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 3

Modelling and Simulation of Hybrid Architectures

3.1 Introduction

In this chapter the performance model that we developed to simulate the behaviour of hybrid architectures is introduced. Hybrid architectures were presented in section 1.3.4 as systems where a high performance general purpose computer is coupled to one or more Special Purpose Devices (SPDs). They can be seen as a special case of computer systems described by the heterogeneous computing paradigm (Freund & Siegel, 1996; Palazzari et al., 2000). In section 1.3.4 we also discussed why such a system can be the optimal choice for several fields of Computational Science. The relevance of the GRAPE in the field of Numerical Astrophysics has been discussed in section 1.3. Quantum Chromodynamics is another field that has benefited substantially from the use of SPDs. In a recent review on Computational Quantum Chromodynamics the state of the art for the use of dedicated computers in that field has been presented, with the Japanese CP-PACS computer (Aoki et al., 1999), the

†This chapter is based on work published in:
QCDSP machine built in the USA (Mawhinney, 1999), and the APE system developed in Europe (Tripiccione, 1999).

Configuring a hybrid system and finding the optimal mapping of the application tasks onto the hybrid machine often is not straightforward. Performance modelling, which we discussed in section 1.6, provides a tool to tackle and solve these problems. We developed a performance model to simulate a hybrid architecture consisting of a parallel multiprocessor where some nodes are the host of a GRAPE board. GRAPE, introduced in section 1.3, is a very high performance SPD used in Computational Astrophysics.

We present here the general modelling framework, and the methodological approach that we used to build our model. Based on this modelling background, and on the experimental data presented in chapter 2, we developed the performance models described in this chapter. We present here the details of the implementation of both the model used for the simulation of the direct code, introduced in section 1.4.1 and discussed in detail in section 2.2.2, and the model for the simulation of the treecode, introduced in section 1.4.2. We validate the accuracy and versatility of our models by simulating existing configurations, and use them to forecast the performance of other architectures, in order to assess the optimal hardware-software configuration.

Forecasting and analysing the performance of a hybrid architecture is not trivial. Performance modelling can provide a solution to this problem. The range of applications of performance modelling in Computer Science is vast. Recently, in a review of performance modelling research, applications were presented spanning the range from scheduling in global computing systems (Aida et al., 2000) to modelling of large-scale scientific applications (Adv & Sakellariou, 2000), based on both the analytical approach (e.g., Gunther, 2000; Hoisie et al., 2000) and on simulation (e.g., Kurc et al., 2000).

Analytic models (see, e.g., Cremonesi & Gennaro, 2002) easily become intractable due to the complexity of the simulated system, and usually show a limited flexibility. Simulation models (Bagrodia et al., 1998; Adv et al., 2000) allow for the study of very complex systems. Their high degree of versatility makes it possible to estimate the performance of hardware or software architectures during the various phases of their development (see, e.g., Pimentel et al., 2001).

We have built a performance model, based on functional task modelling (Dikaiakos et al., 1996). Our model simulates the behaviour of a parallel multiprocessor, where specific nodes can act as the host of an SPD. This helps us to understand the interactions between the SPD, the host, and the application that is run on the hybrid system. Our aim is to have the possibility to adapt and modify the hardware model, in order to find the configuration that gives the best performance, and to simulate a different software application just by changing the higher level software specifications. Hence we developed a model able to make predictions of the performance of the system for a given algorithm, and to tell us how hardware and software can be adapted to one another to obtain the best performance.

The hybrid system that we used to validate our model consists of a local cluster of the DAS parallel computer (Bal et al., 2000), where two nodes are the host of a GRAPE board. A direct summation N-body code (Aarseth, 1999) that, as described in chapter 2,
we parallelised and adapted for use with the GRAPE, was executed on this system. We validated our performance model on this architecture, and used the model to make predictions on the system behaviour, when both hardware and software modifications are introduced. Furthermore, we also studied the behaviour of a treecode (Barnes & Hut, 1986) on such a system.

3.2 Design considerations

3.2.1 Requirements

Hybrid Architectures can be complex to design, and expensive to realise. Performance modelling is an effective tool to estimate their performance rapidly and inexpensively. We aimed to build a versatile model, able to simulate different applications running on different computer architectures. Therefore, we have structured our model so as to separate the modelling of the hardware from the modelling of the algorithm. This allows us to modify the model of the application, leaving intact the underlying model of the machine, and vice versa.

Scope of our model

Generally, performance models are designed to simulate an application or a hardware architecture in great detail, and need powerful simulation environments, such as POEMS (Adve et al., 2000), a comprehensive environment for the study of complex computer systems, or Artemis (Pimentel et al., 2001), specifically developed for embedded systems design and analysis. In our case, we do not aim at simulating our application down to the single instruction level, or our machine at the single electronic component level. We focus on the interaction of the SPD with the parallel host, and the interplay of those two components with the application executed on them. We use an iterative refinement approach, starting coarse, and, if necessary, refining those modules that produce unacceptable errors. For this purpose, we found it sufficient to model the system components at a functional level (Dikaiakos et al., 1996). This approach involves much less model complexity, still giving us sufficiently accurate results.

Level of granularity

The level of granularity of our model is dictated by the accuracy that we want to reach in our simulations, taking into account that the aim of our performance analysis research is the optimal performance of the software application, typically achieved by balancing the hardware components' workload for a given set of software application tasks. As shown below, we get a satisfactory accuracy with a rather "coarse grained" functional model. The basic unit of our abstract algorithm is the task, defined as a code block which encompasses a set of instructions performing a specific operation. This operation is characterised by having a non negligible execution time, accessing a set of resources which is constant in time, and
depending on a limited number of application parameters. Similarly, the model granularity for the architectural components has been set at the level where the atomic units are the major computing elements, such as the SPD and the host node.

Model structure

The computational environment that we model is specified, at the more abstract level, by a number of formal entities. These are the algorithm, the hybrid machine, and the mapping interface. In the specific case described here, the algorithm computes the numerical solution of the gravitational $N$-body problem. The algorithm model generates simulation parameters, and activates basic operations. The different operations of this code have different demands for computational power, the force computation being by far the most demanding task. The design of the hybrid architecture on which this algorithm is executed matches these requirements, by including a model of a specialised hardware for the gravitational force evaluation. A sequence of tasks describes the behaviour of each component, and the concurrent access to machine components by a set of application tasks is treated as a critical section.

3.2.2 Functional model and implementation environment

Our functional model approach has been presented in section 1.6, where we described how we identify the main constituents of the modelling environment. We make a model of the software application in the application model, where we specify the time spent on each task $T_i$ as a function of the application parameters $\pi_i$. Similarly, in the machine model, the characteristics of the hardware resources $R_j$ depend on the machine parameters $\mu_i$. The mapping interface maps each $T_i$ of the application model on the appropriate $R_j$ of the machine model. The resulting simulation model returns the simulated execution time, which depends on both the $\pi_i$ and $\mu_i$. In this way, we can study the performance of existing systems, and forecast the performance of the systems under design.

The simulation language used to implement our model is PAMELA (PerforMAnce Mod-Eling LAnguage) (van Gemund, 1993, 2003), developed by Arjan van Gemund at the Delft University of Technology, aimed at either simulation or analytic performance analysis. PAMELA is a C-style procedure-oriented simulation language where a number of operators model the basic features of a set of concurrent processes. In a procedure-oriented language, concurrent process interaction takes place via shared variables, in contrast to message-oriented languages, which describe communications in terms of explicit messages between interacting processes.

The execution time of a process is modelled by the delay statement; the sequential execution of processes is implemented by the seq (prefix) or ; (infix) construct. Parallelism is specified by means of the par (prefix) and | (infix) constructs, which are implemented in a fork/join fashion (i.e. with implicit synchronisation). Explicit synchronisation between a couple of processes is implemented with the wait and signal operators, while mutual exclusion is realised with the P and V semaphore statements, which implement Dijkstra's classical solution to the resource contention problem (see, e.g., Tanenbaum, 2001, §2.3.5).
3.3. MODEL IMPLEMENTATION

PAMELA models the execution of processes in terms of the Discrete Event Simulation paradigm, as the use of the delay primitive suggests. A model can be material-oriented, when the execution flow of the process is specified in terms of the various system resources that the process will access, or machine-oriented, where the emphasis is on the resource, with a specification of the series of operations that each resource should accomplish. PAMELA is more suited for the first procedural approach, although machine-oriented models can also be built within this framework.

In order to show how PAMELA is typically used to describe a parallel system, we give as an example the model of a client-server system, where C concurrent clients execute N iterations each. An iteration consists of local processing with duration $\tau_i$, followed by a request to access the server $s$ which, once accessed, is used for a time $\tau_s$. Such system is modelled by:

$$
\text{par}(p=1\ldots C) \quad \text{seq}(i=1\ldots N) \quad \{ \text{delay}(\tau_i) ; \ P(s) ; \text{delay}(\tau_s) ; \ V(s) \}
$$

where line breaks and indentations are used for the sake of clarity, and have no syntactic meaning. A detailed overview on PAMELA is given in van Gemund (1993).

3.3 Model implementation

In this section we describe how our performance model reproduces the tasks of the codes under study. The direct code tasks have been described in section 2.2.2. Here we specify the dependence of the execution time of each task on the application parameters, like $N_i$, the number of $i$-particles, and $N_j$, the number of $j$-particles.\footnote{We recall that $i$-particles are the particles on which force is computed, whereas $j$-particles are the particles from which force is computed.}

3.3.1 Direct code

Our application model for the direct code is modelled as a sequence of tasks, as sketched in fig. 3.1. Each computation task is implemented by a delay statement (see section 3.2.2), possibly followed by a support function that sets the value of time dependent parameters, as $N_i$ and $N_j$. As described in section 2.3, there are communication operations at the end of the $i$-particle search and force compute tasks, and both are global all-to-all operations. They are implemented in the model by means of a synchronisation operation, followed by a delay statement. The delays in the model of each task depend on the model parameters according to the formulae reported in table 3.1. These expressions have been obtained by analysing the data presented in chapter 2, and inferring the dependence of the execution time of each task on the model parameters. In the following section we describe the subset $\mathcal{P}_i$ of application parameters that affect the performance of each task $T_i$.\footnote{We recall that $i$-particles are the particles on which force is computed, whereas $j$-particles are the particles from which force is computed.}
CHAPTER 3. MODELLING AND SIMULATION OF HYBRID ARCHITECTURES

Figure 3.1: Basic sketch of the direct N-body code tasks.

<table>
<thead>
<tr>
<th>tasks</th>
<th>modelling formulae</th>
</tr>
</thead>
<tbody>
<tr>
<td>i-particle local search</td>
<td>[ \mu_P \cdot \pi_{src} \cdot \frac{N}{P} ]</td>
</tr>
<tr>
<td>i-particle global communication</td>
<td>[ \mu_L \cdot \pi_{igl} + \mu_L \cdot P + (\mu_L \cdot \pi_{int} + \mu_B \cdot \pi_{fb} \cdot \log(P)) \cdot N_i ]</td>
</tr>
<tr>
<td>extrapolation on host (non-GRAPE case)</td>
<td>[ \mu_P \cdot \pi_{xtr} \cdot \frac{N}{P} ]</td>
</tr>
<tr>
<td>extrapolation on host (GRAPE case)</td>
<td>[ \mu_P \cdot \pi_{xtr} \cdot N_i ]</td>
</tr>
<tr>
<td>local force (non-GRAPE case)</td>
<td>[ \mu_P \cdot \pi_{frc} \cdot N_i \cdot \frac{N}{P} ]</td>
</tr>
<tr>
<td>local force (GRAPE case)</td>
<td>[ (\mu_S + \mu_G \cdot \frac{N}{G}) \cdot \left[ \frac{N_i}{n_{pipes}} \right] ]</td>
</tr>
<tr>
<td>j-particle send to GRAPE</td>
<td>[ \mu_P \cdot \pi_{prep} \cdot \left[ \frac{N_j}{90} \right] + \mu_C \cdot \pi_{fpart} \cdot N_j ]</td>
</tr>
<tr>
<td>i-particle send to GRAPE</td>
<td>[ \mu_C \cdot \pi_{ipart} \cdot \left[ \frac{N_i}{n_{pipes}} \right] \cdot n_{pipes} ]</td>
</tr>
<tr>
<td>receive results from GRAPE</td>
<td>[ \mu_C \cdot \pi_{res} \cdot \left[ \frac{N_i}{n_{pipes}} \right] \cdot p_{max} ]</td>
</tr>
<tr>
<td>force global communication</td>
<td>[ (\mu_L \cdot \pi_{fl} + \mu_B \cdot \pi_{fb} \cdot \log(P)) \cdot N_i ]</td>
</tr>
<tr>
<td>orbit integration</td>
<td>[ \mu_P \cdot \pi_{orb} \cdot N_i ]</td>
</tr>
</tbody>
</table>

Table 3.1: Synopsis of the application tasks, and the modelling formulae for their time dependence on the model parameters, whose values are given in table 3.2. Here, G is the total number of GRAPE boards, \( n_{pipes} \) is the number of pipelines in a GRAPE board, \( p_{max} \) is the maximal number of pipelines in a GRAPE board; for the GRAPE-4 \( p_{max} = 96 \) (see section 1.3 for details). The other variables are defined in the text.
3.3. MODEL IMPLEMENTATION

Application model

For an N-body code, the most important parameter is obviously $N$, the total number of particles, which is a measure of the problem size. Moreover, the dynamical parameter that affects the performance of each task in a block time step code is $N_i$, the number of particles for which force is going to be computed. We observed a highly oscillatory behaviour for this parameter, shown in fig. 3.2. This oscillation of $N_i$ between high and low values can be due to a small number of binary stars, which have a strong mutual interaction, requiring a small time step, or to close encounters between pairs of stars. The high occurence of low values of $N_i$ between iteration #60 and iteration #300, implying that one or two particles evolve with a low time step, is an indication of the presence of a binary system in that simulation. The number of particles having a larger time step is also larger; when they are selected as $i$-particles, the value of $N_i$ becomes much higher. We give the value of $N_i$ at each iteration, obtained from the trace of real runs, as an input to our simulator.

$i$-particle search. The task of finding the $N_i$ particles is modelled as a linear function of $N$, since the search is done over a set of candidates, whose number is a nearly constant fraction of $N$. In the parallel case, each processor searches a local list of candidates, which is a subset of the local particle set. The actual $i$-particles are chosen after this local search is completed, again by selecting from the candidates those particles with the smallest time value.

This global search uses a collective communication. The measured communication time shows both a linear dependence on the number of processors $P$, and on $N_i$. The $N_i$ scaling factor is modulated by a term proportional to $\log P$. Based on our measurements, we used the fitting formula given in fig. 3.3 to model the global search task.

Fig. 3.3 shows the dependence of this task on $N_i$, for three different representative sets of values for $N$ and $P$. A data point in this graph is the average value of the timings on each processor at a given iteration of the code. Occasionally, values much higher than the average have been measured, as shown in the figure, arguably due to external data traffic in the network. The fitting formula is not affected by these spurious values.

Extrapolation. The extrapolation phase, in the non-GRAPe case, consists of a fixed number of operations done on every particle in the system. Each processor extrapolates only its own $j$-particle positions, thus the extrapolation time shows a linear dependence on $N/P$, i.e. the workload per processor. A sketch of the dependence of the execution time for this task on $N/P$ is given in fig. 3.4. Each point here is the average value over the entire run, for a given pair $(N, P)$. This figure shows a jump in the dependence of $t$ on $N/P$, due to a cache effect. We chose to model only the out-of-cache behaviour, because we are more interested in situations characterised by a large workload.

---

2The cache size of our system is 256 Kbytes per processor, and each particle carries about 200 bytes of data. Then a workload per processor larger than about 1000 particles will cause the problem to run out of cache.
CHAPTER 3. MODELLING AND SIMULATION OF HYBRID ARCHITECTURES

<table>
<thead>
<tr>
<th>machine parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mu_P ) (( \mu s ) to perform a processor cycle)</td>
</tr>
<tr>
<td>( \mu_L ) (network latency in ( \mu s ))</td>
</tr>
<tr>
<td>( \mu_B ) (( \mu s ) to transmit a byte over the network)</td>
</tr>
<tr>
<td>( \mu_S ) (( \mu s ) to startup the GRAPE pipeline)</td>
</tr>
<tr>
<td>( \mu_C ) (( \mu s ) for the GRAPE to compute a force interaction)</td>
</tr>
<tr>
<td>( \mu_C ) (( \mu s ) to transmit a byte on the GRAPE-host channel)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>application parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>computations</td>
</tr>
<tr>
<td>( \pi_{src} ) (i-particle search)</td>
</tr>
<tr>
<td>( \pi_{xtr} ) (part. pos. extrapolation)</td>
</tr>
<tr>
<td>( \pi_{frc} ) (force computation)</td>
</tr>
<tr>
<td>( \pi_{orb} ) (orbit integration)</td>
</tr>
<tr>
<td>( \pi_{prep} ) (format conversions in packet preparation)</td>
</tr>
</tbody>
</table>

Table 3.2: Values of the performance parameters appearing in the modelling formulae in table 3.1.

When GRAPE is used, the application code has to perform the extrapolation only for the i-particles. The GRAPE contains an extrapolation pipeline for the j-particles, but it does not extrapolate the i-particle positions; the host must perform this operation. Hence, in this case, the extrapolation task is modelled as a linear function of \( N_i \), and is mapped on the host. Since every host must extrapolate all the i-particle positions, there is no dependence
on \( P \). The GRAPE performs the \( j \)-particle extrapolation simultaneously with the force computation. When the GRAPE computes the force exerted by a certain particle \( j_1 \), the \( j_1 \)-data fetched from the GRAPE memory are passed through to the extrapolation pipeline. The pipeline outputs the extrapolated position of \( j_1 \), which is input to the force pipeline. The timing of the force computation task also includes the extrapolation. This is why there is no separate modelling for the \( j \)-particle extrapolation done on the GRAPE in table 3.1.

**Force computation.** The force computation task in the non-GRAPE case scales linearly with \( N_i \cdot N/P \). Fig. 3.5 shows this dependence for a representative set of runs. Also in this case a data point refers to a single iteration, and is the average value of the timings for all the processors.

When forces are computed on the GRAPE, it does this task on \( N_p \) particles at the same time, using its array of pipelines. Then the same amount of time is spent to compute forces, for a number of \( i \) particles ranging from 1 to \( N_p \). This time scales linearly with \( N/G \) (see fig. 3.8) where \( G \) is the number of GRAPEs available, since the force computation consists in an iteration on the \( N/G \) particles constituting the particle subset assigned to a GRAPE. The operation of receiving the result data from GRAPE is similar to the \( i \)-particle send, showing the same step behaviour.

The force computation task performed on the GRAPE shows a rather complicated structure. A number of communication procedures between the GRAPE board and the host must be performed, besides the actual force computation task. Figures 3.6, 3.7, and 3.8
Figure 3.3: Timings averages of the global communication operation associated with the $i$-particles search. The formula we obtain by fitting the measurement values, which is used in the simulation model, and reproduced here as a continuous curve, is: $t = 0.14 + 0.038 \cdot P + (0.97 \cdot 10^{-4} + 0.15 \cdot 10^{-3} \cdot \log(P)) \cdot N_i$ (see parameterised expression in table 3.1).

Figure 3.4: Timings of the extrapolation task for the non-GRAPES case, and fitting formula used into the model. A cache effect is clearly visible at $N/P = 1024$. We are interested in situations with high workload, thus we fit only the out-of-cache subset.
3.3. MODEL IMPLEMENTATION

Figure 3.5: Timings and fitting formula for the local force computation in the non-GRAPE case, for a set of representative runs.

illustrate these tasks. Before GRAPE computes forces, the host sends it the $j$-particle positions that have changed in the last iterations. Since GRAPE stores the $j$-particle data in its internal memory, only the updated $j$-particle data need to be sent to it. Moreover, also the time-advanced position of the $i$-particles need to be sent in the same packet as the $j$-particles. In this way GRAPE avoids computing the self-interaction for the $i$-particles. The actual delivery of data is done in packets of up to 90 particles, and shows a linear dependence on the amount of data sent, plus a fixed latency time for each actual send operation. The $j$-particles send step is then a function of $N_i$ and $N_j$. Fig. 3.6 shows the measured performance of this operation as a function of the data sent.

Another send operation is performed to send the $i$-particles to GRAPE. The actual data delivery is done in packets of $N_p$ particles, where $N_p$ is the number of active pipelines on the GRAPE board (up to 96). The time dependence of this operation with respect to $N_i$, the number of particles sent, is then a simple step function. Fig. 3.7 shows this dependence, for $N_p = 62$.

Besides the local force computation, a global communication is also needed for the parallel GRAPE code, as the total force computation requires a global sum. The execution of this operation does not differ between the GRAPE and non-GRAPE codes. Measurements of this operation from real runs show a communication time linear in $N_i \cdot \log P$, as shown in table 3.1.

$i$-particle update. The final operation, i.e. orbit integration, updating and storing of the particles' physical quantities, is a linear function of $N_i$, with no dependency on $P$, since every
Figure 3.6: GRAPE related tasks. Timings of the $j$-particle send task as a function of the workload are shown. Measurements of the communication tasks show some occasional spike due to external processes, e.g. operating system function calls. A GRAPE with 62 pipelines is used for these measurements.

Figure 3.7: Same as fig. 3.6; here we show timings of the $i$-particle send task.
3.3. MODEL IMPLEMENTATION

3.3.2 Treecode

The treecode, introduced in section 1.4.2, is widely used in Computational Astrophysics for the simulation of systems that do not require high computational accuracy. By trading lower accuracy with higher speed, the treecode is able to reduce the computational complexity of the N-body problem from the $O(N^2)$ scaling of the direct code to $O(N \log N)$.

The treecode computes force on a given particle $i$ by grouping particles in larger and larger cells as their distance from $i$ increases, force contributions from such cells being truncated multipole expansions. A simple pseudo-code sketching the basic tasks of the treecode is given in fig. 3.9.

The first task of a treecode iteration is to build a tree structure by hierarchically connecting each cell to the "child" cells that the cell encompasses (see section 1.4.2 for details). Then force is computed for each $i$-particle by traversing the tree, and looking for cells that satisfy an appropriate acceptability criterion (see section 1.4.2 and chapter 4 for details on acceptability criteria).

The original treecode algorithm has been modified in several ways to improve its performance. An optimisation of the tree traversal phase has been realised by grouping particles according to their spatial proximity (Barnes, 1990). Then a single traversal for each group is performed, whereas the original algorithm performs a tree traversal for each particle. This drastically reduces the number of tree traversals, and allows for concurrent force computa-

Figure 3.8: Same as fig. 3.6; here we show timings of the force computation task. The pipeline startup latency $\mu_s \approx 75\mu s$ is clearly visible.

processor performs this task for all the $i$-particles.
This optimisation is also suited for the use of the treecode with GRAPE, because each pipeline of the array contained in a GRAPE board can compute force on a different particle simultaneously. The drawback of this technique is an increase in memory use. In fact, for each particle group, an interaction list containing the information concerning all the cells interacting with the group must be written and stored in memory.

The use of interaction lists is also useful for parallelisation on distributed systems, as in the parallel treecode (Warren & Salmon, 1993, 1995). The possibility of decoupling tree traversal and force computation through interaction list compilation, allows for the implementation of latency hiding algorithms for the retrieval of cell information stored in a remote processor memory (Warren & Salmon, 1995; Salmon & Warren, 1997). We will refer to this version of the parallel treecode as HOT, the acronym of Hashed Oct-Tree, as the code was called by Salmon and Warren.

Another modification, introduced in the code GADGET (Springel, Yoshida, & White, 2001), consists in implementing the individual time step scheme, originally introduced in the direct N-body code, as described earlier in sections 1.4.1 and 2.3.1. In this manner, each particle is assigned an individual time step, and at each iteration only those particles having an update time below a certain time are selected for force evaluation (Springel et al., 2001), so that force is computed only on a small fraction of the $N$ particles. In this code, a different approach for remote interactions computation is also implemented: data of the selected particles are sent to the remote processors, interactions are computed remotely, and results are received back. A further modification consists in rebuilding the local tree less frequently than at every iteration. This version will be referred as GDT, which is a short for GADGET. In fig. 3.10 we give a pseudo-code representation of the generic algorithm that our model simulates.

Application model

Many different versions of the treecode have been proposed, implementing different tools and techniques. A recent report on this is given in (Springel et al., 2001). Our performance model is designed to reproduce the behaviour of state-of-the-art parallel treecodes, running on distributed architectures, and able to make use of dedicated hardware. In this section, we describe each task of our application model, together with their modelling expressions.
t = 0
while (t < t_end)
    if code is GDT
        if it is time to rebuild tree
            build local tree
        else if code is HOT
            build local tree
            exchange data to build global tree
    if code is GDT
        for each selected particle
            traverse local tree to compute local forces
            send particles to remote nodes
            receive particles from remote nodes
            compute force on remote particles
            send forces to remote nodes
            receive forces from remote nodes
    else if code is HOT
        for each group
            build interaction list
            (communication needed for remote data retrieval)
            for each group
                for each particle in group
                    compute forces
            integrate orbits
    t = t + Δt

Figure 3.10: Pseudocode sketching the generic parallel treecode tasks. HOT and GDT are the two versions of the treecode modelled in this work. Tasks involving communication are highlighted using a grey background.

Table 3.3 shows a synopsis of the modelling expressions, given as functions of the appropriate application parameters.

Tree building. The tree building task consists of two operations: particle insertion into the tree structure, and computation of multipole terms for each cell of the tree. Both operations, as long as local trees are concerned, do not require communication among processors. The particle insertion operation scales as \( n \log(n) \), where \( n \) is the number of particles per processor. The multipole terms computation depends linearly on the number of cells per processor \( n_c \), which is set equal to 0.1\( n \) divided by the number of particles per leaf cell. Each computing node processes all the particles independently to build a local tree. When
### Chapter 3. Modelling and Simulation of Hybrid Architectures

<table>
<thead>
<tr>
<th>task</th>
<th>parameter definition</th>
<th>modelling formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>build local tree</td>
<td>( n ): number of particles per processor</td>
<td>( n \cdot \log(n_c \cdot n) + )</td>
</tr>
<tr>
<td></td>
<td>( n_c ): number of cells per processor</td>
<td>( n_m \cdot n_c \cdot n_{mp} )</td>
</tr>
<tr>
<td></td>
<td>( n_{mp} ): operations per cell to compute multipoles</td>
<td></td>
</tr>
<tr>
<td>data exchange for HOT global tree</td>
<td>( P ): number of processors</td>
<td>( \pi g \cdot P )</td>
</tr>
<tr>
<td>tree traversal</td>
<td>( m ): fraction of particles selected for force computation (= 1 if code is HOT)</td>
<td>( \pi u \cdot m \cdot n_g \cdot j_{loc} )</td>
</tr>
<tr>
<td></td>
<td>( n_g ): number of groups per processor</td>
<td>( j_{loc} = K_j \cdot \theta^{-3} \cdot \log(\theta^3 \cdot n) )</td>
</tr>
<tr>
<td></td>
<td>( j_{loc} ): number of local force sources per group</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \theta ): opening angle (accuracy parameter for the force computation)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( K_j ): scaling coefficient for force sources</td>
<td></td>
</tr>
<tr>
<td>data exchange for HOT global lists</td>
<td>( j_{rmf} ): number of sources per group from remote processors</td>
<td>( \pi g \cdot j_{rmf} \cdot n_g )</td>
</tr>
<tr>
<td></td>
<td>( j_{rmf} ): number of sources per group from remote processors</td>
<td>( j_{rmf} = K_j \cdot \theta^{-3} \cdot \log P )</td>
</tr>
<tr>
<td>compute forces</td>
<td>( j ): total number of force sources per group</td>
<td>( \pi_{cf} \cdot m \cdot n \cdot j )</td>
</tr>
<tr>
<td></td>
<td>( N ): total number of particles</td>
<td>( j = K_j \cdot \theta^{-3} \cdot \log(\theta^3 \cdot N) )</td>
</tr>
<tr>
<td>data exchange for GDT remote forces</td>
<td>for parameter definition see above</td>
<td>( \pi_{rj} \cdot m \cdot N \cdot \log P )</td>
</tr>
<tr>
<td>remote i-particle data sent to GRAPE hosts</td>
<td>for parameter definition see above</td>
<td>( \pi_{gr} \cdot (m \cdot n + j \cdot n_g) )</td>
</tr>
<tr>
<td>force data sent back by GRAPE hosts</td>
<td>for parameter definition see above</td>
<td>( \pi_{gs} \cdot m \cdot n )</td>
</tr>
<tr>
<td>integrate orbits</td>
<td>for parameter definition see above</td>
<td>( \pi_{or} \cdot m \cdot n )</td>
</tr>
</tbody>
</table>

Table 3.3: Synopsis of the modelling expressions for each task of the application model. All \( \pi \) terms are constant factors depending on the operations per particle performed, or the bytes per particle transmitted. Communication task expressions are highlighted using a grey background. Parameter values are given in table 3.4.
the tree building task is accomplished, the local tree contains all the particles which are located within the geometrical domain assigned to the processor. GDT uses only local trees for the force computation task, hence it does not execute other operations to complete the tree building task after the local tree building. Conversely, the HOT code exchanges information among the processors after the local tree building, so that each processor is able to build a global tree. In this way, no communication will be necessary during the force computation task. The data exchanged to build the global tree, the so-called local essential tree, are assumed to be equal for each processor, so that this operation is assumed to scale linearly with the number of processors $P$.

Tree traversal. This task is performed by HOT before the force computation, while GDT performs, for each $i$-particle, tree traversal and force computation as the same task. Namely, HOT first traverses the tree in order to build an interaction list for each group of nearby particles (an input parameter states how many particles make up a group), then uses the list to compute forces on each particle of the group (see also section 3.3.2 above). GDT instead, for each particle selected for force computation, traverses the tree and computes force simultaneously. The local tree traversal has to be done once for each particle group (GDT does not use groups, so in this case the number of groups is equal to the number of particles). It depends linearly on the number of local force sources. An expression for the total number of force sources $j$ was found by Makino (Makino, 1991b), who gives

$$j \propto \theta^{-3} \log(\theta^3 N),$$

where $\theta$ is the opening parameter (see equation 1.3 in section 1.4.2). The number of local force sources is then $j_{loc} \propto \theta^{-3} \log(\theta^3 N/P)$, and the number of remote force sources is $j_{rm} = j - j_{loc} \propto \theta^{-3} \log P$. GDT performs this operation only for a fraction $m$ of the particles per processor $n$. In the case of HOT, we simply set $m = 1$.

The HOT code completes this task with a communication operation, where information concerning remote force sources is received by each processor. This operation depends linearly on the number of groups, and on the number of remote force sources $j_{rm}$.

Force computation. The cost of the force computation task on each processor is proportional to the number of $i$-particles per processor $m \cdot n$, times the number of force sources $j$. For the HOT code, this task does not require communication, since all information about remote force sources has been exchanged in the tree traversal task. In the GDT case, local $i$-particles are sent to the remote processors, then remote partial forces are retrieved to finally obtain the total force on each $i$-particle. This operations are global communication operations, and are assumed to depend on $\log P$, consistently with the modelling formulae of the global communication tasks of the direct code model in section 3.3.1 (see also table 3.1). This task also depends linearly on the total amount of data exchanged, i.e. on the total number of $i$-particles $m \cdot N$.

If GRAPEs are used, the force computation task is performed only by the GRAPE hosts. As a consequence, the communication operation executed in the HOT case during the tree build task in order to build the global tree, is executed only by the GRAPE hosts. As far as the actual force computation is concerned, the GRAPE hosts first compute force on their local particles, then receive remote $i$-particles and corresponding interaction lists from
### Table 3.4: Values of the performance parameters appearing in the modelling formulae in table 3.3

The “un-graped” processors, compute force on remote \(i\)-particles, and finally send back forces to the remote processors.

The actual force computation operation on the GRAPE is modelled using the same expression as in table 3.1, where the number of \(i\)-particles is in the present case put equal to the number of particles per group \(n/n_g\), and the number of \(j\)-particles \((N/\text{GRAPEs in the formula in table 3.1})\) now is equal to the total number of force sources per group \(j\).

The cost of the communication operation to send remote particle data to the GRAPE
host is proportional to the amount of bytes which are sent, which is proportional to the number of \(i\)-particles sent, plus the length of the correspondent interaction lists. Hence it is modelled as a linear function of the number of \(i\)-particles \(m \cdot n\) plus the number of force sources for all groups \(j \cdot n_g\). The cost of sending back the forces is proportional to the number of \(i\)-particles \(m \cdot n\).

**Orbit integration.** This task consists in the updating of the \(i\)-particles positions, and does not require communication. It is modelled as a linear function of the \(i\)-particles per processor \(m \cdot n\).

**Computer architecture**

The parallel system simulated in our machine model is a generic distributed multicomputer, where given nodes can be connected to one or more SPDs. When SPDs are present, the appropriate task is executed on them. The application model needs no modification in this case. According to an input parameter which tells whether SPDs are present, the mapping interface chooses the routine that maps the task to the SPD, or to the general purpose processor. Since we are interested in SPDs dedicated to the gravity force computation, the machine model of the SPD reproduces the GRAPE activity, and its communication with the host. The modelling of the fairly complicated data exchange machinery between GRAPE and its host is discussed in section 3.3.1.

The hardware characteristics of the simulated multicomputer are parameterised by two constants, \(\mu_P\) and \(\mu_N\), where \(\mu_P\) accounts for the processor speed, in nanoseconds per floating point operations, and \(\mu_N\) accounts for the network speed, its value being the transfer rate in \(\mu s/B\). In the execution model, each computation-related function will be multiplied by \(\mu_P\), and each communication-related function (those highlighted with a gray background in table 3.3) will be multiplied by \(\mu_N\). Parameter values for the simulations presented in section 3.4.3 below are given in table 3.4.

**3.4 Simulations**

In section 3.3 the modelling of the various application tasks of the \(N\)-body codes that we study have been described. In this section, we show how our models reproduce the real system, and simulate possible modifications. The models consist in a sequence of tasks, as described in section 3.3, each one specifying, by means of appropriate `delay` operations, how much wall-clock time is spent to perform them. The access to GRAPE is controlled by a semaphore. Model results are compared with data obtained from the performance analysis study of our system reported in section 2.4.
3.4.1 Serial direct code

Validation

The reliability of our model has been checked by making a comparison between the simulation results, and the actual measurements of a set of runs of NBODY1 on a system consisting of a GRAPE connected to its host, which is a node of the DAS, as described in section 2.2.1. Two separate series of runs have been performed, on each GRAPE at our disposal. The GRAPE board with 62 pipelines is labelled GRAPE0; the other board, with 94 pipelines, GRAPE1. Each run consists of 300 iterations, with $N$ ranging from 1024 to 32768. The initial condition is a Plummer model (Plummer, 1915), i.e. a star distribution with density decreasing as the fifth power of the distance from the cluster centre (see, e.g., Spitzer, 1987, p. 13). In section 3.3.1 we compared measurements with simulation for single tasks. Now, in fig. 3.11 and 3.12, we present a global comparison, where we show how each task scales with $N$. In these figures we plot the time share spent by the application in accomplishing each task. These figures show that our model produces results in good agreement with the real measurements. These measurements are presented in section 2.4.3.

It can easily be seen how the system performance is strongly penalised by communication overhead, unless the workload is high (i.e. $N \geq 16384$). Even in such cases, GRAPE is not fully exploited yet, due to the large time-share taken by host computations. It is clear from this that a faster host and an improved communication interface are needed to achieve an optimal GRAPE utilisation. A comparison between fig. 3.11 and 3.12 shows that the time share for the force computation is smaller for GRAPE1. This is due to the higher number of
3.4. SIMULATIONS

Fig. 3.12: Model validation. Same as fig. 3.11. GRAPE1 refers to the GRAPE where the number of pipelines is 94.

pipelines in this GRAPE board, which makes the force computation faster. See also fig. 2.14 and 2.15 on page 45, where the same performance data for the real system are presented.

To test the versatility of our model, we also validated it with respect to a system configuration without GRAPE. In this case the mapping interface of the model, instead of selecting the procedure where the use of GRAPE is modelled, maps the force computation task on a different procedure, where the force computation task is modelled as a linear function of $N_i \cdot N$. The user selects whether the force computation will be modelled as a host related task, or as a task involving the use of GRAPE, simply by changing an input parameter. Fig. 3.13 shows the task time shares, while fig. 3.14 shows the total execution time for the application that does not make use of GRAPE. Measurements data and simulation results are compared.

Fig. 3.13 shows that the force computation task dominates the system activity. From a comparison with fig. 3.11, where the force computation share is remarkably smaller, it becomes clear how effective is GRAPE in optimising this task. In fig. 3.14 the execution time for the code that uses GRAPE1 is also plotted to show how large is the speedup achieved thanks to the GRAPE.

Predictions

The above discussion highlights the need for a faster host and communication interface. Our model has been used to forecast the benefit obtainable by operating such improvements.

We modified our model to simulate a host twice as fast as our present host, and with
Figure 3.13: Model validation for a system not using GRAPE. Since the force computation time share for this code is by far larger than the other task shares, the y-axis scale has been changed to make data more readable.

Figure 3.14: Model validation for a system not using GRAPE. Comparison of the total execution times of the GRAPE system and the non-GRAPE system.
3.4. SIMULATIONS

Figure 3.15: Simulated time shares for a system with a faster host and communication interface, but with the same GRAPE board as GRAPE1.

communication performance two times faster as well. In this manner we try to reproduce the system measured in Kawai et al. (1997), consisting of a DEC workstation 500MHz, using a GRAPE board with 94 pipelines of the GRAPE-4 cluster. The time-shares, and a comparison of the estimated performance gain of the simulated system with respect to the system described in the previous section are given respectively in fig. 3.15 and 3.16.

These figures show that the GRAPE board is used more efficiently now, and the overall system performance benefits of this. Nevertheless, it appears that when the workload is high, this performance gain decreases. This is predictable, since in this case the relevance of the host and the communication interface is not as large as with a lighter workload. The estimate in Kawai et al. (1997) is in agreement with ours, it only attributes a larger time share to the host computation tasks at small values of $N$. This discrepancy can be explained considering that they model $N_i = 1.6 \cdot N^{1/2}$, with no oscillation. In this way, when $N < 3600$, $N_i$ is always smaller than 96, i.e. the maximal number of pipelines in a GRAPE board. Now, since the force computation and communication step is always done in a single iteration, the relevance of GRAPE and the communication interface is reduced. Conversely, if the value of $N_i$ can oscillate and assume values greater than the number of pipelines in a GRAPE board, as it happens in our model, two or more iterations are necessary, increasing the relative load of the communication and GRAPE computation tasks.

Another use of our model is the estimation of the performance gain that can be reached by improving the communication operations. As mentioned above, the $i$-particle send and

\footnote{More precisely, the performance is set to be two times faster for the send operations, and five times faster for the receive operations, in order to reproduce in any aspect the performance figures given in Kawai et al. (1997).}
CHAPTER 3. MODELLING AND SIMULATION OF HYBRID ARCHITECTURES

Figure 3.16: Simulated performance gain for a system with a faster host and communication interface, but with the same GRAPE board as GRAPE1.

the result retrieval are performed in a buffered fashion, where data are sent for a number of particles always equal to \( N_p \), also when \( N_i \) is less than \( N_p \). For the result retrieval, the situation is even worse; in that case, the operation is performed always for 96 particles, i.e. the maximal number of GRAPE pipelines.

One may then wonder how much the system performance could benefit, if this communication protocol is improved. We simulated a situation where the send and receive operations are accomplished by transmitting a variable size packet of up to \( N_p \) particles per time, in a fashion similar to the \( j \)-particle send. However, an extra amount is added to the packets to represent the defective pipelines in the GRAPE board. Every GRAPE board includes 96 pipelines, but some of those can be defective. We assume the worst-case situation, in which the \( N_d = 96 - N_p \) defective pipelines are the first ones to be accessed. Assuming that data regarding \( N_i = \tilde{N} \) particles have to be transmitted, the operation that we model consists in writing to (or reading from) the first \( \tilde{N} \) non-defective pipelines. Then our packet contains \( N_d + \tilde{N} \) items.

The results of this simulation are shown in fig. 3.17 and 3.18. The simulated board has 94 pipelines working, and 2 defective. It can be seen how little influence the discussed modification has on the system performance. In this case, the model forecast discourages the enterprise of implementing such a modification in the real system. This example shows how performance modelling can be useful in evaluating whether a new project is promising, or, as in this case, it is likely to be unsuccessful.
Figure 3.17: Simulated time shares for a system with modified communication operations. The GRAPE board is the same as GRAPE1.

Figure 3.18: Simulated performance gain for a system with modified communication operations. The GRAPE board is the same as GRAPE1.
3.4.2 Parallel direct code simulations

Validation

In order to check the validity and the versatility of our model, we compare our simulation results with the performance analysis data presented in section 2.4. In the first case, we consider a situation with 2 processors, each one connected to a GRAPE; in the second case, we scale up to 24 processors. This corresponds to the architecture at our disposal, described in section 2.2.1. Each run, either real or simulated, consists of 300 iterations, with $N$ ranging from 1024 to 16384 for the non-GRAPe case, and to 32768 for the GRAPE case. As initial condition in the real runs we used, as in the serial case, a Plummer model (Plummer, 1915).

Timing results for the overall execution time are presented in fig. 3.19. We show results for the non-GRAPe case, with two different values of $P$, and for the GRAPE case with the two GRAPEs each attached to its own host. The ability of our model to fit the measurement values can be readily inferred from this figure. The worst case error amounts to $\simeq 40\%$ for the non-GRAPe case with $P = 24, N = 2048$ (a case that lies well outside the parameter range that we are really interested in), whereas the average error is $\simeq 10\%$.

Besides the overall timing, we also show the fraction of time spent to perform each task. Fig. 3.20 and 3.21 are for the non-GRAPe case, and fig. 3.22 is for the GRAPE case. It can be seen in figures 3.20 and 3.21 how the different application tasks scale with $N$, i.e. with the total workload. For the case with $P = 24$, shown in fig. 3.21, a large communication overhead at low values of $N$ is visible.

The large share of execution time taken by the force computation task is a clear evidence...
Figure 3.20: Model validation for the parallel non-GRAP-E system. For each task, the ratio $t_{task}/t_{tot}$ is shown, as a function of $N$. Here a system with two processors is shown. Points are measurement data from test runs, lines are simulation results. The scale on the y-axis does not start from 0 (cf. fig. 3.13 for the serial case).

Figure 3.21: Model validation for the non-GRAP-E system. Same as fig. 3.20. In this case a system with 24 processors is shown.
of the need for a tool to accomplish this task faster. Fig. 3.22 shows how effectively GRAPE solves this problem. The relative importance of the force computation task has been drastically reduced by using GRAPE, even though at the cost of a large communication overhead with the SPD.

**System comparison.** Fig. 3.19 shows that the GRAPE system is two orders of magnitude faster than the non-GRAPE system having the same number of processors, while with respect to the most powerful non-GRAPE configuration available, the one with 24 processors, the performance gain is still about one order of magnitude. Our model can reproduce the behaviour of both systems quite satisfactorily. In the following, we use our model to predict how this behaviour changes as a consequence of system modifications.

**Predictions**

In this section we present some examples of the use of our model in order to predict the performance of systems where either hardware or software modifications have been carried out. Performance estimation and algorithmic design are the main fields of application that our simulation model is designed to serve.

**Clustered GRAPEs vs distributed GRAPEs.** A fundamental question that we want to answer is whether it is more efficient to connect several GRAPEs to the same host node, or to have a network with several nodes, each one being the host of a GRAPE. The first
configuration, which also reflects the original system architecture for this device,\(^4\) does not exploit a multiprocessor host in order to perform the particle search in parallel but, on the other hand, does not incur any overhead cost for the two global communications required by the parallel code. The configurations that we simulated contain 1, 2 or 4 GRAPEs, either connected to one single host, or distributed one GRAPE per host. The result of our simulation is shown in fig. 3.23. The total number of particles is increased up to 32768 particles per GRAPE board. It can be seen that the performances are almost equal for all cases having the same number of GRAPEs. The gain for the multiprocessor configuration to distribute the local search, is roughly of the same amount as the loss due to communication overhead. It can be inferred that both hardware configurations analysed here, i.e. localised SPDs versus distributed SPDs, perform about equally well, the single host configuration performing slightly better.

A more realistic \(N\)-body code. The code modelled in the preceding sections is a basic \(N\)-body code. State-of-the-art astrophysical codes will contain additional functionality, e.g. to model close encounters to binary stars and the evolution of stars. In state-of-the-art direct \(N\)-body codes, a binary star is treated as a single entity. When a third star approaches the binary, the motion of the two components of the binary, plus the encountering star, is resolved analytically, by means of a rather complex procedure (Funato et al., 1996). This additional functionality must be provided by the host, leading to an additional workload. In this case,

\(^4\)I.e. the GRAPE-4 system at the University of Tokyo, consisting of 36 GRAPE boards connected, in a hierarchical fashion, to a single workstation (see section 1.3 and fig. 1.5).
it is interesting to see when the multiple host configuration begins to outperform the single host configuration. This extra task is assumed to be linearly dependent on $N_i$ through a coefficient $\eta$, and perfectly parallelisable. This last condition is likely to hold in reality as long as $P$ remains reasonably small.

We simulated the case with $N = 32,768$, and compared the "clustered" case where one processor hosts four GRAPEs to the "distributed" case where four processors host one GRAPE each. From the previous section, the clustered configuration is faster when the algorithm without the extra task is used. The results are shown in fig. 3.24. We can see that the configuration with $P = 4$ begins to perform better at $\eta \approx 0.01$, when the time spent in the extra task is still negligible compared to that spent in the force task (at least for the distributed case). In order to compare $\eta$ with the parameters reported in table 3.2, we have to divide it by the processor time cycle, which is $0.5 \cdot 10^{-8}$ ms. This results in $\tau_\eta \approx 2000$, i.e. about ten times the value of the extrapolation or local force task parameter in table 3.2. The amount of computations for the close encounters procedure mentioned above, is of this order of magnitude. This example shows that the multiple host configuration is more appealing because of its better performance potential.

**Distributed GRAPEs load balancing.** A problem in the distributed GRAPEs configuration is the load balancing of the force computation task. When the number of pipelines per board is not the same for all the boards, the boards with the highest number of pipelines are faster in performing the force computation, because they handle more $i$-particles per unit time. In our case, the idle time is up to 10% of the total averaged time of the two

![Figure 3.24: System performance comparison when an extra task, modelled as $t = \eta \cdot N_i/P$, is added, where $\eta$ is an experimental factor of proportionality. In this case is $N = 32,768$.](image)
GRAPE boards, which means that the fastest board is idle for about 20% of the time. For the sake of readability, the idle time was not shown explicitly in fig. 3.22, but was included in the communication with GRAPE task. We used our performance model to find the optimal partitioning of particles between the two GRAPEs in the distributed configuration, where each GRAPE is connected to its own host.

Fig. 3.25 shows the result of this study. The configuration analysed here includes a GRAPE with 62 pipelines, called G0, and a GRAPE with 94 pipelines, G1. The total number of particles is 32 768. The figure shows how the total execution time changes, as the number of $j$-particles on G1 is increased in order to better exploit its higher computational potential. The optimal distribution, i.e. the one with the minimal idle time, is not reached when the ratio $r$ between the $j$-particles on the two GRAPEs is equal to $94/62 \approx 1.52$, i.e. the ratio between the pipelines, but at a slightly higher value of $r$. Unbalance between the two GRAPEs is due to both computation and communication. When $r = 94/62$, only computation is balanced. The host-GRAPE communication time is faster for G1, because the time cost of the receive result operation is inversely proportional to the number of pipelines (see table 3.1). Then, at $r = 94/62$, G1 is still faster than G0, because of its better communication performance. A further slight overload of G1, such that $r \approx 1.6$, balances the overall execution time of the two devices.

This somehow unexpected result, produced by the complex dependence of the computation and communication tasks on $N_i$ and $N_j$, illustrates how a detailed simulation is useful to analyse the behaviour of a hybrid architecture.
Table 3.5: Comparison of the timings breakdown between the real measurements and the simulation results. Timings are in seconds for a single code iteration. The HOT on Delta case refers to a 8.8 million particle run on 32 i860 40 MHz processors; GDT on T3E refers to a 0.5 million particle run on 16 Alpha 300 MHz processors, and seq. tree on GRAPE-5 to a one million particle run on a 500 MHz Alpha processor connected to a GRAPE-5 board.

3.4.3 Parallel treecode

Model validation

We present here the result of running our simulation model of the treecode, described in section 3.3.2. We use this model with parameter values such that performance measurements reported in the literature are reproduced. We show for each case the scaling with the total particle number $N$ of each task of the code, compared with the corresponding real system timings, as reported by the measurements authors. Finally we present a plot comparing the total compute time for a code iteration of each configuration. We had to deal with the fact that in most cases data were available only for one measurement run. Therefore a conclusion on the ability of our performance model to reproduce the scaling behaviour of the simulated system can only be incomplete from these data. The partial information that we obtain from this work is nevertheless fundamental to provide us the main guidelines for the realisation of a parallel environment for the simulation of $N$-body systems, as reported in the section on model forecasts. Once this environment will be realised, we will be able to validate our performance model thoroughly, having a system of our own to carry out performance measurements. In table 3.5 we show the comparison between the timings breakdown of the real measurements and our simulation results. In fig. 3.27 the compare the global timings of the various cases.

**HOT on Touchstone Delta.** The Touchstone Delta was a one-of-a-kind machine installed at Caltech in the early nineties. It consisted of 512 i860 computing nodes running at 40 MHz, and connected by a 20 MB/s network. The performance measurements reported in (Warren & Salmon, 1993) are based on a run using the whole 512 nodes system, and consist in a timing breakdown of a code iteration taken during the early stage of evolution of a cosmological simulation, when the particle distribution is close to uniform. The total number of particles is $N = 8.8 \cdot 10^6$. Implementation limitations prevented our performance model to simulate 512 concurrent processes, so that we limited our simulation to 32 processes, and scaled
Figure 3.26: Timings of the GDT tasks. The real system timings are also reported. The hardware architecture is a Cray T3E. Performance scaling with the number of processors, with $N = 500,000$. Down 16-fold the measured compute time reported in (Warren & Salmon, 1993). Since the communication overhead for that run was just $\approx 6\%$, we assumed a linear scalability of the code. The timings breakdown of our simulation is presented in table 3.5. The real system measurements are reported for comparison.

The table shows that the force computation task and the tree traversal are the most expensive tasks. The relative computational weight of each task is qualitatively well reproduced by our model. Quantitatively, a large discrepancy between our model and the real system timings originates from an over-estimation of the tree traversal and the force computation tasks, which also results in over-estimating the total time, as shown in fig. 3.27. We must conclude that in this case our model is not sufficiently well matched.

**GDT on T3E.** This case reproduces the configuration described in (Springel et al., 2001), where the GADGET code is run on the T3E hosted at the supercomputing centre in Garching, Germany. Each computing node has a frequency of 300 MHz, and the communication network has a throughput of 500 MB/s. Three cases are reported in (Springel et al., 2001), each running the same cosmological simulation, where a system of 500,000 particles is evolved for 3350 time steps. The difference among the three cases is in the number of processors used. Since in this case measurements from three different hardware configuration are reported, we could compare our model results with a larger set of timing values. As reported in (Springel et al., 2001), we assumed that only 5% of the particles are selected on average at each time step for force computation. Similarly, we assumed that the local tree is rebuilt each 10 time steps. Timing breakdowns are shown in fig. 3.26 and in table 3.5.

Fig. 3.26 shows the performance gain as the number of processors increases. The trend
in the measurements suggests a saturation in the attained performance, arguably due to an increasing load imbalance. This trend is not visible in our model results, because load imbalance is not modelled. Data in table 3.5 show that, with respect to the HOT case, now the tree build task is more expensive, despite the fact that it is performed only every ten iterations. This is to be expected, since the tree build task is performed for all particles, while the tree traversal and force computation tasks are performed only for a small fraction (5%) of the selected particles. Also in this case our model results match the real system timings. Springel et al. (2001) did not provide separate values for the tree traversal and the force computation tasks, so that only the aggregate value can be reported on the plot.

**Sequential treecode on GRAPE-5.** Here we simulate the configuration described in (Kawai et al., 2000). In that case, a modified treecode is used to simulate a system containing one million particles, and groups of $\approx 2000$ particles share the same interaction list. This code is run on a Compaq workstation with a 500 MHz Alpha 21264 processor, connected to a GRAPE-5 board containing 96 virtual pipelines, each one able to compute a force interaction in 75 ns. Estimating a force interaction as 30 flops, the aggregate performance of a GRAPE-5 board is 38.4 Gflop/s. Table 3.5 shows the results of our simulation model, compared with the real system timings, as reported in Kawai et al. (2000).

In this case, the force computation task is performed by the GRAPE. An important fraction of the total timing is taken by the communication between the host and the GRAPE. The decrease of importance of the tree traversal task, due to the particle grouping technique, is clearly observable.

**Cases comparison.** We compare here the three cases presented above. We show in fig. 3.27 a plot of the time taken by a code iteration versus $N$, as obtained from our simulation model, compared with the real system measurements. The value for the HOT code on the Touchstone Delta is 16 times greater than the value reported in Warren & Salmon (1993), in order to scale their 512 processor run to our 32 processor simulation. Conversely, scaling our simulation data for 32 processors to 512 processors, would have resulted in simulation values overlapping the values for the GRAPE case.

The simulation values match within approximately a factor 2 the real system measurements. In the next section we present results of a performance simulation, where our model is used to forecast the behaviour of other configurations.

**Model forecasts**

In this section we explore the possibility of using a hybrid architecture consisting of a distributed general purpose system, where single nodes host zero or more GRAPE boards. We span the two-dimensional parameter space defined by the two quantities $P$, the number

---

5 A GRAPE-5 board contains in fact 16 physical pipelines, each one running at 80 MHz, which is 6 times the speed of the board bus. The board "sees" $16 \times 6 = 96$ logical pipelines, running at 80/6 MHz. Appropriate hardwiring manages the data exchange between the pipelines and the board.
Of nodes, and \( G \), the number of GRAPEs. We assign to those quantities values as follows: \( P \in \{1, 2, 4, 8, 12, 16, 20, 24\} \), \( G \in \{0, 1, 2, 4, 8, 12, 16\} \). We simulate the same software configuration as described in the previous section with respect to the case related to the sequential treecode on GRAPE-5. The SPD we simulate in this case is the GRAPE-4 (Makino et al., 1997), whose performance per board is 30 Gflop/s, comparable to GRAPE-5’s. It provides a higher accuracy with respect to GRAPE-5, and is used in fields as Globular Cluster dynamics on Planetesimal evolution (Hut & Makino, 1999), where high computing precision is required. The general purpose nodes are assumed to perform a floating point operation in 22 ns, and the communication network is assumed to have a 100 MB/s throughput.

When a node of the distributed system is a GRAPE host, forces on its local particles can be computed on the GRAPE that it hosts. Forces on particles residing on nodes that do not host GRAPEs can be computed on remote GRAPEs, provided that both particle positions and particle interaction lists be sent to the appropriate GRAPE host. This implies a very large communication traffic. With our simulation we try to evaluate the effect of this communication overhead.

Fig. 3.28 shows our results. It is clear that, as long as all nodes are connected to one or more GRAPEs, a significant performance gain is obtained. For comparison, we also provide timings of a system without GRAPEs. When not all nodes are GRAPE hosts, the very large communication overhead due to sending particle and interaction list data is disruptive for performance. This result suggests that the communication task needs a very careful analysis, in order to design an efficient parallel treecode for hybrid architectures. Here we assumed that an “un-graped” node sends all its data to a single “graped” node. We
CHAPTER 3. MODELLING AND SIMULATION OF HYBRID ARCHITECTURES

discuss this point further in the next section. The plot in fig. 3.28 also features an oscillatory behaviour, particularly evident in the case with 8 GRAPEs. The local minima (i.e. better performances) correspond to configurations where \( P \) is an exact multiple of \( G \). In this case the computational load on the GRAPEs is perfectly balanced, whereas in the other cases some GRAPE bears a higher computational load from remote data.

3.4.4 Direct code vs treecode

A main goal of our research is to develop a distributed hybrid architecture optimised for the treecode. The treecode does not compute all particle-particle interactions directly. Instead, it computes partial forces on a given \( i \)-particle from a truncated multipole expansion of groups of particles, see section 1.4.2. The force interaction from a group is computed if the group is far enough, according to a Multipole Acceptability Criterion (MAC). Groups become larger and larger as their distance from the \( i \)-particle increases. This technique allows a decrease in the computing time of the force evaluation to \( \mathcal{O}(N \log N) \), at the cost of a reduced accuracy, due to the truncated multipole expansion. Moreover, this asymptotic performance is reached for large values of \( N \). Because of this, the treecode is widely used to simulate systems like clusters of galaxies, or large scale structures, where high accuracy is not needed, and \( N \) is large.

For a sufficiently large problem a treecode can outperform a direct code also for the simulation of systems that require high accuracy. In order to increase the treecode accuracy, we can tune two parameters: the highest term of the multipole expansion, and the MAC
parameter that decides if a group is far enough to compute the interaction. As already discussed in section 1.4.2, the most widely used MAC (see, e.g., Barnes & Hut, 1986) states that a multipole expansion is accepted if

\[ \frac{l}{d} < \theta \]  

(3.1)

where \( l \) is the size of the cell containing the group, \( d \) is the distance of the \( i \)-particle from the cell, and \( \theta \) is the MAC parameter. For the low accuracy computations that usually involve the treecode, is \( \theta \lesssim 1 \). A more accurate code will run more slowly.

In order to have a higher accuracy code, \( \theta \) has to be smaller. A realistic choice for a multipole expansions up to the quadrupole term, is \( \theta = 0.2 \). For a comparable accuracy with a multipole expansion up to the octupole term, we have to set \( \theta = 0.5 \) (McMillan & Aarseth, 1993). There is a trade-off between the two choices. A smaller \( \theta \) implies a much larger amount of interactions to compute; it has been shown that the number of interactions scales as \( \theta^{-3} \) (Makino, 1991b). On the other hand, a multipole expansion up to the octupole term implies a larger number of computations to obtain the multipole terms, and a larger number of computation to evaluate the force contributions from the octupoles. With our model, we can simulate the two cases, and obtain an indication of the most effective choice. In our simulations, we assume that also the force contributions from the multipoles can be computed on the GRAPE, by means of a pseudo-particle transformation (see chapter 4). In this case, multipole expansions are converted to pseudo-particle distributions that produce the same force. In this way, GRAPE can also compute force contributions from the multipole terms.

Moreover, we compare the performance of two different parallel treecodes, i.e. the HOT and GDT codes described in section 3.3.2. The main difference between the two codes is that in HOT, i.e. the parallel treecode originally developed in Warren & Salmon (1995), each processor computes forces only on the local \( i \)-particles. Information about remote particle groups, the so-called local essential tree, is obtained before the force computation starts.

Conversely, in the GADGET code (Springel et al., 2001) (referred here as GDT), processors do not exchange information about remote particle groups. Instead, local \( i \)-particles are sent to remote processors. With our model, we can see which approach is better suited for a distributed hybrid architecture.

Our comparison has the goal to assess whether a system size exists at which treecodes outperform the direct code for the simulations of systems requiring high accuracy. Then we use our performance model to find which hardware-software combination gives the best performance, provided that high accuracy is ensured from the treecodes, either by decreasing \( \theta \), or increasing the multipole order. Therefore, we choose for each method the most suitable hybrid architecture. Namely, we simulate the direct code running on a system including a single host with 16 GRAPEs attached to it, since the clustered configuration has the best performance, as shown in section 3.4.2. As treecodes place a higher load on the host, the optimal system for them is a distributed 16 processor hybrid machine, each node hosting a GRAPE board. High accuracy from treecodes is obtained by setting \( \theta = 0.5 \) with octupole term expansion, and \( \theta = 0.2 \) with expansions up to the quadrupole term.
Figure 3.29: Performance of the treecode, compared to the direct code. We report here the simulated execution times of 300 code iterations. The direct code runs on a single host connected to a 16 GRAPE cluster, the treecodes run on a 16 processor machine, each node hosting a GRAPE. \textit{Quad} refers to multipole expansions up to the quadrupole term, with $\theta = 0.2$, and \textit{Oct} to octupole expansions, with $\theta = 0.5$. Other symbols are explained in the text.

Fig. 3.29 shows the results of our simulation. We can see in the figure how the direct code performs better for low $N$, but is eventually outperformed by the treecodes. The two treecode implementations show a very similar performance. Both perform better than the direct code for high particle numbers, and are faster when an expansion up to the octupole term is used. We can conclude that a distributed hybrid architecture can be the system of choice for the simulation of large astrophysical systems requiring a high accuracy, such as stellar globular clusters. A treecode equipped with the software tools for the accurate treatment of close encounters could supersede the direct code for the realistic simulation of phenomena such as globular cluster secular evolution, or black hole binary formation in merging galaxies (Hut & Makino, 1999).

### 3.5 Discussion

A hybrid architecture is a system with a high degree of heterogeneity among its components, whose complex interplay requires an appropriate tool in order to be understood and optimised. Performance modelling is an important tool to study the behaviour of such complex systems. We implemented and tested a simulation model able to reproduce the behaviour of hybrid architectures. We validated this model against our GRAPE-DAS system, both
for the serial and the parallel case. We showed some examples of its use for predicting the performance of other configurations, where hardware and/or software modifications have been introduced. We simulated the use of several different hardware systems, and numerical algorithms for the solution of the $N$-body problem, as the direct particle-particle code, and the treecode. We showed that performance simulation allows us to discover unexpected behaviours of a complex computer system, as in the case described at the end of section 3.4.2. In our case-studies, distributed hybrid architectures show their superior computational potential, as compared to "clustered" configurations, when large problems, at the higher limit of the available computational capability, are considered.

Our research is particularly focussed on the efficient integration of treecodes and hybrid architectures. This could lead to a very high performance computational environment for the solution of the $N$-body problem. We validated our treecode model by simulating existing configurations and comparing our results to real system measurements, even though very few measurement data were available, limiting the accuracy of our model calibration. We used our model to evaluate the performance of a hybrid architecture used to run the treecode, and highlighted that an efficient implementation of the treecode on such architecture is made difficult by an intrinsically high communication overhead. Issues like latency hiding, or partial redistribution of work to remove load imbalance, could help to solve this problem, and will be the object of further research. The model would also benefit from an accurate parameterisation of load imbalance.

In the next chapter, we describe our research in the framework of the aforementioned efficient integration of treecodes and hybrid architectures. We have implemented a version of the treecode, which makes use of pseudo-particles (Makino, 1999; Kawai & Makino, 2001) in order to represent the multipole expansion of the gravitational potential. This approach allows us to make use of the GRAPE not only for the computation of the force from the monopole term. The pseudo-particle scheme allows the GRAPE to compute force contributions from all terms of the multipole expansion. In chapter 4 the pseudo-particle method will be described, together with the accuracy and performance improvements that we introduced.