Towards High-Level Grid Programming and Load-Balancing: A Barnes-Hut Case Study

Martin Alt, Jens Müller, and Sergei Gorlatch

Westfälische Wilhelms-Universität Münster, Germany
{mnalt,jmueller,gorlatch}@uni-muenster.de

Abstract. We propose a high-level approach to grid application programming, based on generic components (*skeletons*) with prepackaged parallel and distributed implementations and integrated load-balancing mechanisms. We present an experimental Java-based programming system with skeletons and use it on a non-trivial, dynamic application – the Barnes-Hut algorithm for N-body simulation. The proposed approach hides from the application programmer many complex details of grid programming and load-balancing, and demonstrates good performance on an experimental grid testbed.

1 Introduction

Grid programmers are faced with the challenge of developing applications that can run distributed across several heterogeneous hosts. Programs must use grid resources efficiently and incorporate flexible load-balancing strategies in order to distribute tasks among hosts that are not known at compile time. The success of grid technology will depend on creating suitable programming models and middleware to liberate application programmers from the complex and low-level details that have to be taken into account during grid software development.

In this paper, we argue for a high-level approach to programming grids, which combines application program development and load-balancing strategies. We present an implementation of the approach as an experimental Java-based programming system that provides application programmers with a set of high-level, reusable components, called *skeletons*, which are customisable for particular applications by means of data and code parameters. We demonstrate the use of our system on a non-trivial case study with a complex, dynamic behaviour – the Barnes-Hut (BH) algorithm for N-body simulation. We show how the use of high-level components hides from the application programmer most of the complexity of distributing computations over the grid and how the load-balancing mechanisms incorporated in our system can evenly balance work between heterogeneous grid servers. We conclude the paper by reporting experimental results for the BH algorithm on a grid-like testbed and by discussing related work.

2 Programing and Load-Balancing with Skeletons

In our programming model, each grid server provides a set of generic algorithmic components called skeletons. When a program is executed on a client, calls to

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 391–400, 2005.

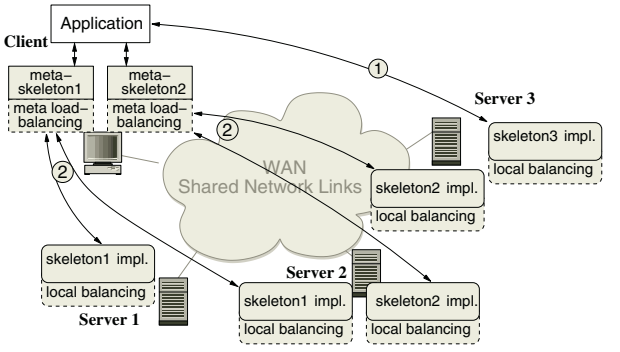


Fig. 1. Application using three skeletons executed in the Grid.

resource intensive skeletons are delegated to the servers as shown in Fig. 1. The client program either calls skeletons directly (①) or uses *meta-skeletons* which combine several skeletons of the same type running on different servers and present themselves as a single skeleton to the client (②). While each skeleton comes with a prepackaged efficient implementation on a server, meta-skeletons are implemented locally on the client; they distribute computations and perform load-balancing, coordination and monitoring of the distributed execution.

Due to space constraints, we limit our discussion to programming and load-balancing issues, and omit details about resource management in our system.

2.1 The Skeleton-Based System and Its Implementation

We present here a basic repository of skeletons, which will be used for implementing the Barnes-Hut case study, using a functional notation:

Map: Apply a unary function f to all elements of a list:
$map(f, [x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)]$

Sort: Sort all elements of an input list according to a given order $\prec$:
$sort(\prec, [x_1, \ldots, x_n]) = [x_{i_1}, \ldots, x_{i_n}]$ where $x_{i_j} \prec x_{i_k}$ for all $j < k$

Reduce: Compute the "sum" of a list using a binary associative operator $\oplus$:
$reduce(\oplus, [x_1, \ldots, x_n]) = [x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_n]$

Apply: Applies a unary function f to a parameter x: $apply(f, x) = f(x)$. The *apply* skeleton is used to remotely execute a function f on a server.

In addition to these data-parallel skeletons, the system also provides personalised all-to-all communication facilities to transfer data directly between servers.

Our skeleton-based grid programming system shown in Fig. 1 is implemented on top of Java and RMI, mostly for reasons of portability (see [1] for "10 reasons to use Java in Grid computing"). Skeletons are offered as Java (remote) interfaces, implemented in an architecture-specific way on different servers. All skeletons operate on single objects or arrays which can be distributed blockwise

over several servers. For each skeleton, the system provides an interface which is implemented both by client-based meta-skeletons and on the servers.

To avoid unnecessary remote communication, we developed a special mechanism, "future-based RMI" [2]: instead of sending the actual data, skeletons work as much as possible with *remote references*, which are small pointers to the actual data on the servers. Skeletons are executed asynchronously, returning such a remote reference immediately when called. This remote reference can be passed to the next server, while the first server is still executing. Upon completion of the first skeleton, the actual data is sent directly to the second server.

2.2 Integrating Load-Balancing into Skeletons

On the grid, it is critical to distribute data and computations efficiently across the potentially heterogeneous hosts. In our approach, we integrate load-balancing policies into skeleton and meta-skeleton implementations, thereby hiding the complexity of load-balancing from the application programmer. Load-balancing is realised on two levels in the system: (1) each skeleton running on a single parallel server balances the load between the processors of the server, and (2) the meta-skeleton is responsible for distributing data and balancing work between different servers. On both levels, skeleton-specific load-balancing strategies are used, which distribute data based on the skeleton structure. On the server level, implementation and architectural aspects can also be taken into account.

3 Case Study: The Barnes-Hut Algorithm

The *Barnes-Hut (BH)* algorithm [3] is a widely used approach to computing force interactions of bodies (particles) based on their mass and position in space, e.g. in astrophysical simulations. At each timestep, the pairwise interactions of all bodies have to be calculated, which implies a computational complexity of $O(n^2)$ for n bodies. The BH algorithm reduces the complexity to $O(n \cdot logn)$, by grouping distant particles: for a single particle in the BH algorithm, distant groups of particles are considered as a single object if the ratio $boxsize/distance$ is smaller than a simulation-specific coefficient θ chosen by the user (see Fig. 2).

For an efficient access to the huge amount of possible groups in a simulation space with a large number of objects, the BH algorithm subdivides the 3D simulation space using a hierarchical *octree* with eight child cubes for each node

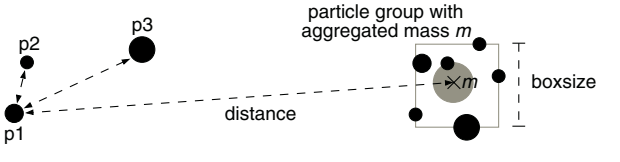
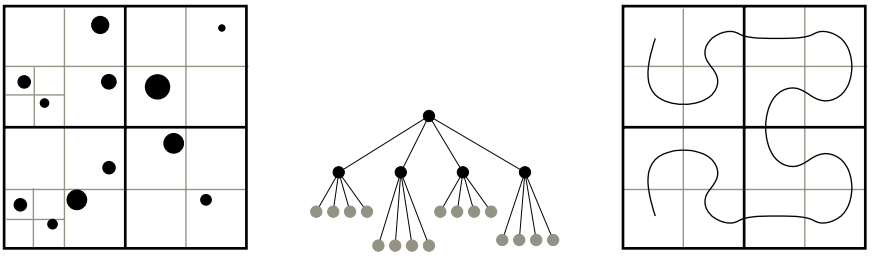


Fig. 2. When calculating forces for p1, particles p2 and p3 have to be considered individually. For a distant particle group, aggregated calculation using m is performed.



(a) Space partitioning (b) Quadtree (c) Peano-Hilbert order

Fig. 3. Barnes-Hut octree partition of the simulation space.

(or *quadtree* for the 2D case). The tree's leaves contain single particles, parental nodes represent the particle group of all child nodes and contain the group's centre and aggregated mass. The force calculation of a single particle then is performed by a depth-first traversal of the tree. Fig. 3(a) and 3(b) depict an example partition and the resulting quadtree for the 2D case (see [3] for further details and cost considerations).

For computing the force interactions in parallel, the particles are distributed among participating hosts. Our implementation uses the *Peano-Hilbert order* (Fig. 3(c)), providing a total linearisation of a two- or three-dimensional space [4]. In the resulting vector of particles, adjacent objects are placed close together. Thus, a blockwise distribution of the particles vector among hosts results in a contiguous particle space assigned to each host. This leads to a reduced amount of communication between hosts, because particles that are close together (and thus need to exchange information often) are likely to be on the same host.

3.1 Barnes-Hut Using Skeletons

The first step in our high-level programming approach is to express the desired application using the skeletons contained in the repository of Section 2.1. First, the particle vector is partitioned into p segments which are distributed among the p servers taking part in the computation. Then, the following five steps are performed iteratively:

```
1     bb = reduce(boundaries, particles);        //bounding box
2     map(PHIndex, particles);                   //Peano-Hilber index
3     sort(LessEqual, particles)                 //sorting
4     tree = reduce(treebuild(bb), particles);   //treebuild
5     map(interact(tree), particles);            //interaction
6     map(particles, update);                    //update
```

1. Calculation of the total boundary of the simulation space: In order to build the tree, the size of the simulation universe is calculated by the reduce operation in line 1. The function `boundaries` compares two particles or a particle and a bounding box, and returns the new bounding box.

2. Sorting: Sorting involves two steps: (1) computing an index for each particle according to the Peano-Hilbert order (using the *map* skeleton in line 2), and (2) sorting the particles vector in ascending order using this index (*sort* in line 3).

3. Building the octrees: For each remote Grid host, the local octree is built in a bottom-up fashion by combining neighbouring particles into trees of depth one. Neighbouring trees are combined into larger trees until a single tree is formed, using a merging algorithm similar to the one described in [6]. Line 4 of the code expresses this process as a reduction skeleton, using the tree merging operation as a parameter, where `bb` is the bounding box computed in the first step.

4. Force computation: The particle interactions are computed using the *map*-skeleton as depicted in line 5. For each particle in `particles`, the operation `interact` traverses the octree `tree` and adds the force effects of the current node to the velocity vector of the particle if $boxsize/distance < \theta$. If this criterion is not yet met, the eight child nodes are processed recursively.

5. Particle update: Line 6 uses a *map* invocation to update the current particle position according to the new velocity vector. For each particle, the unary function `update` adds the velocity, multiplied by the length of time for each iteration, to the current particle's position.

The six-line skeleton code of the BH algorithm has a clear structure, where the details of the parallelisation are hidden in the skeleton implementations. The application programmer is, therefore, liberated from low-level considerations.

3.2 Barnes-Hut on the Grid

The critical problem when bringing the BH algorithm on the grid is that it is infeasible to make the entire particle tree available on every host by replication: this would require each host to send the updated values for its own particles to all other hosts after each iteration, thus increasing communication time. Therefore, a distribution among several hosts requires a more elaborate algorithm.

We adopt a solution similar to [4]: Each host constructs a *locally essential tree* by requesting all tree nodes from other hosts that are relevant to the force computations for its own local particles. The goal is to minimise information exchange between nodes during the interaction phase, by sending all necessary particles between nodes once in advance. Each host executes three steps:

1. Send a description of the sub-space with local particles to all other hosts.
2. For each sub-space received from another host, traverse the local tree recursively and add all nodes matching the $boxsize/distance < \theta$ criterion to the result vector. Send the result vectors to all other hosts.
3. Build the locally essential tree by incorporating the particles and tree nodes received from other hosts.

The skeleton pseudocode for building the locally essential tree is as follows:

```
for each host:
  host.localtree = host.reduce(treebuild, host.particles);
  host.subspace = host.exec(constructSubspace, host.particles);
otherSubspaces = allToAll(subspaces);
for each host:
  relevantParticles = host.map(selectParticles(host.localtree),
                               host.otherSubspaces);
alltrees=personalisedAllToAll(relevantParticles);
tree = reduce(treebuild, alltrees);
```

For the amount of data communicated between hosts in the essential tree building phase, there is a trade-off between the accuracy of space approximation sent to other servers and the remote tree-nodes received in reply. Our approach (whose details we omit due to the lack of space) is to describe local sub-spaces by several boxes of varying sizes, where the number of boxes used for the sub-space approximation can be adjusted before the start of a simulation run.

4 Load-Balancing

Our approach is to aid the application programmer in the task of load-balancing by providing generic load-balancing strategies for skeletons, which can be adapted to the application. In our grid programming system, each single-server skeleton implementation is responsible for distributing work equally among all processors of the server, and meta-skeletons distribute work among all participating servers. Load-balancing within a server depends on the server's hardware and is considered a black-box from the viewpoint of the application and the meta-skeleton.

Load-balancing at the meta-skeleton level can either be done dynamically (e.g., using load-stealing), or statically, i.e., before the skeleton execution is started. For our BH case study, dynamical load-balancing is not feasible because redistributing a particle would also require to redistribute parts of the particle tree. Therefore, we will focus on statical load-balancing throughout the remainder of the paper. Load-Balancing for meta-skeletons is done in two steps: (1) a skeleton specific *load-balancing function* computes an optimal distribution of data according to a predefined load-balancing strategy, and (2) a generic redistribution method is responsible for the actual communication required to distribute the input data according to the new distribution.

In general, load-balancing for grids considers two factors: (1) the amount of work required for an input element (application-specific), and (2) the amount of work that a host can process per second (host-specific). Therefore, our load-balancing functions have both application- and hardware-specific parameters.

As an example, we will discuss the load-balancing strategies implemented for the *map* skeleton (both single-server and meta-skeleton), using a static load-balancing approach, where the load generated by each particle is known in advance.

4.1 Example: Load-Balancing for the Map Skeleton

In order to statically balance the work among processors, it is necessary to assess the load generated by each element of the input list in advance, i.e. we need a *load-prediction function*. For our BH implementation we use the work necessary for the previous iteration as an estimate for the work of the next iteration.

Single-Server Map Skeleton. On a single server with homogeneous processors, the *map* skeleton implementation has to distribute the load equally among all processors. This is done in two steps: in the first step, for each element, the partial sum of the work required for all elements up to the current one is computed (accumulated load). The total work (i.e., the value computed for the last element) is divided by the number of processors to obtain the amount of work to be assigned to each processor (processor share). Then, the elements of the input list are assigned to processors in a blockwise fashion, assigning elements to one processor until the total work required for completing the share assigned to the processor equals the processor share computed in the first step. Thus the first data element assigned to processor p is the first element for which the partial sum of work exceeds the share that should be assigned to processor $p-1$.

Map Meta-skeleton. For computing an efficient load distribution between several grid hosts, we take into account the heterogeneity of the hosts. We introduce a performance factor c_p for each host p, which is proportional to the number of "load-units" that host p can process per second. The amount of work for server p, w_p, is proportional to the performance factor of that server and computed as follows: $w_p = c_p / \sum_i c_i$. The load is then distributed according to the computed distribution $[w_p]$ using the same method as for the single-server case.

4.2 Load-Balancing for Barnes-Hut

The particle interaction phase which computes force interactions for each particle is the most time consuming phase of the BH algorithm: it accounts for well over 90% of the overall runtime. Compared to the interaction phase, all other phases account for only very little time, so that the overhead for balancing load for these phases can be expected to outweigh the gain in performance. Additionally, the locally essential particle-trees constructed for a particular host in the tree-build phase are specific to the particles on that host. Thus, the particles assigned to a particular host need to be the same in the tree-build and the particle interaction phase, impeding load balancing for the tree-build phase. Therefore we only do load-balancing for this phase, which is expressed as a *map* skeleton (line 5 in the code in Sect. 394). Because the distribution of the particles must not be changed between the tree building and the interaction phase, redistribution of the particles is inserted between lines 3 and 4 in the code.

We use *map* skeleton's load-balancing function presented in the previous sections, which requires two parameters: a load predictor which estimates the amount of work necessary for each particle, and a performance factor for each

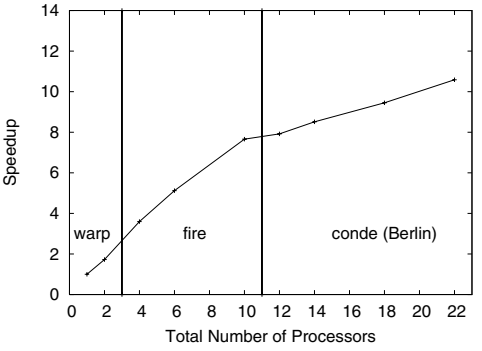


Fig. 4. Relative speedup for three shared-memory servers with a total of 22 processors and $2 \cdot 10^5$ particles ($\theta = 0.25$).

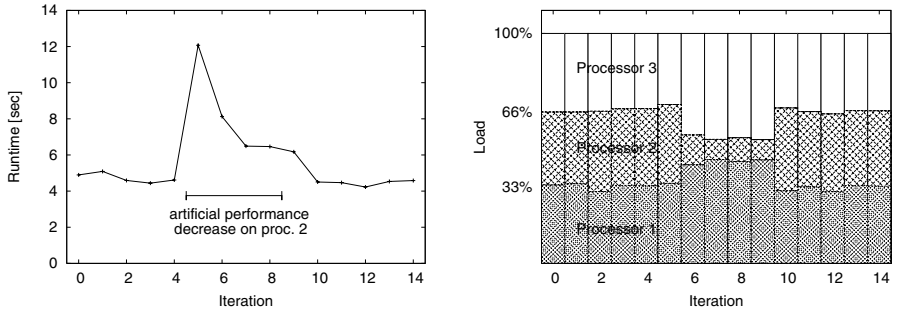
grid host (amount of work per second). For the performance factors, we use the factors obtained a-posteriori from the previous skeleton iteration. For the first iteration, the performance factor of each host is set to the number of processors available on that host. The load predictor for the BH algorithm returns for each particle the number of interactions recorded for that particle during the last interaction phase. For the first iteration, the load predictor returns '1' for each particle, assuming the same amount of work for all particles.

Note that the initial performance factors and load predictions are not very accurate and may lead to a considerable load imbalance. However, load is quickly balanced for later iterations, as demonstrated in our experiments.

5 Experimental Results

We measured the performance of our skeletal Barnes-Hut implementation using three shared-memory servers, two at the University of Muenster ("warp" and "fire"), and one at the Technical University of Berlin ("conde"). Server "warp" has two 2.8GHz Pentium4 processors and "fire" has eight UltraSparc III+ processors running at 1.2GHz, the server in Berlin has 12 900 MHz UltraSparc III+ processors. The client is a workstation with a P4, 2.6 GHz processor in Muenster. All experiments were done using SUN's JDK 1.5.0. Client and servers are connected by two LANs and the german academic internet backbone (WiN). The measured bandwidth within the LAN at Muenster was approx. 3.2MB/s, the bandwidth between Muenster and Berlin (450 km) was measured at 1.1MB/s.

Figure 4 shows the relative speedup obtained for an input size of $2 \cdot 10^5$ particles and $\theta = 0.25$, for varying numbers of processors on different hosts (compared to the performance on one processor of "warp"). The absolute runtimes ranged between 315 s on one processor of "warp" and 26 s on all 22 processors of all hosts for one iteration. Our high-level Barnes-Hut implementation shows very good speedups, also across host-boundaries. Note that the decrease in speedup at 4 and 12 processors is at least partly a result of the slower processors on hosts



(a) Performance over time. (b) Work distribution for (a).

Fig. 5. Load-balancing on three processors with varying load.

"fire" and "conde". Additionally, the speedup decreases between 10 and 12 processors due to the increased communication costs when adding a remote host. However, this cost is amortised for higher numbers of processors. Also note that the particle shares allocated to the two servers in Muenster are neighbouring blocks (according to the Peano-Hilbert ordering), thus reducing the amount of communication between Berlin and Muenster.

To evaluate the load-balancing strategy used for *map*, we measured the performance for 10^4 particles on three Pentium 4, 1.7 GHz PCs connected by LAN ($\theta = 0.25$). The runtimes for each of 15 iterations are shown in Fig. 5(a), and the partition of the particles between the hosts is shown in 5(b). For iterations 5 through 8, we have artificially decreased the performance of processor 2 (starting other time-consuming applications on that host). The figure shows that the work is rebalanced in iteration 6, due to the decreased performance of processor 2. The performance decrease seen in Fig. 5(a) at iteration 5 is thus compensated in the following iterations. After processor 2 is fully available again in iteration 9, work is rebalanced for iteration 10, obtaining the same performance as in the first iterations. This shows that the implemented load-balancing mechanisms are able to adapt to varying performance on different hosts.

6 Related Work and Conclusion

Our work is a step towards designing efficient applications for grids by providing high-level components (skeletons), which hide the complexity of parallelisation and distribution from the application programmer. We have demonstrated how pre-defined generic load-balancing strategies can be integrated with skeletons and then specialised for a particular application.

Our approach differs from other Java-based programming frameworks for grids, such as *ProActive* [7] and *Ibis* [8], because it provides predefined computation patterns with built-in load-balancing strategies. *Satin* [9] also provides load-balancing, but it is limited to divide-and-conquer applications. Another skeleton-based approach, *Lithium* [10], focused mainly on task-parallel skeletons rather than our data-parallel skeletons.

We have shown that high-level components allow to implement relatively complex applications, such as the Barnes-Hut algorithm, hiding the complexity of parallelisation and distribution from the application programmer. The numerous previous BH implementations (e. g. [4–6, 11]), mostly targeted parallel or homogeneous distributed architectures, while our implementation aims to be executed in heterogeneous (grid) systems. Our experiments demonstrated good performance both on one server and across several servers.

Static load-balancing strategies for heterogeneous clusters have been widely studied, and the strategy we used for the *map* skeleton is similar to [12] and [13]. We have shown how static load-balancing strategies can be integrated into skeletons, achieving good performance in a dynamic grid setting.

References

1. Getov, V., von Laszewski, G., Philippsen, M., Foster, I.: Multiparadigm communications in Java for Grid computing. Comm. of the ACM **44** (2001) 118–125
2. Alt, M., Gorlatch, S.: Future-Based RMI: Optimizing compositions of remote method calls on the Grid. Euro-Par 2003. LNCS 2790, Springer (2003) 682–693
3. Barnes, J.E., Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm. Nature **324** (1986) 446–449
4. Grama, A.Y., Kumar, V., Sameh, A.: Scalable parallel formulations of the barnes-hut method for n-body simulations. Supercomputing '94, IEEE (1994) 439–448
5. Blelloch, G.E., Narlikar, G.: A practical comparison of N-body algorithms. In: Parallel Algorithms. American Mathmatical Society (1997)
6. Singh, J.P., Holt, C., Totsuka, T., Gupta, A., Hennessy, J.: Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. Journal of Parallel and Distributed Computing **27** (1995) 118–141
7. ProActive: INRIA (1999) http://www-sop.inria.fr/oasis/ProActive.
8. van Nieuwpoort, R.V., Maassen, J., Hofman, R., Kielmann, T., Bal, H.E.: Ibis: an efficient Java-based Grid programming environment. Proc. of the 2002 joint ACM-ISCOPE conference on Java Grande, ACM Press (2002) 18–27
9. van Nieuwpoort, R.V., Kielmann, T., Bal, H.E.: Efficient load balancing for wide-area divide-and-conquer applications. Proc. Eighth ACM SIGPLAN Symp. PPoPP'01, Snowbird, UT (2001) 34–43
10. Danelutto, M., Teti, P.: Lithium: A structured parallel programming enviroment in Java. Proc. ICCS'02. LNCS 2330, Springer (2002) 844–853
11. Sun, Y., Liang, Z., Wang, C.: Distributed particle simulation method on adaptive collective system. Future Generation Computer Systems **18** (2001) 79–87
12. Crandall, P., Quinn, M.J.: Block data decomposition for data-parallel programming on a heterogeneous workstation network. HPDC. (1993) 42–49
13. Kaddoura, M., Ranka, S., Wang, A.: Array decompositions for nonuniform computational environments. Journal of Parallel and Dist. Comp. **36** (1996) 91–105