Energy-aware mixed precision iterative refinement for linear systems on GPU-accelerated multi-node HPC clusters

Martin Wlotzka[1], Vincent Heuveline[1]

[1] Heidelberg University, Interdisciplinary Center for Scientific Computing (IWR)
Engineering Mathematics and Computing Lab (EMCL), Speyerer Str. 6, 69115 Heidelberg
martin.wlotzka@uni-heidelberg.de, vincent.heuveline@uni-heidelberg.de

Abstract: Modern high-performance computing systems are often built as a cluster of interconnected compute nodes, where each node is built upon a hybrid hardware stack of multi-core processors and many-core accelerators. To efficiently use such systems, numerical methods must embrace the different levels of parallelism from the coarse-grained distributed memory cluster level to the fine-grained shared memory node level parallelism. Synchronization requirements of numerical methods may diminish parallel performance and result in increased energy consumption. We investigate block-asynchronous iteration methods in combination with mixed precision iterative refinement to address this issue. We depict our implementation for multi-node distributed systems using MPI with a hybrid node level parallelization for multi-core CPUs using OpenMP and multiple CUDA-capable accelerators. Our numerical experiments are based on a linear system arising from the finite element discretization of the Poisson equation. We present energy and runtime measurements for a quad-CPU and dual-GPU test system. We achieve runtime and energy savings of up to 70% for block-asynchronous GPU-accelerated iteration using mixed precision compared to CPU-only computation. We also encounter configurations where the CPU-only computation is advantageous over the GPU-accelerated method.

Keywords: energy-aware numerics, high-performance computing, mixed precision, asynchronous iteration, graphics processing units

1 Introduction

Numerical simulations play a key role for scientific discovery, complementing theoretical analyses and experiments. The computational power of high-performance computing (HPC) systems in terms of peak floating point operations per second (flops) has increased currently above the petaflops level [ww15]. Seeking to further increase the computational power towards the exascale level, the HPC community is facing the power wall. Simply upscaling current technology would result in a prohibitive power demand in the order of several hundred Megawatts for one exascale system. The issue of energy consumption has therefore become a major issue in the HPC field [ww14].

Many HPC systems are built as a cluster of interconnected compute nodes. Each node usually comprises one or more multi-core processors, and may additionally include many-core accelerators. Thus, HPC clusters often represent a hybrid form of distributed memory interconnected nodes with shared memory multi-core CPUs and possibly many-core devices on the node level. Such systems offer different levels of parallelism. In order to

leverage the computational power, numerical methods must exploit the parallelism provided on the different levels. However, synchronization requirements may diminish the parallel performance and result in increased energy consumption. Asynchronous iteration methods allow to circumvent typical synchronization requirements of classical iteration methods. To address the issues of synchronization and energy consumption, we investigate the block-asynchronous variant in combination with mixed precision iterative refinement for the solution of linear systems of equations.

Related work and paper contribution

The idea of ”chaotic relaxation” was proposed by Rosenfeld [Ro69], who used ”parallel processor computing systems” to simulate the distribution of current in an electrical network. Chazan and Miranker in 1969 [CM69] were the first to study this type of methods on a rigorous theoretical basis. They established a characterization of the chaotic relaxation schemes for the solution of symmetric positive definite linear problems and gave conditions for convergence, as well as examples for divergence. Meanwhile, the denomination ”asynchronous iteration” has been established in the literature. An overview of asynchronous schemes and convergence theory can be found in [FS00]. Asynchronous iteration has successfully been used in the context of HPC, see e.g. [EFS05] and references therein.

Earlier works reported in [An11b] and [An13] investigate convergence properties and performance of block-asynchronous iteration on GPU-accelerated systems, both as plain solver and in combination with mixed precision iterative refinement. However, these works are restricted to single node, single host process configurations, and the host CPUs are not taken into account for computations. We extend this setup to the case of distributed memory machines with several host processes running on the same node and sharing devices. Additionally, we compare with asynchronous CPU-only methods which also benefit from relaxed synchronization requirements. Finally, we perform actual energy measurements to investigate the energy consumption of the methods.

Paper organization

This paper is structured in the following way: We outline the mathematical background in Section 2. In Section 3, we describe the setup of our experiments. In particular, we present the main features of our hybrid implementation using MPI [Me12] for distributed systems and OpenMP [Op13] on the shared memory local level, as well as CUDA [NV14a] for graphics processing units (GPU). Section 4 is devoted to the discussion of the results, and Section 5 concludes the work.

2 Mathematical background

In this work, we investigate the performance and energy consumption of asynchronous iteration schemes in the context of mixed precision iterative refinement. In this section, we introduce the mathematical background of the methods we use.

2.1 Mixed precision iterative refinement

The idea of iterative refinement for the solution of a system of linear equations comes from Newton's method for approximating the solution of $f(x) = 0$, where f is a smooth function. Considering the special case of a linear function $f(x) = b - Ax$ with a regular matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, solving $f(x) = 0$ is equivalent to solving the linear system $Ax = b$. For an approximate solution x^k, the residual is denoted $r^k = b - Ax^k = f(x^k)$. Using $\nabla f \equiv -A$ leads to the following linear iterative refinement method

$$x^{k+1} = x^k + A^{-1}r^k \qquad (k = 0,1,...) .$$

In each iteration, the current approximation x^k is improved by the correction $c^k = A^{-1}r^k$, which is the solution of the error correction equation $Ac^k = r^k$. If an exact correction could be computed, the iterative refinement process would end after one iteration with the correct result. However in practice, often only approximate error corrections $\tilde{c}^k$ can be computed by means of numerical solvers. Let $q^k = r^k - A\tilde{c}^k$ be the residual of the error correction equation. After the correction, the updated solution $x^{k+1} = x^k + \tilde{c}^k$ yields the residual

$$r^{k+1} = b - Ax^{k+1} = b - A(x^k + \tilde{c}^k) = r^k - A\tilde{c}^k = q^k .$$

Thus, the accuracy of the error correction solver determines the accuracy of the solution of the overall iterative refinement process.
Instead of solving the error correction equation in the working precision, one can transfer the system to a lower precision. This approach amounts to the mixed precision iterative refinement (MPIR) [Ba09], see Algorithm 1. For MPIR, we use an absolute stopping criterion based on a given tolerance $\varepsilon > 0$.
For computing the correction in step 6 of Algorithm 1, one may choose any appropriate numerical solver. In this work, we focus on the asynchronous methods explained in the next section.

Algorithm 1 Mixed precision iterative refinement (MPIR)

1: Typecast $A^{\text{low}} \leftarrow A^{\text{high}}$.
2: Set initial solution x^{high}, tolerance $\varepsilon > 0$.
3: Compute residual $r^{\text{high}} = b^{\text{high}} - A^{\text{high}}x^{\text{high}}$.
4: **while** $\|r^{\text{high}}\| > \varepsilon$ **do**
5: Typecast $r^{\text{low}} \leftarrow r^{\text{high}}$.
6: Solve $A^{\text{low}}c^{\text{low}} = r^{\text{low}}$ approximately.
7: Typecast $c^{\text{high}} \leftarrow c^{\text{low}}$.
8: Correct $x^{\text{high}} \leftarrow x^{\text{high}} + c^{\text{high}}$.
9: Compute residual $r^{\text{high}} = b^{\text{high}} - A^{\text{high}}x^{\text{high}}$.
10: **end while**

2.2 Block-asynchronous iteration

The asynchronous iteration methods under investigation in this work can be derived from the classical Jacobi relaxation method [An13]. The Jacobi method relies on an additive

splitting of the system matrix $A = L + D + U$ into a lower triangular matrix L, a diagonal matrix D and an upper triangular matrix U. Assuming D to be regular, the Jacobi iteration reads [Me11]

$$\begin{aligned} x^{k+1} &= D^{-1}\left[b - (L+U)x^k\right] \\ &= Bx^k + d \end{aligned} \qquad (k = 0, 1, \ldots) ,$$

where $B = -D^{-1}(L+U)$ is the iteration matrix and $d = D^{-1}b$. A necessary and sufficient condition for the convergence of the Jacobi iteration is $\rho(B) < 1$, where $\rho(B)$ denotes the spectral radius of B. In terms of the system matrix A, a sufficient condition is strict diagonal dominance of A, or diagonal dominance and irreducibility [Sa00].

The parallelization of this method is straightforward. Each compute unit may compute a part of the new iteration vector x^{k+1}. Note that for computing its part of the new iterate x^{k+1}, any compute unit potentially uses components of the preceding iterate x^k which belong to other compute units. This requires a synchronization of the compute units after each iteration to make sure that all needed values are updated from the last iteration.

The idea of asynchronous iteration is to overcome the synchronization requirements. On the theoretical level, this is accomplished by introducing a shift function s and an update function u in the iteration:

$$x_i^{k+1} = \begin{cases} \frac{1}{a_{ii}}\left[b_i - \sum_{j \neq i} a_{ij} x_j^{k-s(j)}\right] & \text{if } i = u(k) \\ x_i^k & \text{if } i \neq u(k) \end{cases} , \ (k = 0, 1, \ldots)$$

The shift function allows to use not only values from the last iteration, but also older or newer values. The update function chooses one component at a time to be updated, leaving the other components unchanged [An11a]. A sufficient condition for convergence is uniform boundedness of s, and u must take each value in $\{1, \ldots, n\}$ infinitely often, and $\rho(|B|) < 1$ [CM69].

A natural modification of the basic asynchronous scheme is the aggregation of components into blocks [Ba99]. Let L be the number of blocks, and $I_l \subset \{1, \ldots, n\}$ be the index set of all components belonging to block $l \in \{1, \ldots, L\}$. The block-asynchronous iteration reads

$$i \in I_l \ : \quad x_i^{k+1} = \frac{1}{a_{ii}}\left[b_i - \sum_{j \notin I_l} a_{ij} x_j^{k-s(k,j)} - \sum_{j \in I_l, j \neq i} a_{ij} x_j^k\right] \qquad (k = 0, 1, \ldots) .$$

This scheme is synchronized only with respect to the vector components within each block. The block scheme implies a decomposition of the systems matrix A into diagonal and off-diagonal parts

$$D_l = \left(a_{ii}\right)_{i \in I_l} , \quad A_l^{\text{diag}} = \left(a_{ij}\right)_{i,j \in I_l, i \neq j} , \quad A_l^{\text{offdiag}} = \left(a_{ij}\right)_{i \in I_l, j \notin I_l}$$

and a decomposition of the vectors x and b into local parts

$$x_l^{\text{local}} = \left(x_i\right)_{i \in I_l} , \quad b_l^{\text{local}} = \left(b_i\right)_{i \in I_l} , \quad x_l^{\text{non-local}} = \left(x_j\right)_{j \notin I_l} .$$

Such block decomposition is sketched for A and x in Figure 1. The update step for any block l then reads

$$x_l^{\text{local}} \leftarrow D_l^{-1}\left[b_l^{\text{local}} - A_l^{\text{diag}} x_l^{\text{local}} - A_l^{\text{offdiag}} x_l^{\text{non-local}}\right] .$$

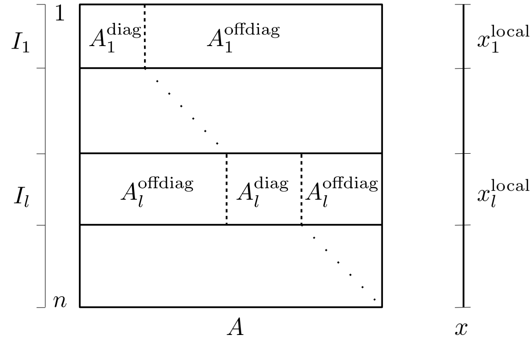

Fig. 1: Decomposition of the system matrix A and solution vector x into blocks.

Note that the actual block sizes in the decomposition depend on the number of compute units, and on the load distribution among them. For balanced load distributions, the local block sizes decrease with increasing number of compute units, while the non-local parts grow.

The block-asynchronous scheme can be extended by performing multiple iterations on the local block before updating the values in the non-local vector part. We denote the resulting algorithm as async-(m) to indicate m local steps between non-local updates. Obviously, each block can be mapped to one compute unit, resulting in Algorithm 2. We use async-(m) with a relative stopping criterion based on a given tolerance $\delta > 0$.

Algorithm 2 async-(m)

1: Set initial solution x, tolerance $\delta > 0$.
2: Compute initial residual $r^0 = r = b - Ax$.
3: **while** $\|r\| > \delta \|r^0\|$ **do**
4: **for** all blocks $l = 1, \ldots, L$ in parallel **do**
5: **for** k=1,...,m **do**
6: $x_l^{\text{local}} \leftarrow D_l^{-1} \left[b_l^{\text{local}} - A_l^{\text{diag}} x_l^{\text{local}} - A_l^{\text{offdiag}} x_l^{\text{non-local}} \right]$
7: **end for**
8: **end for**
9: Update $x_l^{\text{non-local}}$ with corresponding values from other blocks.
10: Compute residual $r = b - Ax$.
11: **end while**

3 Experimental setup

3.1 Linear problem

In our experiments, we use the linear system of equations arising from a finite element discretization of the two-dimensional Poisson equation [EG04]. This equation can be used to model the equilibrium heat distribution in a physical domain with given environmental

temperature and heat sources or sinks. The problem definition reads

$$
\begin{aligned}
-\Delta u = f \quad & \text{in } \Omega \,, \\
u = g \quad & \text{on } \partial\Omega_{\mathrm{D}} \,, \\
\nabla u \cdot n = 0 \quad & \text{on } \partial\Omega_{\mathrm{N}} \,,
\end{aligned}
$$

where $\Omega \in \mathbb{R}$ is the physical domain, f represents any heat sources or sinks and g is the environmental temperature given through the Dirichlet condition on the boundary part $\partial\Omega_{\mathrm{D}}$. Thermal insulation is modeled by the homogeneous Neumann boundary condition on the boundary part $\partial\Omega_{\mathrm{N}}$. For our experiments, we chose the domain Ω to be the unit square. Our finite element discretization with 262,144 mesh cells results in $n = 263{,}169$ unknowns and 2,362,369 non-zero elements for the system matrix A.

3.2 Implementation for GPU-accelerated multi-core shared and distributed memory HPC clusters

Our implementation spans three levels of parallelism. It supports multi-node distributed memory systems where the nodes are connected by a network. Communication between the nodes is done by data transfer over the network using MPI [Me12]. On the node level, it supports both multi-core shared memory systems by means of OpenMP [Op13] as well as CUDA-capable devices [NV14a].
The implementation is integrated in the HiFlow[3] package [An12]. It uses the MPI-parallelized matrix and vector data structures for input and for the MPI communication between nodes. The matrix and the vectors are distributed among the MPI processes, thus defining the block decomposition. The communication pattern is derived from the matrix structure and avoids any unnecessary data transfer. Only vector components corresponding to non-zero entries in the off-diagonal matrix parts of other MPI processes are transferred.
In Algorithm 2, the parallelism of the local block updates corresponding to steps 4-8 is achieved by concurrency of the MPI processes. All computations of the error correction solver are either executed on the host CPUs, or on the accelerator devices. Again, the CPU implementation is parallelized with OpenMP on the node level, while the accelerator version is implemented with CUDA. The update step 9 implies MPI communication and, if the CUDA version is used, data transfer between host and devices.

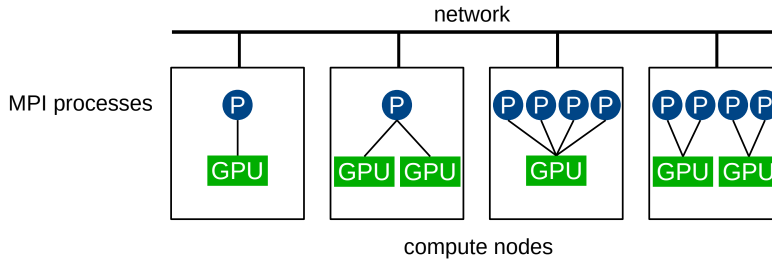

Fig. 2: Supported configurations of MPI process scheduling among compute nodes and GPU usage.

One or multiple MPI processes may be scheduled onto each node. Within each node, the MPI processes can use multiple GPUs. The actual utilization may be configured depending on the number of MPI processes and on the number of available devices. If only one MPI process is scheduled onto a node, this process may use all available devices on that node, as sketched in the two left configurations in Figure 2. In case of multi-GPU usage of a single MPI process, the matrix and vector blocks of this process are further split into sub-blocks as depicted in Figure 3. However, if multiple MPI processes are scheduled onto the same node, GPU utilization must be split such that each process uses only one of the available devices, see the two right configurations in Figure 2. This limitation is imposed by a constraint of the GPU architecture, which we briefly explain in the following.
Each host process establishes its own CUDA context, but there can only be one CUDA context active at a time on the device. If several host processes access the GPU, a time-sliced scheduler switches between contexts to serve them, which implies a serialization. To efficiently use the same device by several host processes, the multi-process service (MPS) [NV14b] can be used. With MPS, host processes connect to the MPS server instead of directly accessing the device. The MPS server maps the different host CUDA contexts into one context on the GPU. This avoids the context switching and enables to benefit from the Hyper-Q feature of devices based on the Kepler architecture [NV12]. With Hyper-Q, up to 32 independent CUDA streams may be executed concurrently on the GPU. The drawback of MPS is that the MPS server can only manage one device such that any process can only use one GPU. If multiple devices are available on the node, one MPS server instance is needed for each device, and host processes need to connect to exactly one of them. A practical way to meet these technical requirements can be found in [WSC14]. For Algorithm 1, all steps except the solution of the error correction equation in step 6 are implemented in C++ for execution on CPUs. In addition to the MPI parallelization for distributed systems, all local computations are parallelized with OpenMP to exploit multi-core shared memory nodes. The error correction solver itself uses Algorithm 2 and can be executed either on CPUs or on accelerators.

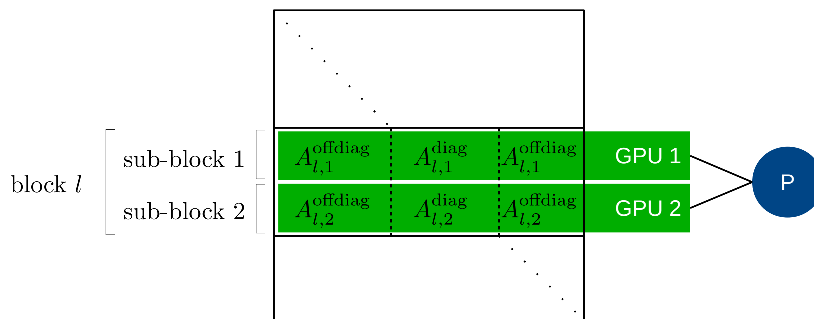

Fig. 3: Sub-block decomposition in case of multi-GPU usage by a single MPI process.

3.3 Solver parameters

As the working precision, denoted high precision in the context of MPIR, we chose IEEE 754 double precision floating point format, and as low precision we chose IEEE 754 single precision floating point format [In85]. We set an absolute tolerance of $\varepsilon = 10^{-6}$ as stopping criterion for the iterative refinement method in Algorithm 1. This absolute tolerance is achievable in both double and single precision. For the error correction solver, we chose a relative tolerance of $\delta = 10^{-1}$ in Algorithm 2. This resulted in several error correction loops, each improving the high precision residual by the factor 10^{-1}.

3.4 Hardware and measurement system

Our test system consisted of one compute node equipped with 4 x Intel Xeon E-4650, 512 GByte DDR3 main memory and 2 x Nvidia Tesla K40. We used GCC compiler version 4.8.2, OpenMPI version 1.6.5, CUDA version 6.5.12, and NVIDIA device driver version 340.65.

For power measurement, we used the ZES Zimmer Electronic Systems LMG450 external power meter. Our test system comprises two power supply units, each connected with one line to the external power source. The LMG450 has four independent measurement channels. We used one channel for each of the two input lines, and the other two channels were left unused. We attached the power sensors of the LMG450 to the input lines between the external power source and the power supply units of the compute node. Thus, we measured the total power consumption of the whole node. We used the maximum possible sampling rate of 20 Hz of the LMG450 power meter. The measurement was controlled using the `pmlib` tool [Ba13]. We instrumented the solver code using the `pmlib` client API to measure exactly that portion of the overall program which constitutes the solution process. This excluded all initialization overhead from the measurements. The `pmlib` server ran on

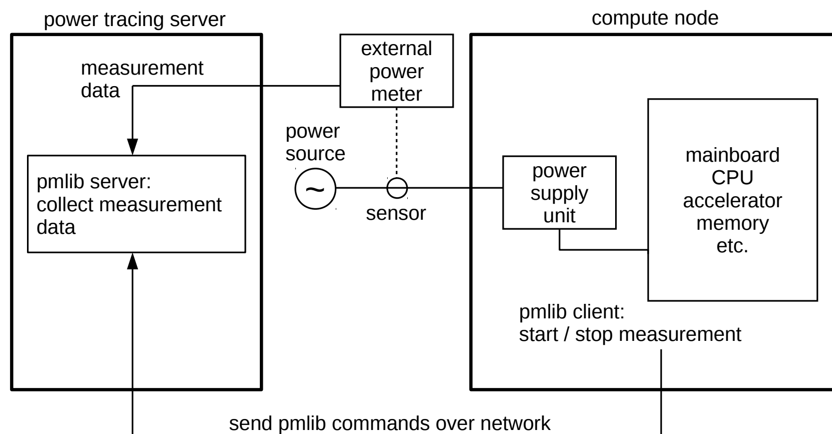

Fig. 4: Measurement setup using an external power meter controlled by the pmlib tool.

a separate machine to avoid a perturbation of the system under investigation. The setup is shown in Figure 4.

4 Results

Through empirical testing, we figured out a number of $m = 20$ local block updates to be a reasonable choice for the linear system at hand. We carried out three series of tests:

1. **MPIR async-(20) GPU**
   Mixed precision iterative refinement using block-asynchronous iteration as error correction solver in single precision running on the GPUs.
2. **dp IR async-(20) GPU**
   Iterative refinement using block-asynchronous iteration as error correction solver in double precision running on the GPUs.
3. **MPIR async-(20) CPU**
   Mixed precision iterative refinement using block-asynchronous iteration as error correction solver in single precision running on the host CPUs.

We defined these test series to evaluate two effects. On the one hand, to evaluate the effect of using single precision error correction in contrast to using double precision error correction. On the other hand, to evaluate the effect of using accelerators in contrast to using only the host CPUs.
Figures 5 and 6 show plots of the performance related data of runtimes and total number of iterations. The parallel configuration is denoted $p \times t$, where p is the number of MPI processes, and t is the number of OpenMP threads per MPI process. We scheduled the MPI processes to run on distinct CPUs when using $p = 1, 2, 4$, and to equally share CPUs when using $p = 8, 16, 32$. The number of OpenMP threads was chosen to use all of the eight cores of the CPU available for the corresponding MPI process.

The host-only test runs from MPIR async-(20) CPU showed a reduction of the runtime for p ranging from 1 to 32. The fact that speedups were clearly inferior to the ideal linear speedup can be explained by the increasing number of iterations and the increased communication overhead. The phenomenon of increasing number of iterations for growing p

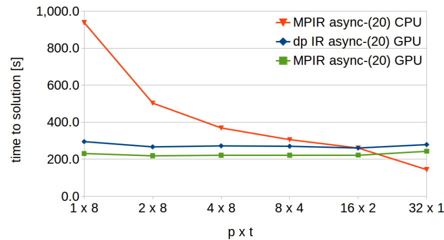

Fig. 5: Time to solution plot.

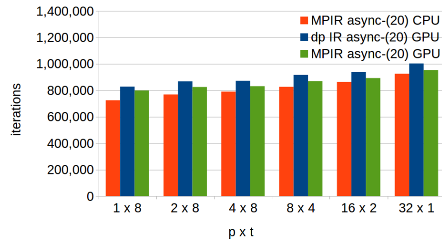

Fig. 6: Total number of iterations.

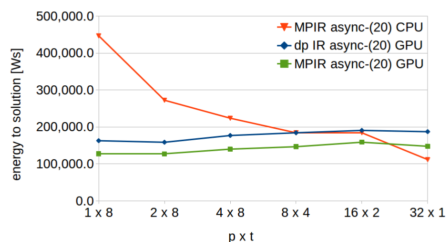

Fig. 7: Energy to solution plot.

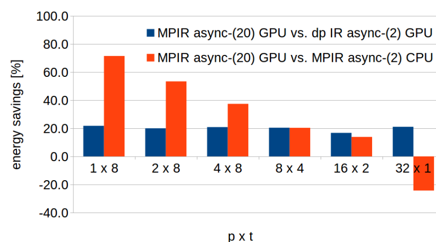

Fig. 8: Energy savings of MPIR async-(20).

was similar for all methods we tested. The reason is the decomposition of the matrix and vectors into smaller blocks, causing the local vector parts to shrink and the non-local parts to grow. Thus, the local block updates include more potentially outdated information from the non-local vector parts, and an increased number of overall iterations is necessary to compensate this effect.

In contrast, the methods using the GPUs showed a different behavior. The runtimes were nearly constant for $p = 2,4,8,16$ with slightly larger runtimes for $p = 1$ and $p = 32$. The GPU methods perform almost the whole computations on the GPUs. Only the double precision residual computation in step 9 of Algorithm 1 is performed on the host CPUs, but this computational effort is negligible. Instead, the nearly constant runtime reflects the fact that the sum of the problem sizes of all processes using the same GPU is constant, namely half of the total problem size.

Using single precision error correction instead of double precision gave approximately 20% improvement in the runtimes. The pure floating point arithmetic is twice as fast in single precision than in double precision, but our algorithms require data transfer between host and devices and across MPI processes for each update of the non-local vector parts. Although we used the fast transfer between devices and page-locked host memory, these memory copy operations required a substantial portion of the overall runtime. The high precision residual computation on the CPUs and the typecasts in case of mixed precision were negligible in this context, since they were not performed every 20 iterations, but only once for each error correction solving loop. Altogether, the runtimes of GPU methods were advantageous over the CPU method for $p = 1,2,4,8$. For $p = 16$, dp IR async-(20) GPU and MPIR async-(20) CPU had nearly equal runtime, while MPIR async-(20) GPU was still faster. Finally, for $p = 32$ the CPU method outperformed the GPU methods.

As Figure 7 shows, the energy consumption of the methods strongly correlated to the runtimes. Figure 8 shows the energy savings of MPIR async-(20) GPU compared to dp IR async-(20) GPU and MPIR async-(20) CPU. We calculated the percentages relative to the latter two methods. Using mixed precision instead of only double precision in the GPU methods gave savings of about 20% with slight variances for $p = 1$ and $p = 16$. However, performing the computations on the GPUs instead of CPUs gave massive savings of $\approx$71% for $p = 1$ and $\approx$53% for $p = 2$. We observed still remarkable savings of $\approx$37% for $p = 4$, while the benefit lay around 20% for $p = 8,16$. Only in the case $p = 32$, the CPU method consumed $\approx$24% less energy than the GPU method. We calculated this last percentage relative to MPIR async-(20) GPU.

5 Conclusion

We investigated block-asynchronous iteration methods for solving linear systems of equations with respect to performance and energy consumption. We introduced the mathematical background of mixed precision iterative refinement and of block-asynchronous iteration methods. We presented our implementation of these methods with support for distributed memory systems be means of an MPI parallelization, as well as shared memory support using OpenMP, and support of CUDA-capable accelerator devices. We ran a series of tests on a compute node equipped with four Intel Xeon E-4650 CPUs and two Nvidia Tesla K40 GPUs. We varied the number of MPI processes and OpenMP threads for the different test runs. All GPU tests used both devices. We designed the tests to assess the effect of using mixed precision instead of plain double precision computations, and to assess the effect of employing accelerator devices for the computations instead of only using host CPUs. We measured performance in terms of runtime, and energy consumption was measured with the help of an external high precision power meter.
We found that massive runtime and energy savings of more than 70% are possible on GPU-accelerated systems compared to CPU-only platforms. However, the actual amount of saved energy for a particular test run depends on the parallel configuration of MPI processes and OpenMP threads. We also found that CPU-only computations may outperform the GPU-accelerated methods if enough CPU resources are available. Also, we found that using mixed precision instead of only double precision gives a benefit of roughly 20% for runtime and energy consumption in the GPU tests. The frequent data transfer between host and devices imposes a substantial overhead which diminishes the impact of the doubled performance of the single precision floating point arithmetic.
Our results show that using accelerators for block-asynchronous iteration methods combined with mixed precision can lead to tremendous benefits in terms of runtime and energy consumption. The largest benefits can be expected for small host systems with only 16 or even less cores. This fits many HPC systems where often not more than two CPUs are available per node. On the other hand, large host systems or "fat nodes" may provide superior performance over the GPUs.

Acknowledgement

This work was supported by the European Commission as part of the collaborative project "Exa2Green - Energy-Aware Sustainable Computing on Future Technology"(research grant no. 318793).

References

[An11a] Anzt, H.; Dongarra, J.; Gates, M.; Tomov, S.: Block-asynchronous multigrid smoothers for GPU-accelerated systems. EMCL Prepr. Ser., 15, 2011.

[An11b] Anzt, H.; Dongarra, J.; Heuveline, V.; Luszczek, P.: GPU-Accelerated Asynchronous Error Correction for Mixed Precision Iterative Refinement. EMCL Prepr. Ser., 17, 2011.

[An12] Anzt, H.; Wilhelm, F.; Weiß, J.P.; Subramanian, C.; Schmidtobreick, M.; Schick, M.; Ronnas, S.; Ritterbusch, S.; Nestler, A.; Lukarski, D.; Ketelaer, E.; Heuveline, V.; Helfrich-Schkarbanenko, A.; Hahn, T.; Gengenbach, T.; Baumann, M.; Augustin, W.; Wlotzka, M.: HiFlow3: A Hardware-Aware Parallel Finite Element Package. pp. 139–151, 2012.

[An13] Anzt, H.; Tomov, S.; Dongarra, J.; Heuveline, V.: A block-asynchronous relaxation method for graphics processing units. J. Parallel Distrib. Comput., 73:1613–1626, 2013.

[Ba99] Bai, Z.Z.; Migallon, V.; Penades, J.; Szyld, D.B.: Block and asynchronous two-stage methods for mildly nonlinear systems. Numerische Mathematik, 82:1–20, 1999.

[Ba09] Baboulin, M.; Buttari, A.; Dongarra, J.; Kurzak, J.; Langou, J.; Langou, J.; Luszczek, P.; Tomov, S.: Accelerating scientific computations with mixed precision algorithms. Computer Physics Communications, 180:2526–2533, 2009.

[Ba13] Barrachina, S.; Barreda, M.; Catalan, S.; Dolz, M.F.; Fabregat, G.; Mayo, R.; Quintana-Orti, E.S.: An Integrated Framework for Power-Performance Analysis of Parallel Scientific Workloads. In: ENERGY 2013: The Third Int. Conf. on Smart Grids, Green Communications and IT Energy-aware Technologies. 2013.

[CM69] Chazan, D.; Miranker, W.: Chaotic relaxation. Lin. Alg. Appl., 2:199–222, 1969.

[EFS05] El Baz, D.; Frommer, A.; Spiteri, P.: Asynchronous iterations with flexible communication: contracting operators. J. Comp. App. Math., 176:91–103, 2005.

[EG04] Ern, A.; Guermond, J.-L.: Theory and Practive of Finite Elements. Springer, 2004.

[FS00] Frommer, A.; Szyld, D.: On asynchronous iterations. J. Comp. Appl. Math., 123:201–216, 2000.

[In85] Institute of Electrical and Electronics Engineergs: , IEEE Standard for Binary Floating-Point Arithmetic, 1985.

[Me11] Meister, A.: Numerik linearer Gleichungssysteme. Vieweg+Teubner, 2011.

[Me12] Message Passing Interface Forum: , MPI: A Message Passing Interface Standard, Version 3.0, 2012.

[NV12] NVIDIA Corporation: , Kepler GK110, 2012.

[NV14a] NVIDIA Corporation: , CUDA Toolkit Documentation v6.5, 2014.

[NV14b] NVIDIA Corporation: , Multi-Process Service, 2014.

[Op13] OpenMP Architecture Review Board: , OpenMP Application Program Interface, 2013.

[Ro69] Rosenfeld, J.: A case study in programming for parallel processors. Commun. ACM, 12:645–655, 1969.

[Sa00] Saad, Y.: Iterative Methods for Sparse Linear Systems. 2 edition, 2000.

[WSC14] Wende, F.; Steinke, T.; Cordes, F.: Multi-threaded Kernel Offloading to GPGPU Using Hyper-Q on Kepler Architecture. ZIB Report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, 2014.

[ww14] www.green500.org: , The Green500 List, November 2014.

[ww15] www.top500.org: , TOP500, June 2015.