A framework for concurrency in numerical simulations using lock free data structures: GraPA

Peter Klein, Dimo Maleshkov
Fraunhofer ITWM
Fraunhoferplatz 1
D-67663 Kaiserslautern, Germany
klein@itwm.fraunhofer.de, maleshk@itwm.fraunhofer.de

Abstract

The development of numerical simulation software tools for the solution of real world problems usually calls for domain experts in modeling. The GraPA framework, as an abstraction layer on top of hardware characteristics, supports modelers in two respects: one is the built in support for code co-processing and the other is the generically delivered high performance achieved by "optimally" using concurrency features of multicore and distributed memory architectures behind the interfaces of GraPA. Technically, GraPA is implemented as a C++ template framework, where the modelers data structures and algorithms instantiate the template framework. Using this approach, we handle threads and message passing entirely within the framework. In this paper, we report on the status of the implementation of GraPA and on its performance characteristics.

1: Introduction

In order to solve for real world problems, for instance automized controllers in production lines or in materials science, usually many models operating on different length and time scales need to be coupled and the CPU times are quite often a critical requirement. Furthermore, the size of the problems to be solved in terms of memory requirements are quite often high enough so that these problems fit in distributed memory only (unless on use slow out-of-core approaches).

But: domain experts in modeling are quite seldom experts on hardware architectures, so their simulation codes are usually not optimal in terms of memory access patterns leading to bad performance characteristics. Moreover, new hardware architectures call for concurrency, a concept which by itself calls for new programming paradigms, see [7], which modelers are nowadays usually not aware of.

A challenge for software architects supporting modelers therefore is: Can we find an *implementable* abstraction of hardware architectures which delivers generically performance, direct support for code co-processing, and provides an easy to use interface for modelers ?

One answer is the GraPA framework (Graph Parallel Architecture), which implements an abstraction of algorithms on graph structures using generic C++ template mechanisms. As template instantiation of the framework, node and link data structures and function objects are necessary which encapsulate the data and algorithms of numerical simulations. The implementation of this abstraction behind the GraPA framework interfaces uses a policy based design to map GraPA to the characteristics of the underlying hardware architectures in order to implement optimized memory access patterns on cache based architectures, threads on multicore architectures, and message passing on distrubuted memory architectures. In this paper, we will report on the basic abstraction mechanisms of GraPA an we will present performance measurements of various applications developed under our prototype GraPA implementation. During the implementation, we instrumented GraPA by the PAPI library ([6]) and by the MPE ([1]) in order track the performance characteristics.

Within this paper, we frequently use graph partitioning for various purposes. The idea behind is that a given graph should be partitioned in a way that the parts contain a balanced number of nodes and that the number of links between partitions are minimized. The current GraPA implementation uses the well known ParMETIS package ([2]) in order to achieve any graph partitioning mentioned in this paper. If not indicated explicitly, all performance results where measured on the hercules cluster of the ITWM. hercules is built on dual Intel Xeon 5148LV ("Woodcrest") nodes, interconnected by an Infiniband network running the MPI, for details see [5].

2: The architecture of GraPA

There are many mathematical models which naturally abstract on a graph. Known examples are finite element, finite volume, and finite difference methods to solve partial differential equations, sparse matrix solvers, particle methods like molecular dynamics and Monte Carlo methods. All these methods and others may be formulated by introducing a basic discretization entity BDE, placing many BDEs in space and connect them usually by operations on neighbors. In this type of formulation, a graph abstraction is quite natural. Simulation tools, once they go for real world applications, quite often have to deal with multi-physics on different length and time scales, so model co-processing or code co-processing is a further requirement for modelers. Simulations of this type usually share another communality: they usually need a lot of resources in term of RAM and CPU time in order to produce results.

GraPA is intended to tackle all these problems. The requirements for GraPA are therefore the following ones:

- Provide plug-and-play mechanisms for concrete node- and link data
- Provide loop functionalities for mathematical algorithms running over the allocated concrete node- and link data.
- Integrate optimal memory access patterns.
- Parallelize on the graph level.

We implemented this requirements in essentially 3 classes as a C++ template framework: (CoCo-)Graph, RegistryInterface, and FrameworkInterface. The (CoCo-)Graph encapsulates the discretized geometry of some domain for instance. On each node of the graph, The modeler has to define a problem color and a unique BDE_Id for each node of the graph. The RegistryInterface has just one templetized member function which gets instantiated by the modelers data types for BDE and Link classes. Internally, RegistryInterface objects store factory objects which allow for the creation of the memory of the users data. These factories need a (CoCo-)Graph in order to generate plain memory by using a hardware specific memory allocation policy. RegistryInterface objects and a (CoCo-)Graph object then initialize the FrameworkInterface which internally uses the factories of the RegistryInterface object and the graph provided. The parallelization on multicore and distributed memory architectures is then handled entirely behind the interface of the FrameworkInterface object and adopts during its configuration (before compilation) to the hardware on which the numerical simulation is supposed to run on. Here, we use graph partitioning in a shell like model: first we partition according to problem colors to get sub-graphs and InterColor-Links, then we use in each sub-graph a partitioning among processors for instance. The sub-graphs are then feeded into the RegistryInterface factory objects in order to allocate the memory for the BDEs and Links. This plain memory gets filled afterwards by calls to the C++ placement new operator using modeler provided constructors in which GraPA feeds the unique BDE_Id and the unique BDE_Id pairs of links. Loop functionalities of the FrameworkInterface then allow for a plug in of the modelers operator objects. The control of time steps and a possible re-initialization of operators is the the modelers responsibility.

In the remaining part of this section, we describe the parallelization on multicore and distributed memory architectures and the L2-cache optimization of GraPA's FrameworkInterface. We describe our shell model from inward to outward, starting with the single core cache based shell for a problem with just one color, since the extension to many problem colors, although rather technical, is straightforward.

2.1: Cache based architectures and operator pipelines

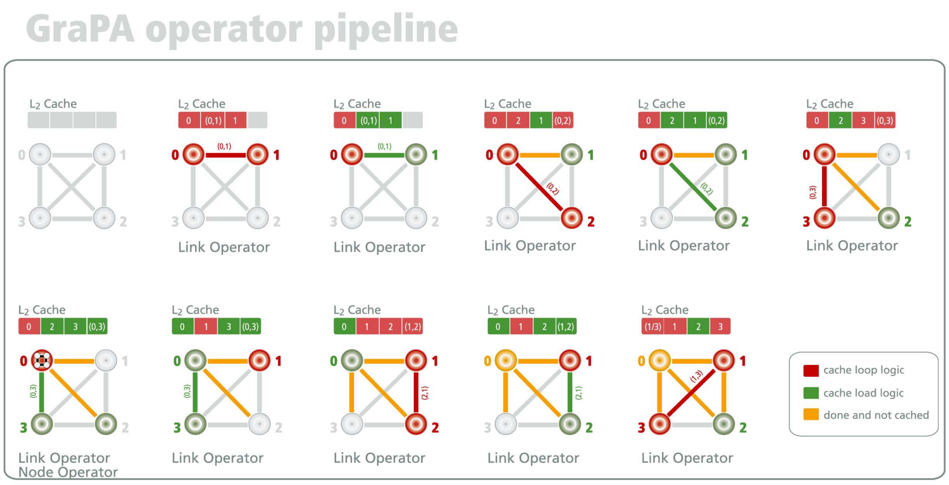


Figure 1. Operator pipeline implemented on L2NodeBlocks, see text.

The idea behind optimizing performance on cache based architectures is to use locality whenever possible. We achieve locality in memory by partitioning the input graph on this GraPA shell in a way that the memory blocks allocated by the RegistryInterface factory objects and stored in the FrameworkInterface object fit into one L2-cache block. These are called L2NodeBlocks. The logic behind temporal locality implemented in GraPA is the illustrated in figure 1. In this figure, we assume a hypothetical system which has just 4 cache lines, managed in a FIFO fashion, of the size of a true L2-cache block.

This type of operator pipelining, or intertwining of Node- and Link-Operators, should lead to favorable cache reuse characteristics. We have instrumented this shell with the PAPI library in order to measure the cache reuse characteristics achieved by our implementation.

2.2: Multicore architectures and lock-free loops

From a performance point of view, multicore architectures share some communalities with distributed memory architectures. On both systems, load balancing and scaling behavior are essential characteristics. Load balancing within the GraPA approach is achieved within the FrameworkInterface object by a graph partitioning, assignment of sub-graphs to cores, and using threads. In each thread, we proceed as in the inner shell described in 2.1 in order to pipeline node and link operators. However, the grap partitioning on this level creates InterThreadLinks. These InterThreadLinks may be ordered in Containers which define links on a coarsened graph, namely a graph with L2NodeBlocks nodes described in 2.1. Loop functionalities have to deal with these InterThreadLinks, so the L2NodeBlocks represent critical sections. A traditional approach using muticees in order to protect the L2NodeBlocks, would lead to a huge amount of kernel entries while locking the muticees and to waiting behavior of threads once another threads holds a lock on a L2NodeBlock.

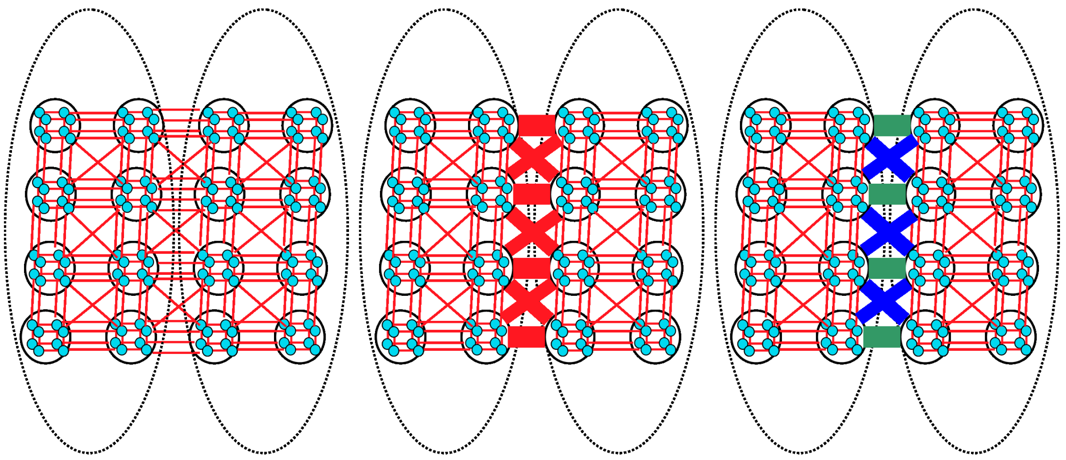


Figure 2. Chromatic ordered InterThreadLink pattern. Left: graph partitioned among threads, within threads the sub-graphs are partitioned in L2-fashion. Middle: fat red links indicate a coarsened graph containing the InterThreadLinks. Right: edge colored InterThreadLinks used in GraPA's lock-free access pattern.

The coarsened graph and our approach to handle this problem is shown in figure 2. We color the edges of the graph in such a fashion that each color entering a node is distinct from all other colors entering the same node. Then, we redistribute the InterThreadLinks over the cores color by color. Within one color, there are by construction no critical sections. Therefore we need no protection of the L2NodeBlocks but just a thread synchronization. The number of the necessary synchronization points is just the number of the edge colors.

Now, the theorem of Vizing tells us, that the minimum number of colors necessary to color all links of the graph is ether n or n+1, where n is the maximal coordination number of a node. n or n+1 respectively is called the chromatic number of the graph. We use an implementation of this theorem within our loop functionalities. This approach has the additional advantage that the maximum chromatic number is bounded from above, independent of the number of threads. therefore, at least in theory, this concept should lead to scaling up to hundreds of cores. This shell is instrumented by both, MPE and PAPI.

2.3: Distributed memory architectures and chromatic ordered message passing

Distributed memory applications using the MPI under the SPMD paradigm are very well discussed in the literature. The general idea behind is to use replica constructs for far away data which get communicated by calls to the MPI. This approach is more ore less general nonsense in the HPC community.

GraPA uses the same approach on the Graph level. First, we partition the input graph in the FrameworkInterface object which leads to sub-graphs for each process and to InterProcLinks. Then, we use the next inner shell described in 2.2. The InterProcLinks are then used in order to allocate the MPI buffers which get automatically communicated by the FrameworkInterface whenever a link operator is feeded in the loop functionalities. The concept of edge coloring is used here again on the level of process graphs. Each compute node defines a node of the graph and links are set up whenever MPI communication between compute nodes is necessary. We then color the links of the MPI graph according to the theorem of Vizing and use the color as ordering principle for the MPI calls. This avoids to a large extent congestion of the network and leads to favorable scaling behavior up to thousands of compute nodes. This has already been demonstrated within the IPACS project, see [4]. We call this pattern chromatic ordered message passing.

2.4: An note on the Cell processors

An early version of the GraPA prototype has been ported on the cell processor, see [**?**]. This was not a big deal: we just took advantage of the formal similarity between L2-Cache on traditional processors and the local store on one SPE of cell processors. In other words, we simply ported the inner shell of GraPA as described in 2.1. SIMD operations were then implemented within the collision operator of the Lattice Bolzmann method discussed in the next section. This version was demonstrated on the SC07 in Dresden, Germany, on a small Playstation3 cluster.

3: The IPACS benchmark ported to GraPA

3.1: Brief description of the IPACS benchmark

Within the IPACS project, a benchmark of the so-called Lattice Boltzmann method of fluid dynamics was developed and distributed as an open source package, see [4]. We reimplemented this benchmark as a GraPA application.

The specific Lattice Boltzmann method used in IPACS is a particle like method which solves directly a lattice version of the Boltzmann transport equation of gas dynamics using a cartesian lattice with unit lattice spacing. The governing equations of this method in the so called BGK formulation are:

$$N_i(\vec{r}+\vec{C}_i, t+1) = N_i(\vec{r},t) + \lambda(N_i(\vec{r},t) - N_i^{eq.}(\vec{r},t)) \; , \qquad i \in \{0, \ldots, b_m\} \; . \tag{1}$$

N_i represent the velocity discrete values of the Boltzmann distribution function along the velocities $\vec{C}_i$, and we use the so called D3Q15 velocity model. $N_i^{eq.}$ represents the discrete velocity version of the Boltzmann equilibrium distribution function. The first term on the right hand side describes a free streaming of the distribution function, the second part is the BGK approximation of the collision operator. One can than show, that the evolution of the moments solves the incompressible Navier-Stokes equation of fluid dynamics, see [3] for instance.

An implementation of this method using GraPA requires the following steps:

- Set up a Graph for the D3Q15 model.
- Associate the velocity discrete distribution function with the nodes of the Graph (class point, the BDE of the Lattice Boltzmann equation).
- Implement the BGK collision operator on one BDE.
- Implement the streaming operator as an operator on one Graph link.
- Instantiate the GraPA template functions of various GraPA objects.
- Instantiate and use the template loop functions of the GraPA framework.

Apart from error handling, I/O, and user interface issues, the following code snip illustrates the usage of GraPA for LBE simulations:

```
int main( int argc, char** argv )
{
   // D3Q15Graph -> Graph:
  D3Q15Graph G();

   // Interface to the factories
  RegistryInterface R;

   // feed the LBE types
  R.register_Graph<point,Link>();

   // Init the GraPA framework
  FrameworkInterface C( R,G );

   // The LBE operators
  Collision Col;
  Propagation Prop;

   // pipelined time loop
  int T = atoi(argv[2]);
  for( int i=0; i<T; i++ )
    C.pipeline( Col, Prop );

   return 0;
};
```

A complete problem of computational fluid dynamics requires usually boundary conditions and external forces (gravity). Usually, boundary conditions need special data like distances from true boundaries, slip fractions, and others. Their integration in the GraPA framework requires some additional steps:

- Derive boundary BDEs from the basic BDE, the point class.
- Derive link classes accordingly.
- Implement the additional features as node- or link operators.

- Use the CoCoGraph, color the additional features using different problem colors.
- Instantiate the GraPA template functions of various GraPA objects using the new class definitions.
- Instantiate and use the template loop functions of the GraPA framework with the new operators.

In the sense of GraPA,i.e. boundary conditions are therefore treated as new codes.

3.2: Benchmark results and discussion

The IPACS setup of the lattice Bolzmann grid is discussed in detail in ref. [4]. As basic Lattice Bolzmann grid, we use 105 points in each direction. Within this grid, according to the IPACS benchmark, we set obstacles and run the benchmark code in order to measure the permeability of this system. This grid is then scaled so that the number of grid points is proportional to the number of cores used in the simulation.

On a single node, we measure the speed-up of the GraPA based benchmark code since on a single node with shared memory, this is the measure of interest. In figure 3, we compare a single hercules cluster node with an 8 core Intel Xeon E5420 (2.5 GHz). The

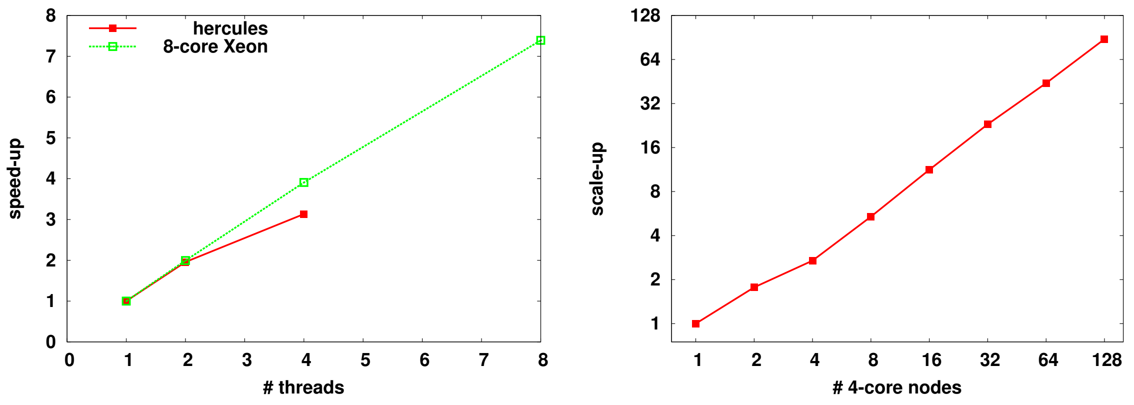
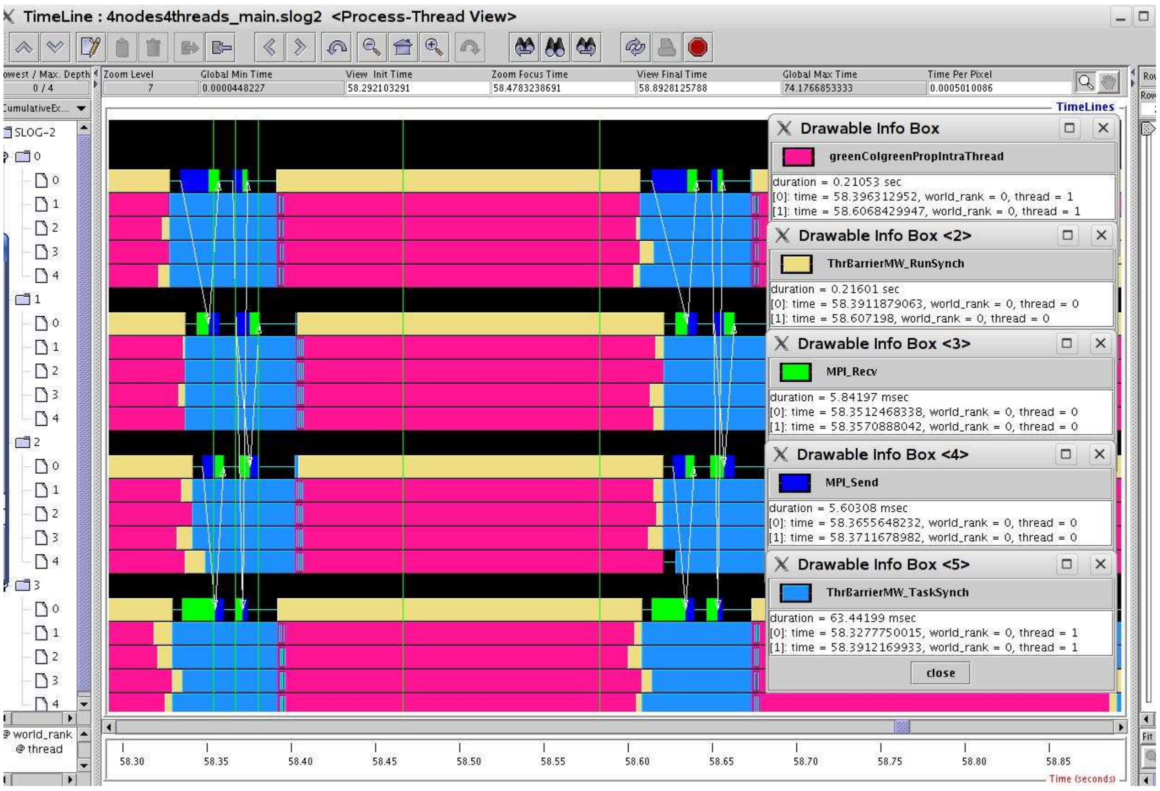


Figure 3. Speed-up and scale-up of the GraPA based IPACS benchmark.

difference between the scaling behavior on the hercules woodcrest architecutre as compared to the Xeon is under investigation, we currently have no explanation for this difference. Nevertheless, the speed-up is on both architectures near to ideal. The PAPI instrumentation measure cache misses as they are expected by us: The InterThreadLink and InterL2Link updates cause around 60% L2-cache misses and the pipeline of the link- and node-updates cause around 30% L2-cache misses.

The scaling of the IPACS benchmark on the hercules cluster, see figure 3, shows a very satisfactory scaling behavior. Already at 4 nodes, we see saturation of the scaling: from 4 to 128 nodes we found perfect scaling. The MPE analysis of the the 4-node, 4-thread run shows us that the MPI overhead causes the down grade of the scaling behavior. On the other hand, it is well seen that the InterThreadLink loop updates cause no performance problem at all since these updates, the tiny structures between the MPI calls and the IntraThread pipeline are nearly invisible on the time axis.



```
Zoom Level   Global Min Time   View Init Time   Zoom Focus Time   View Final Time   Global Max Time   Time Per Pixel
7            0.0000448227      58.292103291     58.4783238691     58.8928125788     74.1766853333     0.0005010086
```

```
Drawable Info Box
greenColgreenPropIntraThread
duration = 0.21053 sec
[0]: time = 58.396312952, world_rank = 0, thread = 1
[1]: time = 58.6068429947, world_rank = 0, thread = 1

Drawable Info Box <2>
ThrBarrierMW_RunSynch
duration = 0.21601 sec
[0]: time = 58.3911879063, world_rank = 0, thread = 0
[1]: time = 58.607198, world_rank = 0, thread = 0

Drawable Info Box <3>
MPI_Recv
duration = 5.84197 msec
[0]: time = 58.3512468338, world_rank = 0, thread = 0
[1]: time = 58.3570888042, world_rank = 0, thread = 0

Drawable Info Box <4>
MPI_Send
duration = 5.60308 msec
[0]: time = 58.3655648232, world_rank = 0, thread = 0
[1]: time = 58.3711678982, world_rank = 0, thread = 0

Drawable Info Box <5>
ThrBarrierMW_TaskSynch
duration = 63.44199 msec
[0]: time = 58.3277750015, world_rank = 0, thread = 1
[1]: time = 58.3912169933, world_rank = 0, thread = 1
```

Figure 4. MPE analysis of the 4-node, 4-thread IPACS run.

4: Linear algebra and GraPA

Linear algebra solvers play an important role in many simulations of partial differential equations, since implicit and semi-implicit methods are from a mathematical point of view the methods of choice for the solution of the discretized versions of partial differential equations. Here, we report on our first attempts to implement sparse linear algebra solvers under GraPA. This should be not that difficult since the well known CRS format of a sparse matrix is essentially a graph format. As a model example, we use the heat equation in 3 dimensions

$$\partial_t T = \kappa \partial_x^2 T, \tag{2}$$

solved for a quadratic rod with variable length in x-direction. At the planes $x = 0$ and $x = x'_{max}$ we use constant temperature, and isolating boundary conditions on all other faces of the rod.

We choose the conjugate gradient method in order to solve the transient heat equation, discretized by cubical finite volumes, fully implicit in time. The implementation of this solver as a GraPA modul uses as BDE class a structure containing one element of the temperature-, old temperature-, residual-, and search-vector of a traditional matrix-vector based implementation, the links carry the off diagonal elements of the discretization matrix. We treat the complete system in the sense of GraPA as a 3 color problem due to the boundary conditions used, with obvious extensions for the boundary voxel BDE‘s. The solution algorithm is a straight forward implementation of text book conjugate gradient methods.

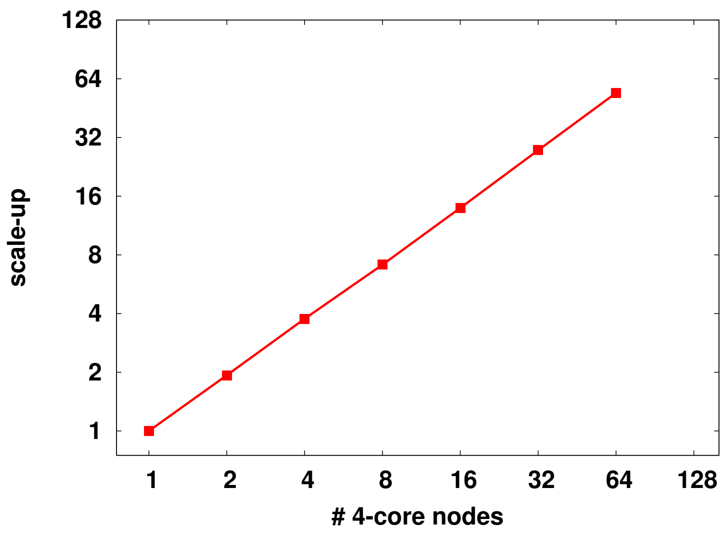


Figure 5. Scale-up of the CG heat equation solver on the hercules cluster.

4.1: Scaling results for the Conjugate Gradient method

As a scaling of the problem size, we simply scaled the physical length of the rod between the constant temperature plates at constant grid resolution. Using this approach, the number of unknowns scales with the number of cluster nodes.

The scaling results measured on hercules are shown in figure 5. The nearly perfect scaling is entirely due to GraPA since, to stress this point once again, all parallelization is behind the interface of the GraPA framework. Although we did not go for a comparison with other implementations of the conjugate gradient method, the scaling results achieved give us the hope to implement successfully more linear algebra methods with similar scaling properties on clusters with multicore nodes.

5: Conclusion and outlook

In an ever faster changing world of technology, flexibility for fast adaption of numerical simulations to technological questions is a key to faster development cycles of materials and devices for instance. On the other hand, numerical simulations share a big enough amount of commonalities among each other to make it worth implementing these commonalities in a single, portable framework like GraPA. The performance and scaling characteristics of the GraPA based applications discussed in this paper on multicore and/or distributed memory architectures turned out to be very satisfactory, independent of the numerical algorithm used and independent of the hardware characteristics. Theoretically, the lock-free pattern of chromatic ordered InterThreadLink accesses developed in this paper will be key for scaling on the “many core architectures” discussed in todays announcements of vendors.

Although, this paper describes work in progress and not all features on the wish list of modelers are currently supported, a clear picture emerges from these attempts: supporting concurrency by frameworks is difficult, but worth to be done since modelers usually need support in order to take advantage of new hardware architectures.

During the implementation of the GraPA prototype, we found it absolutely necessary to instrument our code by PAPI and MPE in order to track down performance bottlenecks. Timing the various shells of our implementation alone is a too crude measure for taking design decisions which lead to improved performance characteristics. On the other hand, the instrumentation used will allow for further optimizations of the GraPA Framework on the one hand. On the other hand, porting GraPA to new hardware systems will take advantage of the already in-cooperated instrumentation libraries. In particular, we will go for asynchronous message passing and we will consider optimizing L1 cache reuse by pipelined operators.

6: Acknowledgment

The porting of GraPA to the Cell architecture has been part of a contract research with IBM, Germany. We acknowledge discussions with Michael Schröder, ITWM, on the subject of thread scheduling algorithms. The final design and the prototype implementation of GraPA where done under the THOKA project, funded by the Fraunhofer association. One of us, Dimitar Asenov, acknowledges an internship contract at the ITWM under which his work on this paper was done.

We gratefully acknowledge the opportunity to use a beta version of the mpe2-1.0.7 library, which has been made available to us by Anthony Chan from ANL upon our request in February 2008.

References

[1] A. Chan and B. Gropp and R. Lusk. MPI Parallel Environment MPE. http://ftp.mcs.anl.gov/pub/mpi/mpe/beta .

[2] G. Karypis and K. Schloegel and V. Kumar. Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.1. http://glaros.dtc.umn.edu/gkhome/views/metis/ .

[3] L. Giraud, D. d'Humières, and P. Lallemand. A lattice boltzmann model for visco-elasticity. *Int. J. Modern Physics*, C 8:806, 1997.

[4] Integrated Performance Analysis of Computer Systems. fundet by the BMBF. http://www.ipacs-benchmark.org/ .

[5] ITWM. Fraunhofer ITWM Cluste homepage . http://cluster.itwm.fhg.de .

[6] J. Dongarra et. al. Performance API. http://icl.cs.utk.edu/papi .

[7] Herb Sutter. Will concurrency be the next revolution in software development ? *C/C++ Users Journal*, 23, Nr. 2:46, 2005.