The Impact of Global Communication Latency at Extreme Scales on Krylov Methods

Thomas J. Ashby[1,2], Pieter Ghysels[1,3], and Wim Heirman[1,4] and Wim Vanroose[3]

[1] Intel/Flanders Exascience Lab, Leuven, Belgium
[2] Imec, Leuven, Belgium
[3] Universiteit Antwerpen, Antwerp, Belgium
[4] Universiteit Gent, Ghent, Belgium

Abstract. Krylov Subspace Methods (KSMs) are popular numerical tools for solving large linear systems of equations. We consider their role in solving sparse systems on future massively parallel distributed memory machines, by estimating future performance of their constituent operations. To this end we construct a model that is simple, but which takes topology and network acceleration into account as they are important considerations. We show that, as the number of nodes of a parallel machine increases to very large numbers, the increasing latency cost of reductions may well become a problematic bottleneck for traditional formulations of these methods. Finally, we discuss how *pipelined KSMs* can be used to tackle the potential problem, and appropriate pipeline depths.

Keywords: Krylov methods, extreme scaling, global communication, reduction latency, pipelining, latency hiding

1 Introduction

Krylov Subspace Methods (KSMs), such as GMRES, CG, BICGSTAB and numerous other variants, are widely used numerical tools for solving linear systems of equations $Ax = b$. They are popular because they are easy to parallelize, meaning that they can be run quickly, and they do not require direct manipulation of the matrix A, only matrix–vector products Av, and so can be used easily with sparse matrices without memory capacity problems. They are the tool of choice for solving extremely large sparse systems of equations on parallel distributed memory machines.

The general trend in large scale parallel computing is towards more cores per node, and more nodes. Because KSMs are such important tools, it is therefore a valuable exercise to model the future performance of the algorithms assuming this machine architecture trend will continue into the future. The potential parallel inefficiency of the KSMs at extreme scales on distributed memory machines is the problem we investigate in this paper.

The dependency structures of the KSM basic operations when vector elements are distributed across a parallel machine are given in Table 1. If the work per node is fixed, then of these operations the only ones which must *necessarily* get more costly as the number of machine nodes increases are the scalar products. Reduction operations on distributed memory machines are dominated by their latency cost, and have necessarily rather poor parallel efficiency for the non-local parts of the computation. Consequently the relative cost of this operation to the others and what the schedule of operations will end up looking like are both important to gauge the overall parallel efficiency as the scalar products get more expensive.

Pipelined (a.k.a. *Communication-Hiding*) KSMs have been proposed as an alternative formulation of KSMs to tackle the case when scalar products become an efficiency bottleneck [7]. The approach requires modifying the algorithms of the KSMs to alter their dependency structure. The extra scheduling freedom thus introduced is then used to implement a form of pipelining that can significantly improve the parallel efficiency of the algorithms.

The contribution of this paper is to show in detail how reduction latencies may become a problem on future exascale machines, and give a first quantification of what degree of pipelining in the KSMs may be required to avoid that latency becoming a bottleneck.

2 Example Problem

As the sparsity pattern, and thus form of the matrix–vector product, is problem specific, we choose a specific instance to make discussion easier. For our example problem we take a simple finite difference stencil, being the nearest neighbour in each direction on a regular 3D grid. The layout is the natural one, with contiguous sub-cubes of the grid allocated to each machine node. Although this problem is relatively simple, it is a reasonable approximation to sparse matrices that have a relatively low degree of connections between grid points. As there are many problems that use grids derived from physical problems with low spatial connectivity, including many finite element problems, this is a useful yardstick.

We stick to cubic grids for simplicity. We have chosen four local grid sizes; 1, 50^3, 100^3 and 200^3 per machine node (this amount is then further subdivided over sockets/cores). The largest size gives around 8 million Degrees of Freedom (DoFs). Although these numbers are not large for full problem sizes, there are several cases when they are relevant. Firstly, particle-mesh simulations often have high particle to mesh ratios resulting in thinly spread linear systems. Secondly, the use of multigrid for preconditioning (or as a solver) is interesting in that KSMs have been used for the “bottom solve” in a U-cycle, that is to solve the system once further restriction steps are abandoned as the resulting system will suffer from too much parallel inefficiency [8]; if the bottom solve requires enough iterations then pipelining can be useful. Thirdly, strong scaling can lead to thinly spread problems. The smallest problem size (i.e. 1) is intended to show the limit case for the problem rather than being a practical grid size.

3 Available Parallelism in *Krylov Subspace Methods*

Table 1. Krylov Subspace Method basic operations

Operation	Notation	Dependencies
Matrix-vector multiplication	Ax	Depends on sparsity of A and machine topology; usually localised
Scalar products	$< v, w >$, $\|\|v\|\|$	Global tree
Vector operations	αv, $v \pm w$	Local (element wise)

KSM basic operations and their dependencies on a distributed memory machine are given in Table 1. The total available parallelism of a KSM is determined by the dependencies within and between these basic operations.

3.1 Parallelism within Operations

The two classes of operations with dependencies that result in communication are the matrix-vector multiplication and the scalar products. KSMs are popular for use with sparse linear systems. Of particular interest are those systems where the sparsity can be modelled by a graph with a low number of edges that approximates a grid structure.

When mapping such a problem and associated matrix onto a parallel machine, it is a natural mapping to distribute the graph on the machine such that a small graph neighbourhood becomes a small machine neighbourhood wherever possible, e.g. in our example problem, grid neighbours become machine neighbours. If the machine network supports low latency communication for all resulting neighbourhoods, then the cost of the matrix-vector product is either fixed or dominated by data transmission cost, and this doesn't change if the problem size is increased by weak scaling. Such an operation can in principle be weakly scaled to arbitrary problem sizes without significantly changing its parallel efficiency.

The reduction operations on the other hand, do not have a fixed level of parallelism. Their non-local operation structure is one large binary tree of operations where the leaves are the vector elements. As the number of input values grows, the height of the tree also grows, albeit slowly. Higher levels of the tree have many fewer operations than available computational resources, thus reductions have bad parallel efficiency, which gets worse as the machine gets larger; the ratio of computation to number of communication events is very low, and the amount of data sent is also low, so the whole operation is dominated by network latency costs, with most of the computing elements spending most of the time idle waiting for values to arrive.

3.2 Parallelism between Operations

In the standard KSM algorithms, scalar products are used in such a way that their parallel efficiency will be affected at extreme scales, and potentially before. The start of an algorithm iteration involves applying the matrix (also called applying the stencil, as our sparse matrix is in stencil form) to produce a new vector, then calculating some scalar products on that new vector. The result of the scalar products are used to construct the input for the next stencil operation, and so there is a dependency cycle between stencil and scalar product operations. This pattern of dependencies occurs in all the methods in one form or other. Thus, the ratio of the reduction time vs. the time required to do the communication necessary for a stencil operation is a proxy to gauge general parallel efficiency of standard KSMs, from the point of view of communication costs. Note that in some methods, i.e. GMRES, there is a dependent sequence of at least two scalar products per stencil.

The validity of the stencil communication time vs. reduction time metric depends on some assumptions about local operations. As well as the two acts of communication (with associated computation for the reduction), a KSM iteration requires the local part of the stencil operation and some purely local vector operations to be performed. The relationship of these to the reduction time and stencil communication time is discussed in sec. 6; it suffices here to say that in some circumstances the reduction time stencil communication time dominate. When this holds, a reasonable portion of the schedule can be spent waiting for the latency dominated and parallel-inefficient non-local scalar products.

4 Relative Cost of KSM Operations at Extreme Scales

We start by modelling the off-node communication costs in this section. Although our model is relatively simple, we note that the problem we are analysing results from the value of the ratios. As such, the absolute values we use in our model may be wrong, but the conclusions that we come to will still be valid provided the actual ratios (now, at exascale or later) are similar to the ones we report on.

A formula for the relative cost of the communicating operations at extreme scale is given below:

$$\frac{\delta_r}{\delta_s} = \frac{\beta + \sum_{i=0}^{T}(\lambda_r^i + \delta_n)}{\lambda_s + (D_{\text{face}}/\theta_s)} \tag{1}$$

where δ_r and δ_s are the total time taken for the reduction and stencil operations respectively. We assume that the reduction takes the form of a series of non-local communication steps, where data is transmitted on the network, and local reduction steps where the operations are actually carried out on the data present. T is the height of the tree of non-local reduction steps, λ_r^i is the latency of traversing the network links and switches for reduction step i, and δ_n is the time for executing the local reduction operations on a compute node or network switch to make an intermediate or final result. β is the cost of broadcasting the reduction result. λ_s is the maximum latency for traversing the network links and switches to logically neighbouring nodes for a single stencil step, D_{face} is the volume of data to transfer to a neighbour, representing the face of the local grid cube, and θ_s is the per-face bandwidth available over the links used for the stencil operation. Our choice of stencil means that what gets communicated to neighbours are the six faces of the locally allocated sub-cube of the problem grid, with the longest time taken to communicate any face determining δ_s. Stating the bandwidth parameter as "per face" allows us to model different network topologies by deriving the per face bandwidth from the network link bandwidth and the topology.

The reduction operations we consider are of type *all-reduce*, meaning that the result of the operation should be made available to all cores. There are several ways to implement such all-to-all operations. We have chosen a reduction to a single value followed by a broadcast for the following reasons. True N-to-N broadcast and butterfly networks of high radix are unlikely to be implemented for a very large number of nodes due to the prohibitive equipment cost. Also, mapping such algorithmic approaches on to the network topologies we consider here is unlikely to gain much if anything in terms of the latency cost after taking switch contention into account.

4.1 Applying the Model

To use formula 1 we need to fill in the parameters; these, and associated assumptions, are given below.

Nodes: Node count estimates for future exascale systems vary, from around 100,000 for "fat nodes" to around 1,000,000 for "thin nodes" [13]. By *node* we mean the parts of a machine with their own separate network interfaces (usually containing several *sockets* and/or *accelerators*, each supporting multiple *cores* or *CPUs*). On the assumption that we will get there eventually, at exascale or shortly thereafter, we take the larger number to illustrate the problem of latency when scaling to larger node counts.

Network: The latency costs λ_s and λ_r^i depend on the corresponding cable lengths, a per switch latency, which we assume is the same for all switches in a given network, and an at-node network to user-process (and vice versa) transfer latency. Our basic model for cables is based on a square warehouse of densely packed cabinets each measuring $1m^2$, with maximum 500 nodes per cabinet. Cable distance for longer cables is calculated using Manhattan distance between cabinets, and cable latency is based on signals propagating at the speed of light. We reduce cable latency within a cabinet to zero to simplify the model (cable latency costs are dominated by the longer links). We derive the link bandwidth (Eqn. 2) and router latency (Eqn. 3) from a simple model of a router, based on the bi-directional router bandwidth and network radix (i.e. number of in or out ports):

$$\text{link}_{BW} = \text{router}_{BW}/(2 \times \text{radix}) \tag{2}$$

$$\text{router}_{latency} = 10ns + 5 \times log_2(\text{radix}) \tag{3}$$

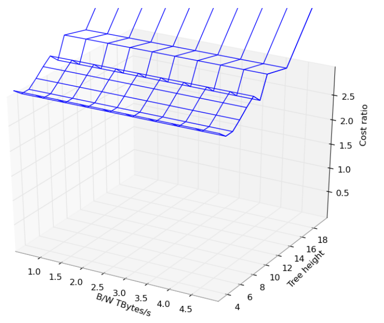
The constant (10 nanoseconds) in Eqn. 3 is to take account of SerDes and signalling. The factor 2 Eqn. 2 is to turn a network radix into the total number of I/O ports on a network router (for bi-directional bandwidth). All router chips in a given model have the same bandwidth. In the case of multiple networks, the number of I/O ports can be different for each network. We set the local node latency cost to move data from a network interface into a user process or back the other way according to machine type (details below). The height of the non-local reduction tree, T, is given by the number of nodes (given above) and the non-local reduction tree radix.

4.2 Machine architecture

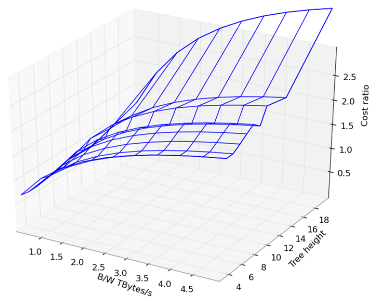
Although for our case the scalar products are the limiting factor for available parallelism in KSMs as the size of a problem grows, the mapping of the KSMs onto a machine architecture needs to be taken into account to understand actual parallel efficiency. The main concerns here are the network topology and the available bandwidth. To apply the model, we take two example machine architectures. The first is a machine where the grid embeds in such a way that nearest neighbour communication links are available, and there is a separate Accelerated Reduction Tree (ART) for reductions; this is a close match with the program dependencies. An ART is a separate network where the switches are capable of buffering the incoming values and executing a reduction operation on them directly before sending the result further up the tree. The reduction requires one trip up the ART to compute the result, and one trip down it to make it available to all cores. Machines with accelerated reduction networks include [2,4,3].

The second is a machine with a single high-radix fat-tree network; this represents a machine where a different network has been chosen due to cost and flexibility issues, and the network architecture is a less good fit to the dependencies. In the case of a standard indirect tree-like network with no in-built acceleration, the logical form of the reduction would still be a tree of a certain radix, but where the local reduction steps are carried out at the leaves of the tree network. Note that the individual links in the operation tree would be mapped onto various sets of links in the machine tree network, and thus would have different total switch latency costs, unlike the ART where only cable length varies.

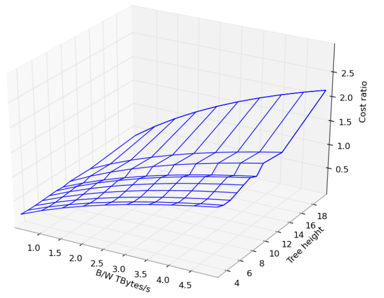
After using the parameters and assumptions in sec. 4.1, we need per face bandwidth θ_s, non-local reduction step tree radix, local reduction cost δ_n, and broadcast latency β to derive a value from our models. These values are either specific to the machine architecture, or used as range parameters to generate plots.



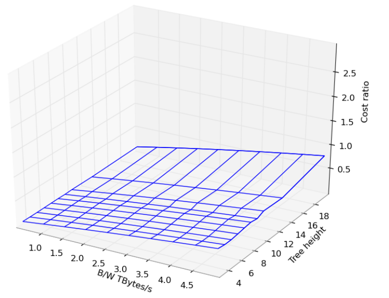
GPN = 1, TFV ≈ 48 bytes



GPN = 50^3, TFV ≈ 117 KB



GPN = 100^3, TFV ≈ 468 KB



GPN = 200^3, TFV ≈ 1875 KB

Fig. 1. δ_r/δ_s for **mesh + ART**. The x-axis is router bandwidth (0.5 to 5 TBytes/s), y-axis is increasing ART height (i.e. the different heights given by the radices in the range 200 to 2), and z-axis is the resulting δ_r/δ_s. GPN is Grid Points per Node, TFV is Total Face data Volume (i.e. per face volume ×6).

θ_s is derived from the router bandwidth. The bi-directional router bandwidth is a free parameter which we vary in the plots from 0.5 to 5 TBytes/s. The local reduction costs are based on an assist circuit serially executing floating point operations at 2 GHz. To get the height of the tree of non-local reduction steps, we need a radix. We fix this radix based on the machine architecture. For an ART, the machine ART radix thus becomes a plot parameter. For a fat-tree, we make the radix of the non-local reduction tree the same as the network radix, so there is again a single architecture radix parameter. We use the radix parameter to generate different tree heights, and plot the tree height and router bandwidth against the resulting ratio of reduction to stencil time cost.

Mesh and ART We take the link latency for the stencil operation on the mesh network, λ_s, based on a cable to each neighbouring node at most 1 metre long, and the assumption that each node has a Network Interface Controller (NIC) that also acts as the switch for the mesh network. We assume the bandwidth to the NIC from the cores is at least as much as the total off-node bandwidth, and we divide the router chip bandwidth between the off-node links only. We assume that the face exchange can be done in parallel, so that all torus router links are used simultaneously at full link bandwidth.

The local reduction latency δ_n is a function of the local operations in the ART. We assume that the ART supports broadcast directly with no at-switch copy overhead, so β is just derived from switch and cable crossing costs on the way down the tree. The core-to-NIC cost is 100ns, based on reported numbers for a specialised low overhead system [6], which is in line with design choices such as reduction acceleration hardware.

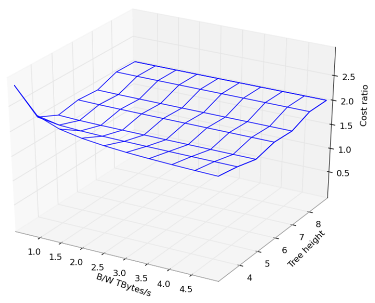
In Fig. 1 we plot the relative cost of the two operations, where the parameters are ART height and bi-directional router bandwidth. The number of bits for a cube face varies from figure to figure; the maximum is $200^2 \times 64$ bits, that is the face of a local 3D cube of double precision floats that has dimension 200^3, and the minimum is 1×64, the case where the local cube is a single point.

The model shows that the ratio δ_r/δ_s gets worse with taller tree heights and increased router bandwidth, with the exception that the results for the smallest face volume are not affected by the bandwidth parameter. Whilst the ratios are mostly under 1 for the largest problem size (200^3), they are mostly above 1 for the other sizes, and there is a significant amount of the design space above 1.5 for both 50^3 and 100^3.

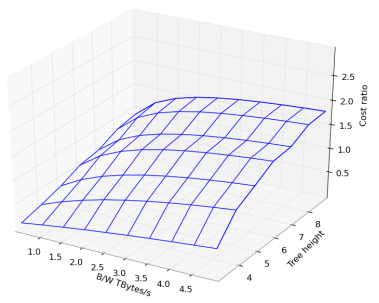
We consider ratios less than 0.25 to be largely irrelevant for pipelining, on the basis that a chain of two such scalar products still only costs at most half as much as a stencil in a standard KSM, and thus the parallel efficiency is already reasonable. This occurs for very few points, suggesting that for the *mesh + ART* the standard KSMs should show a reasonable benefit from pipelining to improve parallel efficiency for these problem sizes.

Fat-tree For the fat-tree the nearest neighbour latency cost for the stencil, λ_s, will be the full cost of a trip via the top node of the tree. Thus the radix and resulting height of the fat-tree will affect the stencil time δ_s. Cable latencies are again derived from the tree layout model, with longer cables now affecting both operations. Given that each node has one cable link to its immediate parent switch, which is not attached to any node, the bandwidth available for parallel face exchange in the stencil operation is one sixth that of the network link bandwidth (as the stencil is 3D); this is different from the *mesh + ART* case. We use a core-to-NIC cost of 800ns for this architecture, based on reported numbers for hardware that would be used in a commodity cluster ([1] reports a one-way MPI latency of 1.6μs, giving a maximum of 800ns per core).

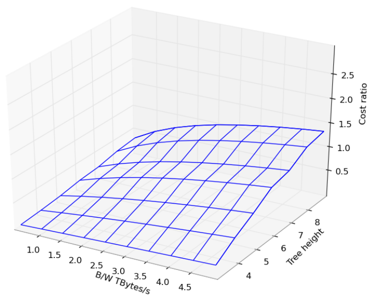
A reduction on a network without full acceleration gives rise to a more complicated calculation for δ_r. As the local reduction of each stage must be computed at the network leaves, the length of the path taken by the operands through the network will change; for example, the operands for the first stage of the reduction could pass through one switch to a neighbouring node, but the operands for the last stage of the reduction must pass through the top of the tree (and get back to a leaf node). Due to the latency cost of getting information from the NIC on to the sockets, we assume that the actual reduction operation itself



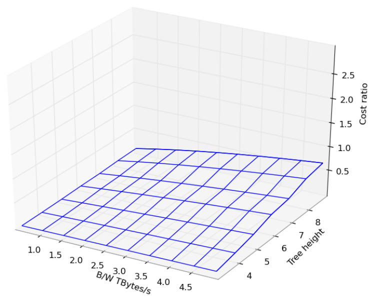
GPN = 1, TFV ≈ 48 bytes



GPN = 50^3, TFV ≈ 117 KB



GPN = 100^3, TFV ≈ 468 KB



GPN = 200^3, TFV ≈ 1875 KB

Fig. 2. δ_r/δ_s for **fat-tree**. The x-axis is router bandwidth (0.5 to 5 TBytes/s), y-axis is increasing fat-tree height (i.e. the different heights given by the radices in the range 500 to 5), and z-axis is the resulting δ_r/δ_s. GPN is Grid Points per Node, TFV is Total Face data Volume (i.e. per face volume $\times 6$).

is done by an assist processor on the NIC to avoid this; for a current example of such an approach, see [12]. We assume that the final broadcast is supported by the switches, so that the cost for it (β) is the same latency as a trip up and down the tree network. Note that the lack of acceleration means that it is not a good fit for the reduction, which must use multiple trips through the tree.

In Fig. 2 we give plots of the relative cost of the two operations, where the parameters are fat-tree height and router bandwidth. The calculations for the fat-tree bear some similarity to the *mesh + ART* results. However, the ratio values tend to be less. For 200^3, roughly half the space is < 0.25, and thus unlikely to profit much from pipelining. Similarly, the lower tree heights, which are more likely for a low diameter network, are quite often under 0.25 for 100^3. However, 50^3 problem sizes will almost definitely benefit from pipelining. The increase in reduction cost for lower router bandwidth for size 1 is a quirk that probably results from a contested fat-tree down-link to the nodes that perform the reduction calculations at each step.

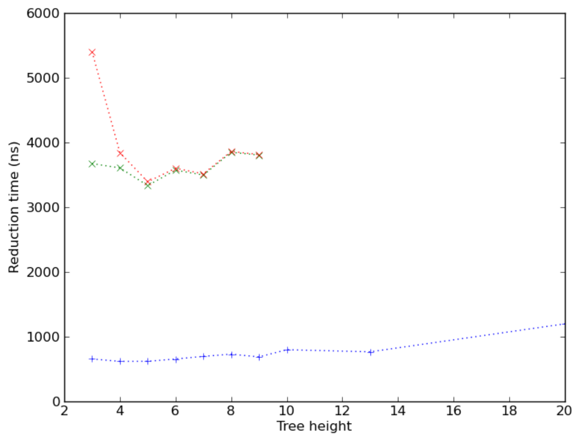


Fig. 3. Reduction times. x-axis is tree network height, y-axis is total all-reduce time in nanoseconds. The bottom line (blue) is for the **mesh + ART**, the top two lines (green and red) are for the minimum and maximum **fat-tree** reduction times respectively (as the radix varies).

5 Reduction Times, Bandwidth and Pipeline Depth

The reduction times computed from the model are given in Fig. 3. The per link one-directional bandwidth varies from 155 GBits/s to 1.55 TBits/s for the torus network as the router chip bandwidth changes, and from 18 GBits/s to 3.7 TBits/s for the fat-tree as both the router bandwidth and tree radix changes.

Measured reduction times for large clusters are difficult to find in the literature. [6] states that a barrier on a full size machine is expected to take about 6μs. It is reasonable to suppose that a reduction will also take roughly this amount of time. Our model gives results which are less that this due to significantly lower assumed latency in the routers, use of tree networks reducing the number of hops, and the fact that the barrier time in [6] is somewhat larger than the value expected from extrapolating directly from their per-hop reduction costs. If our predictions are overly optimistic, then larger values for reduction times at exascale will make scalar products more expensive relative to stencil operations, and deeper pipelines will be necessary.

The required pipeline depths for methods with two reductions per matrix product are (the ceiling of) twice the ratios given in the result plots. Thus, a pipeline depth of at least 1 should be applied to everything except the low bandwidth/low tree height parts of 200^3 for the *mesh + ART*, and many parameter points for the 100^3 and 50^3 problem sizes would benefit from pipeline depths of 3 or 4. Depths are somewhat less for the fat-tree, but depths 1 and 2 are still common, and reasonable parts of the parameter space benefit from 3.

6 Off-node vs. On-node Costs

We consider on-node costs for a stencil or an iteration of a KSM as the floating point operations (flop) and transfers from memory that are required. Instruction bandwidth is unlikely to be important as the algorithms are concise. The amount of non-floating point operations and their throughput is heavily dependant on code implementation details and the features of the core micro-architecture (such as how many integer pipelines there are and how fast they run compared to the floating point vector unit etc), and thus is difficult to include in the model in an unbiased way; thus we omit them.

Rather than very specific predictions of on-node resources, we consider here the general scaling of on-node costs and how technology advancement may affect this in the future. In our example problem, each node is allocated a cube of data to represent its portion of the grid, which is then further subdivided over the different sockets and cores. In the language of the KSMs, each such grid is a (local part of a) *vector*, and the basic operations manipulate these vectors. First we consider when off-node costs dominate for a stencil as a function of the size of the local grid, then we extend this reasoning to the KSMs.

6.1 Stencil Costs

Computational Bandwidth Given the latencies of the reduction operations calculated from the model, the first comparison is against the computational throughput on the node. Estimates of node performance for exascale vary. Here we take the “fat” node estimate from [13], which predicts 10 TFlop/s per node.

The number of flops required per element of a centred difference stencil operation is $2n + k$, where n is the dimensionality of the hypercube and k is some fixed number to take into account the centre point and any normalisation etc.; $n = 3$ in our running example, and we take $k = 3$ to allow for normalisation. For a local cube of side length g, this gives a total number of flops as $9g^3$. This number grows rapidly as g increases. Similarly, the on-node cost for the local part of a reduction is $2g^3$.

The node is not computationally limited provided that the flop rate is sufficient to process the appropriate local cube size in the time taken to perform the corresponding communication. In our example, this means executing $(9 + 2)g^3$ flops in the time taken to perform an off-node all-reduce, when the all-reduce is the longest of the communications. Compute times for the various problem sizes are given in table 2. The figures suggest that the computational bandwidth will start to be a bottleneck shortly before $g = 100$ for the *mesh + ART* and shortly afterwards for the fat-tree topology. We are wary of making any hard claims here due to the fairly wide error margins on the calculations from the model and the predictions of effective future computational bandwidth. Nonetheless is reasonable to suppose that pipelining is likely to be useful for $g \leq 100$, which is also what the δ_r/δ_s ratios show. Due to the cube growth of flops required with g, it is unlikely that $g = 200$ will not be computational-bandwidth bound.

Memory Bandwidth Computing a stencil requires $2n + 1$ reads and 1 writes per element. If we assume modest cache capacities that can capture the reuse of the values between different element read accesses (which is reasonable for low n), then this becomes 1 read and 1 write to main memory per element. An inner-product requires 1 read and 1 write per element. Thus, the number of off-chip memory transfers to compute a local stencil and scalar product when the vector objects do not fit entirely in cache is $4g^3$.

Although the number of transfers from memory required to compute a stencil is less than the number of flops needed, the factor of difference is small; only 2.75. By contrast, the flop to bandwidth ratio in current and predicted future machines is much larger than this. The ratio of computational bandwidth to memory bandwidth given in [13] is 20:1. The implication of this trend is that a problem size that is mostly out-of-cache will be much sooner memory bandwidth bound than computational bandwidth bound. Thus, the problem should fit (almost) entirely in-cache for pipelining to be relevant, or future memory bandwidth would have to be substantially larger than current predictions.

Table 2. Data sizes and local compute times as local cube side length g varies

g	Volume per vector	Total volume GMRES(10)	Compute stencil + reduce	Compute GMRES
1	8 bytes	88 bytes	$< 10ns$	$< 10ns$
50	976.5 kB	10.4 MB	137ns	662ns
100	7812.5 kB	83.9 MB	1100ns	5300ns
200	62500 kB	671.3 MB	8800ns	42.4μs

6.2 KSM Costs

To extend this sort of calculation to a whole KSM algorithm, we need to take into account the change in algorithm resource requirements. A prototypical long recurrence algorithm, truncated GMRES(m), requires $m + 1$ vectors of storage, where m is chosen to balance rough estimates of iteration cost vs. the number of iterations needed for convergence. GMRES(m) requires $2m + 8$ memory transfers and $2n + 4m + 7$ flops per grid element. We show an analysis for GMRES(10). Note that GMRES requires two scalar products, so we are comparing against $2\times$ the reduction costs given in Fig. 3.

Computational Bandwidth The local GMRES(10) computation times are given in table 2. This suggests that, for this recurrence length, pipelining will be interesting for $g \leq 60$ for the *mesh + ART*. As the recurrence length increases, this number will drop, but relatively slowly as the recurrence gives a linear increase in resource requirements. For the fat-tree, pipelining is relevant for $g \approx 100$ for the fastest predicted reduction time. The same caveat for the reduction times derived from the model applies here; a relatively small error in the model could make $g = 100$ a candidate for pipelining GMRES, but the estimates would have to be an order of magnitude off for it to be worthwhile for $g = 200$.

Memory Bandwidth The ratio of required computational bandwidth to data bandwidth for GMRES(10) is 53:28 ($\approx 1.8 : 1$), which is worse than the ratio required for the simple stencil test. Thus, GMRES is even more prone to being memory bandwidth limited. Added to this, the total storage required is $m + 1$ vectors, thus for GMRES(10) the cache should be $100^3 \times 11 \approx 83$MB. Although this is somewhat out of the range of current standard nodes (e.g. 4 sockets $\times$ 10MB L3 cache per socket), a doubling of L3 cache size would be enough to keep virtually all the problem in cache. Thus, $g = 100$ could well still benefit from pipelining on exascale machines if cache capacities increase modestly. However, caches would have to grow by a factor of 15 to fit $200^3 \times 11 \approx 671$MB; this is less likely, but not impossible if capacity follows Moore's law.

Relevant recurrence lengths for GMRES are clearly quite small due to cache capacities when considering pipelining. Pipelining is mostly targeted at short recurrence KSMs though, as they suffer the worst from strong scaling parallel inefficiency.

7 Technology Factors

It is very difficult to predict exactly what the machines will look like at exascale and afterwards due to the jump effects of the introduction of different technologies. The ratios of the resources may be significantly different. For example, the computation to DRAM bandwidth ratio may change in the medium term due to 3D stacking of DRAM and CPU, and/or optical interconnects [16,15]. Any relative increase in DRAM bandwidth will close the gap somewhat between computational and memory bandwidth, and may make pipelining relevant for problem sizes that are currently memory bandwidth limited due to limited cache capacity.

Note that implementing pipelined KSMs requires simultaneous execution of communicating parts of the algorithm that occur together, and thus they must be multiplexed onto the network resources. At the very least it must be possible to execute a stencil whilst waiting for a reduction to complete. We omit a discussion on these details due to lack of space, and refer to [10] for ongoing work on asynchronous reductions in e.g. MPI.

8 Related Work

8.1 Rescheduling

There have been various projects looking at how to combine and schedule basic KSM operations, without altering the dependency structure of the algorithms themselves, and/or the resulting performance; some examples include [5], which considers rescheduling for bandwidth reduction, and [14], which uses careful ordering of the operations of variants of the two-sided KSMs to allow scalar products to be executed at the same time as one of the matrix-vector products; this amounts to a partial pipelining approach. Our work is differs as we consider the future impact of an algorithm that does more extensive reordering.

8.2 Partial Pipelining of Gram-Schmidt Orthogonalization

In [9] the authors improve the scalar product latency tolerance of two iteration Iterated Classical Gram-Schmidt (ICGS(2)) by doing the first iteration of orthogonalization and normalisation as usual, and then launching the second iteration in parallel with the creation of the next basis vector. They report moderate speed-up improvements over standard ICGS, but do not attempt to extrapolate to future computing technology. The context of the work is KSM–based eigensolvers.

9 Conclusions and Future Work

This paper has given the motivation for the study of pipelined KSMs [7] based on a performance model of exascale machines. We have shown how the lack of parallelism could affect the performance of KSM algorithms mapped onto different parallel architectures for smaller problem sizes that occur as a result of strong scaling or use of multigrid. We have also given estimates of the extent of pipelining that will be needed; depths of 1 are likely to be common, and depths up to 4 could easily be needed.

Our work can be expanded in a number of ways. A missing component of our model is OS noise [11] and synchronisation jitter, which would have the effect of increasing the cost of all-reduce operations and make δ_r/δ_s larger, and pipelines thus longer. Secondly, although it is impossible to validate our model (as exascale hardware has not been built yet), we plan to adjust our model parameters to reflect current hardware to see whether pipelining could already be used. Finally, the scheduling freedom introduced to tolerate all-reduce latencies in pipelined KSMs could also be used to improve the temporal locality of access to vector entries. This may have an important impact for larger, memory bandwidth limited problems even when reduction latency itself is not problematic. We will investigate the performance of the algorithms for this size of problem in future work.

10 Acknowledgements

This work is funded by Intel and by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT). Thanks to Karl Meerbergen for his input.

References

1. Retrieved from MVAPICH2 website (2012). `http://mvapich.cse.ohio-state.edu/performance/interNode.shtml`, 2011.
2. N. Adiga et al. An overview of the BlueGene/L supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference*, page 60, nov. 2002.
3. Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6D mesh/torus interconnect for exascale computers. *Computer*, 42(11):36 –40, nov. 2009.
4. B. Arimilli et al. The PERCS high-performance interconnect. In *IEEE HOTI 2010*, pages 75 –82, Aug. 2010.
5. T. J. Ashby and M. F. P. O'Boyle. Iterative collective loop fusion. In *ETAPS CC 2006*, pages 202–216, March 2006.
6. D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM BlueGene/Q interconnection network and message unit. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 26:1–26:10, New York, NY, USA, 2011. ACM.
7. P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. To be published 2012.
8. B. Gmeiner, T. Gradl, H. Kstler, and U. Rde. Analysis of a flat highly parallel geometric multigrid algorithm for hierarchical hybrid grids. Technical report, Dept. Comp. Sci., Universitt Erlangen-Nrnberg, 2011.
9. V. Hernndez, J. E. Romn, and A. Toms. A parallel variant of the Gram-Schmidt process with reorthogonalization. In *PARCO*, pages 221–228, 2005.
10. T. Hoefler and A. Lumsdaine. Overlapping Communication and Computation with High Level Communication Routines. In *Proceedings of the 8th IEEE Symposium on Cluster Computing and the Grid (CCGrid 2008)*, May 2008.
11. T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *ACM/IEEE Supercomputing 2010*, pages 1–11, 2010.
12. A. Moody, J. Fernandez, F. Petrini, and D. K. Panda. Scalable NIC-based reduction on large-scale clusters. In *ACM/IEEE Supercomputing 2003*, pages 59–, 2003.
13. R. Stevens, A. White, et al. Architectures and technology for extreme scale computing. Technical report, ASCR Scientic Grand Challenges Workshop Series, December 2009.
14. L. Tianruo Yang and R. Brent. The improved Krylov subspace methods for large and sparse linear systems on bulk synchronous parallel architectures. In *IEEE IPDPS 2003*, page 11 pp., april 2003.
15. A. N. Udipi, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. Combining memory and a controller with photonics through 3D-stacking to enable scalable and energy-efficient systems. In *ISCA 2011*, pages 425–436, 2011.
16. D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee. An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth. In *IEEE HPCA 2010*, pages 1 –12, jan. 2010.