Hybrid Systems for N-body Simulations

Spinnato, P.F.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 2

*N-body Codes on Hybrid Architectures*†

In this chapter we analyse NBODY1, the direct particle-particle code introduced in section 1.4.1, and study the performance of this N-body code on hybrid architectures, which were presented in section 1.3.4. A detailed analysis of the N-body code performance, in terms of the relative weight of each task of the code, and how this weight is influenced by software or hardware modifications, is essential to understand the interaction of the code with the hardware platform that executes it. Especially the interaction with the GRAPE, the dedicated device for N-body simulation introduced in section 1.3, requires a careful performance analysis. The use of GRAPE results in a dramatic performance leap for N-body simulations, as it provides a very high performance for the computation of gravity interactions, the most expensive computational task of an N-body code. The interaction of the GRAPE, its general purpose host, and the N-body code run on the machine, gives rise to complex execution patterns that need to be studied and understood to find the optimal configuration. We need this performance analysis in order to acquire the necessary experimental data, for our performance modelling and simulation research to devise a very high performance computational environment for N-body simulations.

2.1 Introduction

The importance of N-body codes for the simulation of the dynamics of astrophysical systems has been discussed in chapter 1. The core of an N-body code is the computation of

†This chapter is based on work published in:

the gravitational interactions between all pairs of particles that compose the system. In sections 1.2 and 1.4 we discussed the main algorithms developed for the computation of gravity interactions between a given particle \( i \) and the rest of the system. In this chapter we study the performance of the direct particle-particle method (Aarseth, 1985, 1999), which exactly computes the gravity force that every particle in the system exerts on \( i \). The high accuracy of the direct method is obtained at the cost of a computational load which grows as \( N^2 \) per time step.

The huge computational requirements of the direct \( N \)-body code led to the development of the GRAPE, a special purpose device for gravity force computation, described in section 1.3. A principal objective of our research is the efficient integration of GRAPE boards with a parallel general purpose host, to realise a hybrid architecture for \( N \)-body simulations, as discussed in section 1.3.4.

The performance analysis research presented in this chapter aims at understanding how such architectures interact with the \( N \)-body code. For this purpose, we use NBODY1 (Aarseth, 1963; Aarseth, 1985) as a reference code. NBODY1 was introduced in section 1.4.1. We use it to determine the scaling properties of various parallel versions of the code, running on a hybrid architecture which includes two GRAPE-4 boards connected to a distributed computer (see fig. 2.1). The performance data obtained will be used in chapter 3 for the realisation and calibration of a performance model that we use to study hybrid architectures for \( N \)-body simulations, and their interaction with various types of \( N \)-body codes.

### 2.2 System description

#### 2.2.1 Architecture

Our hybrid architecture, sketched in fig. 2.1, is composed of a parallel general purpose multicomputer, DAS (Bal et al., 2000), and an SPD, GRAPE (see, e.g., Makino et al., 1997; Makino & Taiji, 1998). The DAS multicomputer is a wide-area distributed computer including 200 nodes in total, grouped into four clusters located at different locations in the Netherlands. The cluster at the University of Amsterdam, which served as a testbed for our model, comprises 24 processors. Technical characteristics of our testbed system are summarised in table 2.1.

In January 2002 the new DAS-2 came into service. DAS-2 is also a wide-area distributed computer including in total 200 1-GHz dual Pentium-III nodes grouped in five local clusters interconnected via the Dutch university Internet backbone. Local clusters are connected by a fast Myrinet network having a bandwidth of 250 GBytes/s peak-performance. We used the DAS-2 for the \( N \)-body simulations presented in chapter 5.

The GRAPE project, as described in section 1.3, started in the late eighties, and has produced a series of very high performance devices, mainly for the computation of the gravitational force. The GRAPE-4 system, completed in 1995, was the first computer to reach the TFlop/s peak speed (Makino et al., 1997). The current peak performance of the latest machine, the GRAPE-6, is 63.6 TFlop/s (Makino et al., 2002).
2.2. SYSTEM DESCRIPTION

We study the performance of a system consisting of two GRAPE-4 boards, each one attached to a host processor via a PCI channel. The performance of a single GRAPE-4 board can reach 30 GFlop/s. A single board comprises an array of pipelines (up to 96 per board). Each pipeline performs, at each clock-cycle, the computation of the gravitational (or electrostatic) interaction between a pair of particles. The main technical characteristics of our system are summarised in table 2.1 below:

<table>
<thead>
<tr>
<th>local network</th>
<th>Myrinet</th>
<th>150 MBytes/s peak-performance</th>
<th>40 μs latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>PPro 200 MHz</td>
<td>64 MB RAM</td>
<td>2.5 GB disk</td>
</tr>
<tr>
<td>GRAPE board</td>
<td>up to 320 MFlop/s</td>
<td>62 resp. 94 pipelines</td>
<td>on-board memory for</td>
</tr>
<tr>
<td></td>
<td>per pipeline</td>
<td></td>
<td>~ 44000 particles</td>
</tr>
<tr>
<td>host-GRAPE channel</td>
<td>PCI9080</td>
<td>33 MHz clock</td>
<td>133 MBytes/s</td>
</tr>
</tbody>
</table>

Table 2.1: Technical data concerning our testbed architecture.

Figure 2.1: The DAS cluster at University of Amsterdam, including 24 processor nodes, two GRAPEs, and 4 network switches.
2.2.2 Application

The direct $N$-body method

A formal solution for the $N$-body problem is known only for $N = 2$, making a numerical approach necessary when a solution for a larger system is desired. As discussed in section 1.4, a range of techniques has been developed to implement a numerical solution for the $N$-body problem (Aarseth, 1999; Barnes & Hut, 1986; Cheng et al., 1999; Hockney & Eastwood, 1988). We are concerned with the direct method, which computes gravitational interactions exactly, and with the treecode, which approximates this force evaluation, gaining in performance, at the cost of a lower accuracy. The treecode (Barnes & Hut, 1986) is able to reach a $\mathcal{O}(N \log N)$ scaling, compared to the $\mathcal{O}(N^2)$ scaling of the direct code. Other codes, as the FMM (Cheng et al., 1999) of the Particle-Mesh (PM) code (Hockney & Eastwood, 1988), reach $\mathcal{O}(N)$ (see section 1.2). The FMM is routinely used in applications where the Coulomb force plays a central role. The PM code is primarily used in Computational Cosmology.

In chapter 3 we discuss our simulations of direct code and treecode performance on hybrid architectures. In section 3.4.4 we compare the direct code with two different parallel versions of the treecode. In the sequel we describe the main tasks of the direct code that we analyse in this chapter, i.e. NBODY1 (Aarseth, 1999), introduced in section 1.4.1. The original serial code has been parallelised, and a number of modifications have been made, to obtain an optimal use of the GRAPE's capabilities, as described in section 2.3 below.

Code tasks

In the $N$-body computations, the particles that exert the force are commonly called $j$-particles, and the particles that experience the force are the $i$-particles. As discussed below, for each iteration, force is computed only on a small subset of particles, so that only a few particles are used as $i$-particles. On the other hand, since all particles in the system exert force, every particle plays the role of a $j$-particle, including the $i$-particles.

As mentioned in section 1.4.1, NBODY1 implements the individual time step scheme: particles experiencing a strong or rapidly changing force field need to be updated more frequently than particles moving through a quiet, nearly constant potential region. NBODY1 computes forces, and integrates orbits for each particle at the rate required by the particle dynamics itself. The individual time step is described in more technical detail in section 2.3 below.

A basic task graph of NBODY1 code-flow is given in fig. 2.2, together with the mapping of each task on the appropriate hardware resource, in case the GRAPE is available. If GRAPE is not available, all tasks are executed on the general purpose machine. The tasks shown in the figure are described in the following.

1. The first task of the main cycle is to find the $i$-particles. This is implemented as a search through a list of candidates, which is scrolled at each iteration, and rebuilt every DTLIST time units, where DTLIST is the average particle time step. The $i$-particles are selected simply by picking those particles that need to be updated first.
2.2. SYSTEM DESCRIPTION

2. Then, since stored values of positions and velocities of different particles refer to different times because of the individual time step, an extrapolation of the position values for the entire set of particles is done, to "synchronise" the system to the time value of the \( i \)-particles. The GRAPE also contains a pipeline to perform the extrapolation of the \( j \)-particle positions (see fig. 1.4). Hence, when the GRAPE is available, this task is executed on it. Still, the host has to extrapolate the \( i \)-particle positions, therefore in fig. 2.2 the extrapolation task is mapped on the host for the \( i \)-particle extrapolation, and on the GRAPE for the \( j \)-particle extrapolation.

3. Now accelerations are computed; when the GRAPE is available, \( i \)-particle data are sent to it, and it will return the accelerations.

4. Finally, orbits are integrated using the forces computed in the previous task, and relevant physical quantities are evaluated and updated.

**Code parallelisation**

In the parallel application, we distributed the \( j \)-particles equally between two GRAPEs, i.e. we loaded the \( j \)-particle memory of each GRAPE with half of the particle set. All GRAPE hosts have a copy of the entire set of particles. Each SPD computes the partial force exerted on the \( i \)-particles by the \( j \)-particles that it stores; these values are then communicated to the host. A global sum done by the hosts makes the total force on each \( i \)-particle available to all processors, that finally integrate the \( i \)-particle orbits. When the GRAPE boards are not available, the algorithm works in a very similar fashion. In this case the \( j \)-particles are distributed by assigning each processor a different subset of particles, so that a processor will evaluate only forces exerted by its own \( j \)-particles. In section 2.3 below, we describe in detail the parallel codes that we studied.
Figure 2.3: Sketch depicting the individual time step machinery. The update time is determined as the smallest $t_i + \Delta t_i$ (which in this figure is the value of particle #3). Particle positions are then extrapolated from $t_i$ to the update time, in order to compute force on particle #3, and integrate its orbit to the update time. Finally, the new $\Delta t$ for particle #3 is computed, and the next update time is determined. In the figure above, the next update time will probably be $t_4 + \Delta t_4$, unless the new $\Delta t_3$ is very small.

2.3 Code parallelisation

In this section, we describe the parallelisation of the various flavours of the direct $N$-body code used for our performance analysis and simulation. We chose NBODY1 (Aarseth, 1963) as the instantiation of a direct $N$-body code to experiment with, because it is a rather simple code, but uses almost all the functionalities of GRAPE. This allows us to evaluate the performance of our system. A number of modifications have been made to the code, in order to parallelise it, and to let it make full use of the functionalities of GRAPE.

An overview on the code is given below. We made use of MPI communication primitives (Message Passing Interface Forum, 1997) to parallelise it.

2.3.1 The basic: individual time-step

As already mentioned in section 1.4.1, NBODY1 uses individual time-steps. Each particle is assigned a different time at which the force will be computed. Fig. 2.3 depicts this procedure. The time-step value of each particle $\Delta t$, sketched in fig. 2.3 for each particle as a segment, depends on the particle dynamics (Aarseth, 1999). Smaller $\Delta t$ values are assigned to particles having faster dynamics (i.e. those particles which have large values in the higher order time
2.3. CODE PARALLELISATION

derivatives of their acceleration) according to the formula (see, e.g., Aarseth, 2001)

$$\Delta t_i = \sqrt{\frac{\eta |\mathbf{a}_i| |\mathbf{\dot{a}}_i| + |\mathbf{\ddot{a}}_i|^2}{|\mathbf{\dot{a}}_i| |\mathbf{\ddot{a}}_i| + |\mathbf{\dddot{a}}_i|^2}},$$

where $\eta$ is an accuracy parameter of order unity. At each iteration, the code selects that particle having the smallest $t + \Delta t$ value (particle 3 in fig. 2.3), and integrates only the orbit of that particle. This reduces the computational complexity, with respect to a code where a single global time step is used. The individual time step approach reduces the temporal complexity to $O(N^{1/3})$, whereas the global time step approach is $O(N^{2/3})$ (Makino & Hut, 1988). This temporal complexity refers to the computational effort needed to integrate the system for a dynamical time, i.e. the average time taken by a particle to cross the system.

An effect of individual times is that, for each particle, values stored in memory refer to a different moment in time, i.e. the moment of the particle’s last orbit integration. This means that, before force on particle $i$ is computed, an extrapolation of the other particle positions to time $t_i + \Delta t_i$ is needed. The time value $t_i + \Delta t_i$ is marked in fig. 2.3 by the “update time” line.

Parallelisation

Since contributions to the gravity force on a given particle $i$ are computed from all the other particles using eq. (1.1), regardless of their distance from $i$, a uniform distribution of particles to each processing element (PE), i.e. to each DAS node, suffices to assure load balancing. The force computation is done by broadcasting the coordinates of the currently selected particle $i$. Then each PE computes the partial component to the force on $i$, by accumulating contributions from its own particles. Finally such components are sent back to the PE which hosts $i$, where the force resultant is computed, the particle’s orbit is integrated, and the new values are stored.

To identify the particle $i$ on which force will be computed, a global reduction operation is done, in order to find which particle has the least $t_i + \Delta t_i$ value, and which PE owns it. This information is broadcast to all PEs, since they must know the extrapolation time, and the $i$-particle owner.

2.3.2 Towards a GRAPE code: block time-step

Since its introduction, NBODY1 has evolved to newer versions, which include several refinements and improvements (see, e.g., Aarseth, 1999). In the version of NBODY1 used in our study we implemented the so called hierarchical block time-step scheme (McMillan, 1986; Makino, 1991a). In this case, after computing the new $\Delta t_i$, the value actually assigned is the value of the largest power of 2 smaller than $\Delta t_i$. This allows for more than one particle to have the same $\Delta t$, which makes it possible to have many $i$-particles per time step, instead of

\footnote{These figures for the temporal complexity are valid for a uniformly distributed configuration. More realistic distributions show a more complicated dependence on $N$, although quantitatively only slightly different.}
only one. Using this approach, force contributions on a, possibly large, number of \(i\)-particles can be computed in parallel using the same extrapolated positions for the force-exerting particles, hereafter called \(j\)-particles. Moreover, when a GRAPE device is available, it is possible to make full use of its array of pipelines, since each pipeline can compute the force on a different particle concurrently.

**Parallelisation**

Having many \(i\)-particles, instead of just one, makes it profitable to use a somewhat different parallel code structure. If the \(i\)-particles reside on different processors, distributing the particles as in the individual time-step case could result in complex communication patterns, with consequential increase of code complexity. Therefore, we chose to let every PE have a local copy of all particle data. The force computation is done in parallel by making each PE compute force contributions only from its own set of \(j\)-particles, assigned to it during initialisation. A global reduction operation sums up partial forces, and distributes the result to all PEs. Then each PE integrates the orbits of all \(i\)-particles, and stores results in its own memory. Concerning the search for \(i\)-particles, each PE searches only among its \(j\)-particles, to determine a set of \(i\)-particles candidates. Then a global reduction operation is performed on the union of these sets, in order to determine the real \(i\)-particles, i.e. those having the smallest time. The resulting set is scattered to all PEs for the force computation. Since every PE owns a local copy of all particle data, only a set of labels identifying the \(i\)-particles is scattered, reducing the communication time.
2.4. CODE PERFORMANCE

2.3.3 The GRAPE code

The software library interface for the GRAPE hardware consists of a number of function calls, the most relevant for performance analysis being those which involve communications of particles data to and from the GRAPE. Such communication operations include sending $j$-particle data to GRAPE, sending $i$-particle data to GRAPE, and receiving results from GRAPE. A sketch of the program flow for an $N$-body code which uses GRAPE is given in fig. 2.4.

Parallelisation

The presence of the GRAPE boards introduces a certain degree of complexity with respect to code parallelisation. The GRAPE-hosts obviously play a special role within the PEs set. This asymmetry somehow breaks the SPMD paradigm that parallel MPI programs are expected to comply with. Besides the asymmetry in the code structure, also the data distribution among PEs is no more symmetric. The force computation using GRAPE is performed, similarly to the non-GRAPE case, by assigning an equal number of $j$-particles to each GRAPE. The GRAPE computes the partial force exerted by the $j$-particles assigned to it on the $i$-particle set, which is the same for all GRAPEs. After that, a global sum on the partial results, performed on the parallel host, will finally give the total force.

Since force computations and $j$-particle position extrapolations are done on the GRAPE, the only relevant work to execute in parallel by the PE set is the search for $i$-particle candidates, which is accomplished exactly as in the code described in section 2.3.2 above.

2.4 Code performance

We describe and analyse in this section the measurements that we carried out for the performance evaluation of the codes described in section 2.3. Our measurements are intended to explore the scalability of parallel $N$-body codes. We performed runs varying both the number of particles $N$ and the number of processors PEs; we scaled $N$ from 1024 to 16384, and PEs from 1 to 24. NBODY1 does not need a large amount of run-time memory, just about 200 bytes per particle, but is heavily compute-bound (Hut, 1996). Our timings were carried out in order to show the relative computational requirements of the various code tasks, and how these change as a function of $N$ and PEs. Reported values are averages of the values measured for each processor. These measurements showed a negligible deviation, which is thus not reported on the figures below.

Our runs were started having a Plummer model distribution (Plummer, 1915) as initial condition, in which density decreases outward as the fifth power of the distance from the cluster centre. The gravity force is modified by introducing a softening parameter, which is a constant term, having the dimension of a length, whose squared value is inserted in the denominator of the gravity force expression eq. (1.1) (see also caption of fig. 1.1). The softening parameter reduces the strength of the force in case of close encounters and thus
prevents the formation of tightly-bound binaries. In this way very short time steps and correspondingly long simulation times are avoided. In our runs, this parameter was set equal to 0.004. As a reference, the mean inter-particle distance in the central core of the cluster, when $N = 16384$, is approximately equal to 0.037 in $N$-body units (Heggie & Mathieu, 1985).

### 2.4.1 Individual time step code

The essential tasks of this version of the code (hereafter called IND) are shown in the code flow sketched in fig. 2.2. As described in section 2.3.1, the parallel version of this code implements communications in the $i$-particle search task, then when the $i$-particle position is broadcast, and when partial forces are gathered by the PE that owns the $i$-particle. Fig. 2.5 shows the timings, and fig. 2.6 the performance of the parallel version of the IND code.

The metric we use to quantify the code performance is the parallel efficiency, defined as:

$$P_n = \frac{t_1}{n \cdot t_n}$$

where $n$ is the number of PEs used, and $t_n$ the execution time when using $n$ PEs. The timings shown in the figures refer to 1000 iterations of the code. The $t_n$ values depend about linearly on $N$, since the number of operations to compute the force on a given particle scales linearly with $N$, and in each run the same number of force computations is performed, i.e. 1000, independently of the total number of particles. An interesting super-linear speedup is visible in fig. 2.6, which can be explained with an optimised cache utilisation. The IND code, when the work-load is high ($N \geq 8192$), is highly compute-intense, as fig. 2.8 clearly shows.
2.4. CODE PERFORMANCE

![Graph showing performance vs. PEs for different values of N.]

Figure 2.6: Performance of the parallel individual time-step code, running for 1000 iterations. One iteration consists of advancing a single particle per time step. A super-linear speedup effect, discussed in the main text, can be seen for intermediate values of the work-load per processor.

In this case, also when the number of processors is high, thus with relative small number of particles per processor, the communication overhead is still small. Since the number of particles per processor decreases as PEs increases, the number of cache misses decreases too, thus the cache is better exploited as PEs increases. This effect, combined with the limited importance of the communication overhead for the high workload cases, leads to the super-linear speedup visible in fig. 2.6.

Fig. 2.7 and 2.8 show the fractional time shares of each task, and how these shares change as the number of PEs changes. Fig. 2.7 shows the time shares for runs with $N = 1024$, and fig. 2.8 for runs with $N = 16384$. Fig. 2.7 clearly shows how the IND code suffers from a communication overhead when the computational work-load is light. On the other hand, as shown in fig. 2.8, the code performs quite satisfactorily when this ratio is high, thanks to the compute-intense characteristics of the $N$-body code, and the high performance communication network of our architecture.

2.4.2 Block Time-step Code

The basic tasks of this version of the code (BLOCK hereafter) are the same as the IND code. The only difference is that now the number of $i$-particles per iteration can be larger than one. As stated in section 2.3.2, this optimises the force computation procedure, also in view of the use of GRAPE, but, on the other hand, increases the communication traffic, since information about many more particles must be exchanged at each time step.
CHAPTER 2. N-BODY CODES ON HYBRID ARCHITECTURES

Figure 2.7: Execution time shares vs number of processors for the IND code. Runs with 1024 particles.

Figure 2.8: Execution time shares vs number of processors for the IND code. Runs with 16384 particles.
2.4. CODE PERFORMANCE

![Graph](image)

Figure 2.9: Global timings for the parallel block time-step code. In this case, at each time step, force is computed on many particles.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\langle N_i \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>35.05</td>
</tr>
<tr>
<td>2048</td>
<td>43.91</td>
</tr>
<tr>
<td>4096</td>
<td>111.16</td>
</tr>
<tr>
<td>8192</td>
<td>207.52</td>
</tr>
<tr>
<td>16384</td>
<td>351.14</td>
</tr>
</tbody>
</table>

Table 2.2: Mean number of $i$-particles per iteration in the BLOCK code runs.

The effect of this is clearly shown in the figures presented here. Fig. 2.9 shows total timings, and fig. 2.10 shows performance of this code. In this case the execution time grows as a function of $N^2$ because the number of $i$-particles, i.e. the number of force computations, grows approximately linearly with $N$. Since the computational cost for the force on each particle also grows linearly with $N$, the resulting total cost is $O(N^2)$. The mean number of force computations per iteration as a function of $N$ is given in table 2.2. Fig. 2.10 shows how the performance gain of this code is less spectacular than the gain of the IND code, since communication overhead plays a larger role in the total execution time. This large overhead can be seen in fig. 2.11 and 2.12, that show how the execution time shares evolve as a function of PEs number. These figures show that for the BLOCK code, almost all the computational part of the execution time is spent in the force computation task. The $j$-particles extrapolation, that takes roughly 25% to 30% of the total time in the IND code (see figures 2.7 and 2.8), is reduced to less than one percent.

2.4.3 GRAPE Code

The code version which makes use of GRAPE boards will be called GRP hereafter. We present performance results of both the serial, and the parallel implementation. The communication overhead of the parallel version is composed of host-GRAPE communication and network communications. The parallel code runs have been done by using only the DAS
Figure 2.10: Performance of the parallel block time-step code.

Figure 2.11: Execution time shares vs number of processors for the BLOCK code. Runs with 1024 particles.
nodes connected to the GRAPE boards at our disposal, thus the maximum number of PEs in this case is 2.

It is clear from fig. 2.13 that the parallel performance is very poor. In that figure, GRAPE0 refers to the GRAPE with 62 pipelines, and GRAPE1 to the GRAPE with 94 pipelines. Fig. 2.13 also shows that runs on GRAPE1 are a bit faster, thanks to the larger number of pipelines available. The low parallel performance shown in fig. 2.13 can be explained by the low number of $i$-particles, especially for the low-$N$ runs, that prevents the GRAPE boards to be fully exploited. Moreover, a large communication overhead dominates the GRP code, as fig. 2.14 for the GRAPE0 case, fig. 2.15 for the GRAPE1 case, and 2.16 for the parallel case show. These figures also show that the time share spent in GRAPE computations (i.e. force computations) is quite low, resulting in a low efficiency of this code in terms of GRAPE exploitation. One reason for that is of course the very high speed of the GRAPE. This device is by far faster in accomplishing its task than its host and the communication link between them.

Another effect that can be seen in the figures is the increase of the time share of the orbit integration task when $N$ goes from 2048 to 4192. This can be explained by the increase of cache misses when this task is executed. The cache size of a node is 256 Kbytes, which makes it able to contain data for about 1000 particles (each particle carries about 200 bytes of data). The orbit integration task works on data which are located randomly on the memory, thus the chance of a cache miss when the cache does not contain the whole data set is relatively high. This effect produces the increase of the orbit integration time share between $N = 2048$ and $N = 4192$. Subsequently, the timings are more and more dominated by the increase of the GRAPE computation time share.
The figures clearly show that for our hardware configuration the capabilities of the GRAPE will only be fully utilised for problems involving over 40,000 particles per GRAPE. This number is, however, limited by the GRAPE on-board memory for $j$-particles, which is only slightly higher than 40,000.

Our measurements of the low level host-GRAPE communication routines show that a large amount of time spent in communication is due to software overhead in copy operations and format conversions. As an example, we show in fig. 2.17 measurements done on the $j$-particle send operation. Similar measurements (Kawai et al., 1997), performed on a faster host, showed a higher communication speed, linearly dependent on the host processor clock speed. Nevertheless, even though the GRAPE boards are not exploited optimally, the execution times for the GRP code are by far shorter than those for the BLOCK code. The heaviest run on 2 GRAPEs is about one order of magnitude faster than the heaviest run of the BLOCK code on 24 PEs. Considering the total amount of computing power used in these two cases, i.e. the execution time times the number of processors used, shows that the BLOCK code needs about 140 times more computing time to perform the same amount of work as the GRP code. A global comparison of the throughput of all codes studied here is given in section 2.4.4 below.

### 2.4.4 Code Comparison

In order to evaluate the relative performance of the three versions of the $N$-body code studied in this chapter, a series of runs has been made, where both a 8192 particles system, and a 32,768 particles system were simulated for 7200 seconds. We compare the performance of
Figure 2.14: Execution time shares vs number of particles for the GRP code. Runs on GRAPE0 (62 pipelines).

Figure 2.15: Execution time shares vs number of particles for the GRP code. Runs on GRAPE1 (94 pipelines).
Figure 2.16: Execution time shares vs number of particles for the GRP code. Runs on both GRAPEs.

Figure 2.17: Software overhead in the $j$-particle send operation. The difference between the top-most timing (the cumulative task timing) and the timing immediately below is due to format conversion. The other differences are mainly due to copy operations.
2.4. CODE PERFORMANCE

![Performance Comparison](image)

Figure 2.18: Performance comparison for the three versions of the N-body code. Runs with 8192 particles. The IND and BLOCK codes are run on 24 processors, the GRP code is run on two processors each connected to a GRAPE.

the GRP code, with respect to the other codes run on the general purpose host, against an increasing computational load. The fastest hardware configuration is used in each case, i.e. 24 PEs for the IND and BLOCK code runs, and 2 PEs (and hence 2 GRAPEs) for the GRP run. Fig. 2.18 and 2.19 show the evolution of the simulation time, as a function of the wallclock time. In this way, the performance of each code is specified in terms of how long one should wait before a simulation reaches a certain N-body time. Those figures show that the GRP code outperforms the other two codes by a factor 8, when the computational load is lighter, and by a factor 20, with a heavier computational load. In both cases, the BLOCK code is 1.5 times faster than the IND code, thanks to the optimisation of the j-particles extrapolation step. Fig. 2.19 shows an initial overlapping of these two codes performance curves, due to a start-up phase, which is not visible in fig. 2.18, because at the first timing event (after 60s) this system is already stabilised.

Fig. 2.18 and 2.19 clearly show the large performance gain obtained with GRAPE. Using only two PEs, an order of magnitude better performance was attained compared to the BLOCK code on 24 PEs. Due to the reduction in the time needed for the force calculation, the communication overhead for the GRP code accounts for approximately 50% of the total execution time (see fig. 2.15 and 2.16). Hence an even larger relative gain may be expected for larger problems, as the relative weight of the communication overhead will decrease. The difference in performance between the two cases shown respectively in fig. 2.18 and 2.19 clearly illustrates this effect.
2.5 Discussion

Our performance analysis reveals a very good parallel performance of the BLOCK and especially the IND code. We also show that the use of GRAPE leads to a dramatic performance gain, even at a low efficiency in terms of GRAPE boards exploitation. Such low efficiency is mainly due to a very high communication overhead, even for the largest problem studied. This overhead can be greatly reduced by the use of a faster host, and by the development of an interface requiring fewer format conversions. The GRAPE-hosts in the system studied in this chapter have a 200 MHz clock speed. Nowadays standard clock speeds are up to one order of magnitude faster. The use of a state-of-the-art processor would reduce the host and communication times significantly. The low utilisation of GRAPE, shown in fig. 2.14, 2.15 and 2.16, suggests that the problem size has to be increased to attain a optimal SPD utilisation.

The measurements described in this chapter are the basis for the calibration and validation of our performance simulation model. In chapter 3 our model will be described, and used to simulate different classes of $N$-body codes, running on different hybrid architectures.