied Mathematics 123 (2000) 201–216.
[20] N. Henze, Stochastik für Einsteiger: Eine Einführung in die faszinierende Welt des Zufalls; Mit über 220 Übungsaufgaben und Lösungen, Vieweg+teubner Verlag, 2010.
[21] M.A. Heroux, M. Hoemmen, Fault-tolerant iterative methods via selective reliability, Tech. Rep., Sandia National Laboratories, 2011.
[22] J. Hubbard, B. Hubbard, Vector Calculus, Linear Algebra, and Differential Forms: A Unified Approach, Matrix Editions, 2007.
[23] S. Iqbal, S. Gao, GPU direct improves communication bandwidth between GPUs on the C410X, February 2012.
[24] S. Iqbal, S. Gao, T. Mckercher, Comparing GPU-direct enabled communication patterns for oil and gas simulations, February 2012.
[25] G. Karniadakis, R. Kirby, Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation, Cambridge University Press, 2003.
[26] C.T. Kelley, Iterative Methods for Linear and Nonlinear Equations, SIAM, 1995.
[27] K. Malkowski, P. Raghavan, M. Kandemir, Analyzing the soft error resilience of linear solvers on multicore multiprocessors, 2010, pp. 1–12.
[28] U. Meier Yang, On the use of relaxation parameters in hybrid smoothers, Numerical Linear Algebra with Applications 11 (2011) 155–172.
[29] NVIDIA Corporation, CUDA Toolkit 4.0 Readiness for CUDA Applications, fourth ed., March 2011.
[30] NVIDIA Corporation, NVIDIA CUDA Compute Unified Device Architecture Programming Guide, second ed., August 2009.
[31] NVIDIA Corporation, TESLA C2050/C2070GPU Computing Processor, July 2010.
[32] M.D. Powell, A. Biswas, S. Gupta, S.S. Mukherjee, Architectural core salvaging in a multi-core processor for hard-error tolerance, in: Proceedings of the 36th Annual International Symposium on Computer Architecture, ISCA'09, ACM, New York, NY, USA, 2009.
[33] Y. Saad, Iterative Methods for Sparse Linear Systems, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
[34] Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual method for solving nonsymmetric linear systems.
[35] J.C. Strikwerda, A convergence theorem for chaotic asynchronous relaxation, Linear Algebra and its Applications 253 (1–3) (1997) 15–24.
[36] Super Micro Computer, Inc., Supermicro X8DTG-QF User's Manual, revision 1.0a ed., 2010.
[37] D.B. Szyld, The mystery of asynchronous iterations convergence when the spectral radius is one, Tech. Rep. 98-102, Department of Mathematics, Temple University, Philadelphia, Pa., October 1998. Available at: http://www.math.temple.edu/szyld.
[38] A. Üresin, M. Dubois, Generalized asynchronous iterations, in: CONPAR, 1986.
[39] A. Üeresin, M. Dubois, Sufficient conditions for the convergence of asynchronous iterations, Parallel Computing 10 (1) (1989) 83–92.

Hartwig Anzt received his Ph.D. in mathematics from the Karlsruhe Institute of Technology (KIT) in 2012 and holds a Diploma in a joint program including mathematics, physics, and computer science from the University of Karlsruhe. Anzt's main topics of research are in scientific high performance computing and include simulation algorithms, hardware-optimized numerics for GPU-accelerated platforms, communication-avoiding and asynchronous methods, and power-aware computing. In June 2013, Anzt will join Jack Dongarra's Innovative Computing Lab (ICL) at the University of Tennessee, where he had already spent several months during his Ph.D. research collaborating on a number of problems in HPC. Furthermore, he enjoys a fruitful research cooperation with the High Performance Computing & Architectures group at University of Jaume. During his masters Anzt also spent one year at the University of Ottawa, Canada.

Stanimire Tomov is a Research Director in the Innovative Computing Laboratory (ICL) at the University of Tennessee. Tomov's research interests include parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). He has been involved in the development of numerical algorithms and software tools in a variety of fields ranging from scientific visualization and data mining to accurate and efficient numerical solution of PDEs. Currently, his work is concentrated on the development of numerical linear algebra libraries for emerging architectures for HPC, such as heterogeneous multicore processors, graphics processing units (GPUs), and Many Integrated Core (MIC) architectures.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004 for his contributions in the application of high performance computers using innovative approaches; in 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; and in 2011 he was the recipient of the IEEE IPDPS 2011 Charles Babbage Award. He is a fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.

Vincent Heuveline is the Director of the Engineering Mathematics and Computing Lab (EMCL). He is full professor at the Karlsruhe Institute of Technology (KIT) where he is leading the Chair on Numerical Simulation, Optimization and High Performance Computing (NumHPC). His research interests include numerical simulation, high performance computing, software engineering with main application focuses on Energy Research, Meteorology and environment as well as Medical Engineering. He serves as a programme committee member of several international conferences on high performance computing. He is widely consulted for industry with respect to the deployment of numerical simulation and high performance computing in industrial environments.