Doctoral Thesis

Adaptive parallelism in distributed shared memory environments

Author(s):
Scherer, Alex Patrick

Publication Date:
2001

Permanent Link:
https://doi.org/10.3929/ethz-a-004215733

Rights / License:
In Copyright - Non-Commercial Use Permitted
Adaptive Parallelism
in Distributed Shared Memory Environments

A dissertation submitted to the
Swiss Federal Institute of Technology Zurich
(ETH Zürich)

for the degree of
Doctor of Technical Sciences

presented by
Alex Patrick Scherer
Dipl. Informatik-Ing. ETH
born February 3, 1968
citizen of Lucerne, Switzerland

accepted on the recommendation of
Prof. Dr. Thomas Gross, examiner
Prof. Dr. Willy Zwaenepoel, co-examiner

2001
Abstract

Workstations in networks of workstations (NOWs) are sometimes little used, especially in multi-user environments. Employing their compute power for parallel processing when not used otherwise is an attractive venture, if a practical means to do so can be found. In non-dedicated NOW environments, individual machines become available or unavailable as the workstation “owner” goes away or returns, so the desired parallel processing system must be able to adapt to a continually changing pool of available nodes. Such adaptations should be transparent, allowing the user to program in a relatively standard way, without requiring special-purpose code in the application. In dedicated NOW environments, adaptation permits the job mix to be changed easily.

This dissertation claims to demonstrate that attractive solutions for such adaptive parallel processing are feasible, fulfilling the above requirements. To this end, the dissertation presents the design, implementation and evaluation of a system running on standard networked workstations. Another part of the work shows how adaptations occur transparently for the applications, such that application programmers are not burdened by new programming models.

As a basis for the practical work, a state-of-the-art software distributed shared memory (DSM) runtime system was employed, and this system was augmented to support adaptivity.

As a programming model, the OpenMP standard was chosen, because it provides for simplified and standardized parallel programming, and it is becoming widely accepted in the shared memory programming community. OpenMP frees the programmer from having to deal with lower-level issues such as the number of nodes being used, the iteration or data partitioning, or the communication of data between nodes. These properties at the same time allow for transparent adaptive parallelism, as the system alone can handle the number of processes, without intervention from the application, so standard OpenMP applications generally need no code changes to support adaptivity.

The work presented supports data parallel and task parallel OpenMP applications, thereby including a large class of parallel programming styles. No code is added to
data parallel applications specifically to obtain adaptivity, whereas task parallel applications need one minor code change for this, an extra function call at task boundaries. Often, this code modification can be done automatically by a preprocessor.

Performance results show how adaptivity only adds little overhead, and the sources of such costs are evaluated in detail. A system such as the one presented therefore enables otherwise-idle workstations in NOW environments to perform parallel processing with attractive performance while not disturbing any other potential users.
Kurzfassung


Als Basis für die praktische Arbeit wurde ein dem aktuellen Stand der Forschung entsprechendes sog. software distributed shared memory (DSM) Laufzeitsystem verwendet. Dieses System wurde erweitert zur Unterstützung von Adaptivität.


Messungen zeigen wie Adaptivität nur eine geringe Laufzeitzunahme bewirkt, und die Ursachen dieser Kosten werden im Detail untersucht. Demzufolge ermöglicht es ein System wie das vorgestellte, unbenutzte Workstations in NOW Umgebungen so einzusetzen, dass darauf parallele Berechnungen attraktiv ausgeführt werden können, ohne dabei irgendwelche potentielle andere Benutzer zu stören.
Acknowledgements

I wish to thank first and foremost my advisor Prof. Thomas Gross for his wise and encouraging guidance throughout my work, first helping me to focus my interests on distributed shared memory and later arranging the very fruitful collaboration with Rice University, all the while supervising my progress and giving valuable feedback while at the same time granting me very much freedom and patiently letting me work. I really appreciated how he always swiftly dealt with my questions despite having so many other responsibilities, and I am amazed to see how he remained closely involved in not only my work but numerous other projects as well. I must also mention how he strived to create a pleasurable atmosphere in our research group, for example with our yearly “retreat”.

I am further very much indebted to my co-advisor Prof. Willy Zwaenepoel for his positive and supportive input and for key ideas that guided the direction of my work: These include our resolve to support the OpenMP standard and to use the TreadMarks garbage collection mechanism to simplify the implementation. I enjoyed the close collaboration in 1998 during his stay in Zurich and am grateful that we were able to meet on various other occasions despite normally living on different continents.

I also wish to thank Honghui Lu very much for all her contributions and input, she did all the OpenMP-related work, in particular creating the preprocessor to translate OpenMP code to TreadMarks code, and she implemented the migration and checkpointing enhancements using the libckpt library.

I likewise wish to thank all the colleagues in our research group and in the Institute for Computer Systems for many good talks and discussions, work-related and otherwise, and simply for the very good and happy working environment that we may all enjoy.
## Contents

1 **Introduction** 1
   1.1 Adaptive parallelism ........................................... 2
   1.2 Motivation ....................................................... 3
   1.3 Background ..................................................... 4
   1.4 Challenges for adaptive parallelism ............................ 6
   1.5 Thesis ........................................................... 7
   1.6 Contributions ................................................... 8

2 **Solutions for adaptive parallelism** 9
   2.1 Migrating the work .............................................. 9
   2.2 Adjusting the work .............................................. 10
      2.2.1 Adapting anytime ........................................... 11
      2.2.2 Adapting at global synchronization points ................. 17
   2.3 Conclusions and proposal ...................................... 17

3 **Programming model** 19
   3.1 OpenMP .......................................................... 19
      3.1.1 Data parallel OpenMP applications .......................... 20
      3.1.2 Task parallel OpenMP applications .......................... 20
   3.2 OpenMP on a NOW ................................................. 21
   3.3 Transparent adaptation ........................................ 22
      3.3.1 Data parallel OpenMP applications .......................... 22
      3.3.2 Task parallel OpenMP applications .......................... 22

4 **A DSM system supporting adaptive parallelism** 25
   4.1 Functionality .................................................... 25
      4.1.1 Join and leave events ...................................... 25
      4.1.2 Multiple adapt events ...................................... 27
   4.2 Enhancements .................................................... 28
      4.2.1 Urgent leave events ........................................ 28
4.2.2 Fault tolerance ............................................. 29
4.3 Limitations .................................................. 30
4.4 User interface .............................................. 30
4.5 Usage models ............................................... 32

5 Methodology .................................................  35
  5.1 Experimental environment ............................... 35
  5.2 Classification of applications ........................... 36
  5.3 Applications .............................................. 37
  5.4 Data sharing patterns .................................... 40
  5.5 Adaptation cost measurement methodologies .......... 42
    5.5.1 Adaptation cost ...................................... 42
    5.5.2 Method A: Overview of costs ...................... 42
    5.5.3 Method B: Detailed results for single adaptations .. 43
  5.6 Experimental setup ...................................... 45
    5.6.1 Individual adaptations ............................... 45
    5.6.2 Periodic adaptations ................................. 46
  5.7 Test scenarios ........................................... 46
    5.7.1 Default test values .................................. 47
    5.7.2 Representative subset of applications ............... 48
    5.7.3 Dimensions examined ................................. 48

6 Performance overview ....................................... 53
  6.1 Non-adaptive performance .............................. 53
  6.2 Overview of adaptation effects
      on total runtime ......................................... 55
  6.3 Response time for adaptation requests ................. 57
  6.4 Selected adaptation costs .............................. 60
  6.5 Length of adaptation effects ........................... 61
  6.6 Summary ............................................... 64

7 Analysis of adaptation costs ............................... 65
  7.1 Cost components ......................................... 65
  7.2 Data movement by the system ........................... 69
  7.3 Data movement by applications .......................... 72
    7.3.1 Process identifier reassignments .................. 72
    7.3.2 Data parallel applications ........................ 73
    7.3.3 Task parallel applications ........................ 78
  7.4 Key bottlenecks ......................................... 79
CONTENTS

7.4.1 Background ................................................. 79
7.4.2 Data parallel versus task parallel computation ................. 80
7.4.3 Analytical results ........................................ 80
7.4.4 Measurements ............................................. 82
7.4.5 Empirical results ........................................ 82

7.5 Factors determining the adaptation cost .......................... 87
    7.5.1 Timing of adaptation request ............................... 88
    7.5.2 Identifier of joining or leaving thread ..................... 89
    7.5.3 Number of threads ....................................... 91
    7.5.4 Batching of adapt events ................................. 97
    7.5.5 Interval between adapt events ............................ 100
    7.5.6 Problem set size ........................................ 101
    7.5.7 Type of application and data layout ....................... 103
    7.5.8 Data sharing patterns .................................... 106
    7.5.9 False sharing ........................................... 107
    7.5.10 Execution platform ..................................... 107
    7.5.11 Summary ................................................ 108

7.6 Length of adaptation effects ..................................... 109
7.7 Evaluation .................................................... 110
7.8 Summary ...................................................... 113

8 Implementation .................................................. 115
    8.1 TreadMarks ................................................ 115
    8.2 Supporting adaptivity ....................................... 116
        8.2.1 Before an adaptation .................................. 118
        8.2.2 Performing the adaptation ............................ 119
        8.2.3 Maintaining consistency when adapting ................. 124
        8.2.4 Extra actions for task parallel applications ............ 129

9 Related work .................................................... 137
    9.1 Adaptive computing ......................................... 137
        9.1.1 Sequential computing .................................. 138
        9.1.2 Distributed computing ................................ 138
    9.2 Parallel computing .......................................... 139
    9.3 Adaptive parallel computing ................................ 139
        9.3.1 Parallel computing with load-balancing ................... 141
        9.3.2 Parallel computing with semi-dynamic processor set .... 142
        9.3.3 Parallel computing with dynamic processor set .......... 143
        9.3.4 Adaptive parallel computing on multiprocessors .......... 145
9.3.5 Comparative evaluation of adaptively parallel systems . . . . . . 145

10 Discussion and conclusions

10.1 Future work ........................................................................ 150
10.2 Trends and visions ............................................................... 153
Chapter 1

Introduction

Networks of workstations (NOWs) are becoming an evermore appealing option for parallel computing, as rapidly improving microprocessor performance and advances in high-speed networking are constantly narrowing the performance gap between workstation clusters and supercomputers for more and more classes of applications. The workstation approach offers more low-cost compute power than more tightly coupled designs. NOWs are often built solely from mass-produced and therefore cheaper commodity hardware, and the same is also true for the software. The free Linux operating system is another factor contributing to the cost advantage, as witnessed by its recent and on-going success.

Due to the above reasons, many organizations already have extensive NOWs in place or are increasingly switching to NOWs for their compute needs. At the same time, such NOWs are often not dedicated to a single user or compute problem. Many individual users may concurrently be using one or several machines each. Some users may do interactive and/or development work on their machines, while others are executing shorter- and longer-running programs of varying degrees of parallelism. In-between, many machines may be idle.

One major challenge to ubiquitous parallel computing on NOWs is the difficulty of parallel programming such that attractive speedups are achieved. Many programs were devised using a message passing paradigm. Message passing systems require the application programmer to handle all communication. The programmer may thus potentially optimize communication by only sending data exactly when needed to the correct processes, but achieving this may be a daunting task for more complex programs. The shared memory paradigm relieves the programmer from this burden, as the system automatically handles the data movement. A distributed shared memory (DSM) system provides such a shared address space abstraction for processes executing on machines that do not physically share memory. A DSM system consisting of a set of networked workstations, each with their own memory, offers the
abstraction of a globally shared memory. As such systems generally have less knowledge of an application's data usage patterns than the application programmer, a shared memory program tends to generate more messages than an optimized message passing version of the same program, at the expense of performance. Nevertheless, many improvements have significantly narrowed the performance advantage of message passing systems over distributed shared memory systems. We believe that the benefit of a much simpler programming model more than justifies using the shared memory paradigm instead of the message passing paradigm in many cases, outweighing the slight performance disadvantage of the former in distributed shared memory environments.

In this dissertation, we examine the feasibility of a system offering shared memory parallel computing with adaptive parallelism in a distributed memory environment such as a NOW. Adaptive parallelism lets a parallel program vary the number of processors it uses during its runtime, as detailed in the next section. The goal is for an on-going computation to employ extra machines and to withdraw from machines according to their availability or any given usage policy. We show how the shared memory paradigm lets a wide class of programs be run adaptively with little or no extra effort required by the application programmer.

As a basis for our work, we have used OpenMP, an emerging industry standard for shared memory programming, and TreadMarks, a state-of-the-art software distributed shared memory (DSM) system.

In the rest of this chapter, we first motivate our work in more detail, followed by some background information about the programming model and the system we used as foundations for our work. Next, we outline the key challenges that must be met by a successful solution. Finally, we state the claims that make up our thesis, and we show how our contributions validate these claims.

1.1 Adaptive parallelism

Adaptive parallelism refers to parallel computations on a dynamically changing pool of processors, where individual processors may join or withdraw from the computation as it proceeds. The main idea is to utilize otherwise-idle machines for parallel processing while retreating from such machines as soon as they are used (again) for other purposes, but adaptive parallelism has other attractive usage models as well. Networks of workstations are the most common setting for adaptive parallelism at present, as such environments often have many different users or owners of the individual machines, leading to frequent variations of the set of available machines.

The exact form of adaptivity depends on many issues such as the maximum num-
ber of processors $n$ in a given cluster, what degrees of parallelism an application supports, the response time of the system etc. We generally use a variation between 1 and $n$ processors.

1.2 Motivation

In this section we examine the reasons why we believe adaptive parallelism in distributed shared memory environments is a very attractive proposition worth pursuing determinedly:

- Attractive price to performance ratio of NOWs.
  Networks of workstations (NOWs) offer much more compute power for their cost than larger systems, and this cost advantage appears to be growing ever more in future. If these resources can be harnessed appropriately, NOWs can offer very cost-effective parallel processing. Many researchers have recognized this potential and strongly advocate the use of NOWs as parallel-computation platforms (e.g. [7]). Adaptive parallelism is a prerequisite for such parallel processing in non-dedicated environments and advantageous even in dedicated environments, as we see in Section 4.5.

- Lazy protocols make parallel computing attractive.
  Originally, the distributed nature of NOWs inhibited any parallel programs from achieving good speedups, except for programs with low communication-to-computation ratios. Over time, with the advent of faster networks and more relaxed memory consistency models, much improvement has been made. Protocols such as lazy release consistency (LRC) [46], coupled with technological advances, have significantly increased the communication-to-computation threshold up to which the performance of programs for DSM systems and hardware shared memory systems remains competitive [26].

  In addition, the LRC protocol produces favorable results when comparing the shared memory and the message passing paradigms: Many regular programs perform only a few percent worse with LRC than with message passing, while irregular programs using LRC may even outperform their message passing counterpart [27, 56].

  Attractive non-adaptive parallel computing is a vital foundation of our work: Adaptive parallelism lets the performance possible under ideal circumstances such as a dedicated NOW be achieved also in non-ideal circumstances where the availability of machines varies over time.
• Much idle time of machines in NOWs.

Many studies have observed that it is not uncommon for 60-80% of machines to be idle in typical NOW environments, even during peak daytime hours [2, 8, 60]. A system that permits these unused cycles to be exploited productively must be very attractive, as long as it does not have any serious disadvantages for any existing users. The idea is very appealing because the idle cycles are basically available for free, so no extra hardware or software costs are involved (except for the initial installation of the system).

• Shared memory paradigm vastly simplifies application use of adaptivity.

In the message passing paradigm, a programmer must decide when and with whom a processor must communicate and what data to send. This task is already challenging to handle in the non-adaptive case, but even more so when adapting.

In contrast, the shared memory programming paradigm lets each process access any shared data item, so the programmer is completely relieved from the handling of such data distribution and can focus on algorithmic development.

We make use of this automatic data distribution feature to automatically redistribute the data after an adaptation. Shared memory therefore lets the benefits of adaptive parallelism be enjoyed without requiring any significant effort by the application programmer.

1.3 Background

In this section we provide some more details about the programming model and the TreadMarks system we used as a basis for our work, including the memory consistency model supported by TreadMarks.

OpenMP. The OpenMP Application Program Interface (API) [61] describes a model for shared memory parallel programming in C/C++ and Fortran on all architectures, including Unix and Windows NT platforms. OpenMP is an emerging industry standard jointly defined by a group of major computer hardware and software vendors. The portable and scalable model defined by OpenMP gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer. OpenMP essentially provides a number of compiler directives that allow for a step-wise transformation of sequential programs to a parallel version, without the need for automatic parallelization tools. An important subset of OpenMP has recently been im-
implemented for distributed memory machines, in particular for NOWs, and for networks of SMPs [38, 57]. This thesis makes use of part of that work.

Lazy release consistency. First-generation DSM systems usually supported the intuitive sequential consistency (SC) memory consistency model [52], where the operations of each individual processor appear in the order specified by its program and any parallel execution corresponds to some interleaved execution on a single processor. Subsequently, more relaxed memory consistency models were developed. Release consistency (RC) [36] permits a process to delay making its changes to shared data visible to other processes until subsequent synchronization accesses occur. Several variations of RC exist with different definitions of when exactly one process’ modifications have to be propagated to other processes. The most relaxed version of RC is Lazy Release Consistency (LRC) [47], where consistency actions are postponed until a synchronization variable released in a subsequent operation is acquired by another processor, and even then, the modifications are only propagated to this acquirer. Programs written for SC still produce the same results when using LRC, provided that (1) all synchronisation is done using system-provided primitives, and (2) there is a chain of synchronization events between every pair of conflicting ordinary accesses to the same memory location by different processes. These requirements are not in any way restrictive, as any parallel program needs synchronization to avoid race conditions.

TreadMarks. TreadMarks [48] is an all-software DSM system for parallel computing on networks of workstations, providing a global shared address space across the different machines on a cluster. TreadMarks runs on standard Unix and Linux workstations, and a port to Windows NT has also been completed. The fact that it runs on top of existing operating systems and does not require any modifications thereof makes it very portable.

TreadMarks implements the LRC protocol and incorporates several additional innovative features such as an automatic changeover between single- and multiple-writer protocols on a per-page basis to effectively deal with false sharing. Further research has included the integration of compiler and runtime techniques, the use of multithreading, in particular on multiprocessor nodes, support for large address spaces, heterogeneity, and scalability. The LRC model and such refinements together enable many classes of parallel programs to attain attractive speedups with TreadMarks. Good performance has been demonstrated for a number of diverse applications when using up to 32 processors, both for TreadMarks’ homeless LRC protocol and for an alternative home-based LRC protocol [25].

TreadMarks supports programming in C, C++, Java, and Fortran, and more
recently support for the OpenMP API has been implemented as well.

We refer to a number of publications for further detailed descriptions of the implementation (among others [5, 46]).

1.4 Challenges for adaptive parallelism

Let us first examine the key requirements a system must comply with for adaptive parallelism to be attractive:

- Rapid withdrawal.

  The parallel computation should at any moment be capable of retreating rapidly from any of the processors used except the “master” processor. The processors should be freely reallocatable for other uses and (in multi-user environments) other users. Processors should be freed within a short time span such as a few seconds at most. The adaptive system can thereby avoid resource conflicts between individual users and the parallel computation, either adapting automatically as users come and go or yielding to their requests.

  Our model of adaptive parallelism suggests having one processor from which a parallel computation, once started, cannot be evicted, so that it may always proceed, even if “at minimal speed”. Stopping work completely and restarting it later on is not an adaptive parallelism issue but rather a checkpointing issue that we currently do not deal with.

- Modest cost of adaptivity.

  The overhead of adaptivity should only little increase applications’ total run-times, at most by a few percent for reasonable rates of adaptation. The overall speedups should not markedly decrease in runs with adaptivity compared to non-adaptive runs. The provision of adaptivity should further not involve a significant extra use of other resources, especially memory, such that in general an application running non-adaptively can also be run adaptively in the same environment.

- Simple, transparent programming model.

  Adaptations should ideally be transparent to the application, allowing a user to program in a standard way without requiring special-purpose code to support adaptivity. Supporting commonly-used programming models deployed on major platforms is key to a wide acceptance by application programmers. They should not be concerned with the question of whether programs are to execute adaptively or not.
• Ease of use.

Directing the system to perform adaptations should be very simple if not automatic for all users concerned. Starting a parallel computation should not be more complex in the adaptive system than in a non-adaptive system, and also the installation and configuration of the system and selecting the potential pool of machines to use should be straightforward.

• Reasonable implementation complexity.

Supporting adaptivity should not necessitate an overly complex implementation, for obvious reasons: The system should be maintainable and adaptable to new requirements, and for the sake of portability and easy installation, it should not require any operating system modifications.

From the list above, we observe that the main challenges toward the realization of an attractive system for adaptive parallelism can be summarized by two major issues: (1) How do applications handle adaptivity? and (2) How can the system support adaptivity?

1.5 Thesis

**Claim 1** With adaptive parallelism, shared memory applications can be run efficiently in a multi-user environment without compromising other users’ flexibility.

The parallel applications can be run while giving priority to other users who may arrive and leave in an unpredictable fashion. The parallel computation can rapidly be expanded to include any otherwise idle machines, and it can be withdrawn within seconds from machines that are to be used for other purposes. Selecting the machines to use for the parallel computation at any moment is simple and straightforward and can be done automatically.

**Claim 2** Many existing parallel applications using shared memory can benefit from adaptive parallelism either without needing any code modifications or with minor changes that a preprocessor can perform automatically.

Shared memory applications written for the OpenMP API can be converted automatically to TreadMarks code using a preprocessor. Data parallel applications require no code modifications specifically to obtain adaptivity, and task parallel applications require only small code changes to this end. Those changes can often be inserted automatically by a preprocessor.
Claim 3 The benefits of adaptive parallelism far outweigh the costs.

The total amount of work done in a multi-user environment with adaptive parallelism generally far exceeds the total amount of work possible in such an environment without adaptive parallelism. Without adaptive parallelism, the parallel computation and the individual users’ work needs to be scheduled so precisely that hardly any machines remain idle if the same productivity as attainable with adaptive parallelism is desired, as the extra overhead of providing adaptivity is very small. Such a schedule is impossible if the behavior of users is not predictable and the run times of the parallel computations are not known beforehand.

1.6 Contributions

The primary contributions of this dissertation are (a) the design and implementation of the capability to support adaptive parallelism in a software DSM system, (b) the practical performance evaluation thereof using a varied set of standard programs, and (c) the validation of the claims stated in the preceding section.

In Section 6.3 we show for a diverse set of application kernels and applications that the response time to incoming join- and leave-requests is low, so the system can typically adapt within seconds at most to the currently-available pool of otherwise unused resources. Section 4.4 shows how appropriate join- and leave-requests can be generated, such that other users can come and go at will and be assured of a free machine within seconds, provided the usage policy is such. Chapter 6 demonstrates that the overhead of adaptations is modest, such that adaptations affect the overall performance of a given shared memory application very little. These measurements and evaluations defend Claim 1.

Chapter 3 details the programming model supported by our system, showing how data parallel OpenMP applications can be run adaptively without any code modifications whatsoever, and how task parallel OpenMP applications can benefit from adaptivity with only one minor code change that a preprocessor can often perform automatically. These explanations validate Claim 2.

In Section 4.5 we present a selection of usage models for adaptive parallelism, showing many scenarios where parallel computing is only made possible thanks to adaptivity, or where adaptivity greatly enhances the possibilities and efficiency of parallel computing. Together with the performance results presented in Chapter 6, among others, which confirm that the various usage models add little extra cost, the value of an adaptively parallel system such as the one discussed in this thesis becomes clear, validating Claim 3.
Chapter 2

Solutions for adaptive parallelism

The goal of adaptive parallelism is to adapt an on-going computation to a variable pool of available compute resources. A variety of models are conceivable to this end. This chapter provides a conceptual overview of such models, including a closer look at the issues involved and the advantages and disadvantages of the respective models. We conclude with a look at the reasons leading to our choice of the model we have pursued and implemented.

2.1 Migrating the work

One approach for adaptive parallelism uses migration to move work around, distributing it as evenly as possible among the current set of available machines. The number of compute processes or threads remains constant at all times, so applications ideally remain totally unaware of any adaptations, save perhaps for the performance variations. An important advantage of this model is therefore the ability to run existing non-adaptive applications adaptively. All issues concerning the adaptations are handled entirely at a lower level by the system, applications do not require any special support for adaptivity.

Migration may however not be possible for all types of applications, depending on their use of operating system routines. A correct implementation of migration is difficult for operating system calls when state information managed by the kernel is involved (pending signals, process id etc.) or facilities depending on a process’ location are used (shared memory, sockets, pipes, semaphores, shared libraries etc.) or file I/O system calls are employed.

Another serious drawback of migration is the inherent impossibility of distributing fixed units of work evenly among a variable pool of resources. While some allocations will allow a good load balancing, others will not. Figure 2.1.a illustrates this issue: A parallel computation starting with an even load of one compute process per available...
node needs to move processes onto some other nodes as soon as some previously participating nodes wish to withdraw. The resulting multiplexing of processes causes a serious slowdown of these processes. In task parallel applications, the overall effect on the application’s performance is less dramatic, as each process works at its own pace, but in data parallel applications such multiplexing usually leads to many wasted cycles, as the application’s progress is determined by the slowest process. Conversely, additional resources cannot be used as they become available, they remain idle once each compute process is already running on a different node. Another scheme where one or a few machines are left idle initially to accommodate withdrawing nodes later on and avoid multiplexing only aggravates the problem of underemploying free resources, so there is a trade-off between multiplexing costs and good resource utilization.

An attempt to reduce the above performance loss is shown in Figure 2.1.b: Each node may by default run several compute processes or threads. This allows a more fine-grain work allocation and thus better load-balancing after an adaptation. However, the number of such processes or threads running on a node will generally still differ somewhat from node to node after an adaptation, i.e. the load is still somewhat unbalanced. In addition, more processes or threads per node may lead to worse performance overall, especially for larger numbers, due to extra context switches, etc.

2.2 Adjusting the work

The second model for adaptive parallelism suggests adjusting the number and allocation of processes as the available resources vary, such that the work always remains well-balanced among all available resources. As the number of processes decreases, each process increases its share of the work, and vice versa. This model is depicted in Figure 2.2.

One major issue to be solved in this model is the redistribution of work among processes when adapting. Non-trivial questions that must be dealt with are “How does the application continue the work where a leaving process withdraws?” and “How can a new process be given some of the work still outstanding, and how can it start execution at the correct point somewhere in the middle of an on-going computation?”, etc.

The second main issue to be tackled in this model is how to ensure memory consistency among a variable number of processes. A withdrawing process’ modifications to shared memory must not be lost, and a new process must see all the other processes’ writes to date, etc. Previous memory consistency protocols only run with a fixed number of processes specified when the computation begins.

A solution for the above issues should ideally let an adaptation be transparent,
2.2. ADJUSTING THE WORK

Figure 2.1: Adaptation with migration can lead to undesirable multiplexing of processes on the same node or to nodes remaining idle (above). Several processes or threads per node allow a better but still unsatisfactory load balancing after adaptation with migration (below).

allowing the user to program in a relatively standard way, without requiring any special-purpose code in the application, and the overhead of adaptivity should be small. These requirements are vital for this model of adaptive parallelism to be attractive versus migration, because migration also allows for transparent adaptations, but generally at a high cost.

In the next two subsections we consider two approaches for adaptive parallelism by adjusting the work. We examine the fundamental difficulties of each and suggest ways to overcome these. In the third subsection hereafter we set forth the reasons for the solution we have chosen and implemented for this thesis.

2.2.1 Adapting anytime

The ideal model of adaptive parallelism permits an addition and/or removal of compute nodes to or from the on-going computation at any moment. We can again conceive several different solutions for this, two of which we present in the next two subsections. In each case, we analyse the implications regarding the two main issues
that need to be addressed as formulated in the previous section, namely whether and how the application needs to handle adaptivity, and how the system handles adaptivity.

**Solution 1: Redistributing the work when adapting**

The first, most intuitive model calls for all work to be redistributed exactly when adapting, such that an adaptation is simply an interruption of work. After the adaptation, the computation proceeds from where it stopped for the adaptation.

**Programming model.** Allowing work to be redistributed at any given moment poses great challenges for the programming model. Consider a data parallel program with a process in the middle of a loop whose iterations have been divided among all the current processes. An adaptation with a redistribution of this work requires a change in the loop bounds of each process. This means that many iterations begun on one process are continued on another process. Often applications use process-local variables to store intermediate results before writing the result back to shared memory. In such cases, processes’ local states would have to be transferred to other processes, such that they could continue with results calculated by other processes. This difficulty may be overcome by changing applications such that they exclusively use shared memory for all their computations. However, the consequence would probably be a significant performance loss. In addition to such difficulties, another hurdle must be overcome: Apart from global synchronization points, processes do not execute in lock-step with each other. Therefore, an adaptation not only demands an overall work redistribution, but also a monitoring of each process’ progress at instruction-level granularity, so that work can be continued at exactly the right position by a new process after an adaptation.

All the above can probably only be achieved if the programming model is somehow
changed to lend support to such anytime-work-redistribution, which is undesirable for
the application programmer.

Adaptations in a task parallel program pose similar challenges as above. A task is
a unit of work defined in the application, and the work contained in one task does not
depend on the number of compute processes running. A task begun by any process
$x$ before an adaptation can therefore be completed by the same process $x$ after the
adaptation, as any change in the constellation of processes when adapting does not
affect the work comprising one task. Only withdrawing processes $x$ must obviously let
their unfinished tasks be continued by some other, non-withdrawing processes. For
such processes, the progress of the computation must somehow be recorded precisely
within a task and then be transferred to another process, where work is continued after
the adaptation. Joining processes finally simply begin with their first task.

**Memory consistency.** Memory consistency protocols supporting anytime adapta-
tions can be devised. The issues to be solved are the same for data and task parallel
applications.

Let us consider the implications of adaptivity for a memory consistency protocol
such as the one implemented in TreadMarks. The data redistribution associated
with an adaptation’s work reallocation is achieved via page faults, as processes access
pages of which they do not have valid copies, so the protocol need only guarantee the
correctness of any requested shared memory page. For this, the system must perform
the following actions at an adaptation: A *new process*, initially having only empty and
invalid pages, is sent all necessary consistency information matching a lock acquire
or barrier with a request containing an all-zero vector timestamp, indicating that no
intervals have been seen yet. It thereby receives all interval vector timestamps with
write notices, and for each such interval all page numbers of pages with write notices,
from all processes in the transitive synchronization history since the last garbage
collection. With this information, the new process can set all page states correctly.

A *leaving process* must generate any still outstanding diffs, and it must transfer any
lock tokens it is holding to some continuing processes, such as the respective current
or new lock manager processes. The *continuing and new processes* must then fetch
all the leaving processes’ diffs not seen yet by another process and all the pages
valid solely on one or more of the leaving processes and assume ownership of those
pages. The allocation of such diffs and pages to processes is for one part a matter of
performance. However, the difficult part is to let the continuing processes somehow
believe that the leaving processes’ writes had been performed by one of the continuing
processes, so that they direct the corresponding diff and page requests to one of
these processes in future, instead of to a non-existent withdrawn process. The lists
of interval records and write notices belonging to leaving processes must be dealt with. They cannot simply be “merged” with other existing intervals of continuing processes, because these intervals may have been seen by some processes that have not seen the intervals of the leaving processes, or vice versa, and adding extra “new” intervals to accommodate the leaving processes’ intervals leads to similar problems. All synchronization histories, i.e. which intervals a process has seen to date, must remain correct after an adaptation. In addition, also the memory consistency data of the non-leaving processes requires modification at an adaptation: In most cases process identifiers must be reassigned among the processes, to ensure a contiguous process-id space after an adaptation, and all currently stored memory consistency information must be fixed accordingly: All vector timestamps storing the interval indices for each process must be dealt with, i.e. the process vector timestamps and the vector time entries in all interval records, and further the per-process and per-page linked lists of write notices must be moved to the correct new page array entries. All such arrays of interval indices or pointers must be allocated with sufficient entries to accommodate the largest possible number of processes - when less processes are active, some entries remain unused. Finally, all other process-id dependent states must be modified appropriately, such as lock and barrier managers. This is especially tricky if some messages using such information are still in transit when an adaptation is to occur.

The above shows that an attempt to modify the memory consistency protocol such that correctness is maintained while permitting adaptations at any time is not trivial. We therefore consider alternatives that avoid some of the above problems. At an adaptation, we may let each process wait for any messages in transit, then the system may perform a barrier, such that all processes have an up-to-date memory consistency view. This alone does not solve the problem of modifying current memory consistency information yet, but when all processes interrupt their current work at a barrier, we may in addition easily include a garbage collection. This operation eliminates most of the above difficulties, because all shared memory pages are simply either updated or discarded, so most of the memory consistency information is discarded, in particular all write notices, interval records, diffs and twins, and all vector timestamps are reset to all zeros. Joining processes only need to receive page state information for each page (owner, writer, protocol) and set all page states to invalid and empty. After the garbage collection, the system resumes operation with the new number of processes nearly as if it had just been started, only the page state information reminds of the preceding computation, as far as the memory consistency protocol is concerned.

An adaptation accompanied by a barrier and a garbage collection as described is the preferable, simple way of managing memory consistency information when
adapting, as this approach does not have any perceivable disadvantages either, except possibly for the minor overhead of these operations. The system may always perform such an extra barrier, i.e. in addition to any barriers present in the application code, without affecting a program’s correctness. The barrier simply updates some processes’ memory consistency information earlier than the regular operation otherwise would. Consider a process $x$ now having a more recent copy of a page $p$ than without an adaptation. If thereafter $x$ accesses $p$, it should either have explicitly performed a synchronization operation such that it used the same version of $p$ with or without a preceding barrier, or there is false sharing of $p$, so in either case the result would be the same. Any other result would be the consequence of a race condition, i.e. a programming error.

Although maintaining memory consistency can be achieved elegantly with the barrier-plus-garbage-collection approach, the challenges for the programming model mentioned in the foregoing section remain.

Let us consider the implications of adaptivity for some other memory consistency protocols, Automatic Update Release Consistency (AURC) [40] and Home-based LRC (HLRC) [84]. The principal difference between TreadMarks’ LRC protocol and these two protocols is the use of a home for each page, such that all updates are propagated to and all copies are derived from the respective home node. AURC uses a hardware-assisted update mechanism to automatically propagate all writes to the home node’s copy of a shared memory page, thereby eliminating the need for diffs. HLRC uses diffs to capture all updates to a shared memory page within an interval. The diffs are only sent to the respective home nodes, where they are applied immediately and then discarded, and the writers also discards diffs as soon as they are sent. AURC and HLRC both handle page faults by sending the whole page.

Without going into details, the fact that updates are centralized on one page instance significantly simplifies the data structures needed for the memory consistency protocol, as compared to the TreadMarks implementation, where updates are applied locally as a page’s copy is accessed, but write notices are still needed to let a process determine whether it has a valid copy of a page. Again, the combination of an extra barrier and garbage collection at an adaptation poses a simple solution for adaptivity in these protocols, as the write notices can be discarded and all interval indices can be reset to zero. Pages whose home node is a leaving process need to get a new home node, and all other processes must be informed of this move.

**Solution 2: Restarting work from well-defined points**

The second model only redistributes the work at certain well-defined points $p$. When an adaptation takes place, all work a process has done after passing the last such
point is lost, i.e. the adaptation is performed as if the computation was at that point, then work is restarted from there on. This model greatly simplifies the issues outlined for solution 1.

**Programming model.** In this model the work is described in units executable between successive points $p$ by one process, and only at such points $p$ may the work be rearranged among the current processes. As long as the computation’s progress can be clearly described at any point $p$, the process-private local states are not needed and can be discarded at the actual point of adaptation $a$. The only requirement is obviously that such a roll-back of the work from $a$ to the previous $p$ can either be done simply by discarding the process-private local states, or some other efficient way is found.

The application or the system must guarantee that performing the work between $p$ and $a$ a second time after the adaptation does not alter the computation. If the computation’s progress at $p$ is stored in shared memory and all writes between $p$ and a subsequent adaptation point $a$ are performed only to local private memory, then the roll-back can be achieved simply by ignoring the local memory. However, writes to shared memory would have to be “undone” in many cases. If no synchronization operations take place between $p$ and $a$, then the modifications are at least confined to the local process’ copies of shared memory pages, as no other processes would have seen the writes.

In data parallel applications, $p$ may correspond to the boundary between individual iterations of some outer loop, and the iterations of some inner loops may be divided among the current number of processes. In task parallel applications, $p$ may equal task boundaries. As in solution 1, after an adaptation the continuing processes proceed with any partially-completed task and the new processes begin work on their first task. The results of a leaving process’ unfinished task must however be discarded, then the task is started anew by some other process. For this, the system must have some mechanism for keeping track of begun but unfinished tasks, so that these may be restarted as needed.

**Memory consistency.** The same issues apply as for solution 1. In fact, a roll-back of work to a point $p$ by each process implies that all processes must interrupt their work, so the execution of an extra barrier does not cause much extra overhead. The most straightforward solution is again to perform a garbage collection and to discard all memory consistency information except page state information. In addition, as stated above, any shared memory state appertaining to rolled-back work must be undone/discarded.
2.2.2 Adapting at global synchronization points

This model performs adaptations only at global barrier-type synchronization points $p$. These must either be contained in the application code, in which case the frequency of adaptation opportunities is defined by the application, or the system must additionally be able to execute such synchronizations.

Programming model. For this model, work only needs to be defined in units executable between successive points $p$, and only there the work must be reallocatable to a new number of processes. This aspect is similar to the previous subsection’s Solution 2, but unlike there, no roll-back of any extra work has to be done. Rather, whenever an adaptation request arrives, the system continues on to the next adaptation point $p$ and adapts there or at any later $p$.

Memory consistency. As the system is already performing a barrier synchronization, following this with a garbage collection suggests itself, allowing the same simple solution as outlined in the previous subsection.

2.3 Conclusions and proposal

Our main goal is to demonstrate the benefits of adaptive parallelism in DSM environments. Others have developed solutions using migration, albeit for somewhat different environments. Also, any solution based on migration alone must suffer a serious performance penalty as sketched in Section 2.1, and the implementation issues are non-trivial for any system useful for a wide range of applications.

In contrast, to our knowledge no system to date provides adaptivity for shared memory programs in DSM environments by adjusting the work unless some extra specialized programming paradigms are employed.

We propose and have implemented a system consistent with the model presented in Section 2.2.2. As Section 2.2 shows, there is evidently a trade-off between functionality on one side and simplicity of the programming model and the implementation on the other side. The less restrictive the model is as to when adaptations are permitted, the more complex the programming model and the implementation become. We believe that the restrictions imposed by our choice of model are still only non-essential, such that we can demonstrate the viability of adaptive parallelism in DSM environments. The following chapters of this thesis show that our solution is sufficiently flexible to establish this thesis’ claims put forth in Section 1.5. Our strategy is to (at first) pursue the simplest solution (implementation-wise) that is adequate to prove these claims, such that results can be obtained within a reasonable time frame. Actually,
we presume that different, more flexible but more complicated solutions might have required more complex programming models incompatible with Claim 2 of our thesis.

Our proposal avoids the aforementioned performance and implementation problems associated with migration. Our proposal further avoids the difficulties of devising a (proprietary) programming model and system with which work can either be reallocated at will or be rolled back and performed again by some other process, and finally, it permits a straightforward implementation.

The system we propose supports the OpenMP programming model discussed in the next chapter. The use of an industry standard such as OpenMP represents a novel approach, it is the first system we are aware of that demonstrates that a simple shared memory programming model allows for adaptivity with little performance overhead in a DSM environment.

Using OpenMP permits us to adroitly adopt and implement the chosen programming model by exploiting OpenMP’s fork-join model of execution: During all sequential sections, only the master thread is busy and the slave threads do not exist (from the application’s point of view). Performing adaptations exclusively at such moments therefore means slave processes have per definition no application state needing to be saved and moved to another process, so all work can be reallocated from the master process.
Chapter 3

Programming model

This chapter treats the issue of how applications can make use of adaptivity. We first briefly introduce OpenMP, the parallel programming model we support, then we outline how OpenMP can be used in a NOW environment, and finally we explain how standard OpenMP applications can run adaptively without any code modification or with one minor change in some cases.

3.1 OpenMP

OpenMP [61] is an emerging industry standard for shared memory programming. The OpenMP API provides a number of compiler directives that allow a user to indicate the parts of the program that are to be executed in parallel. These directives allow a gradual migration from a sequential to a parallel program, independent of the availability of tools for automatic parallelization. In addition, OpenMP frees the programmer from having to deal with some aspects of parallel programming such as the number of nodes, the low-level details of iteration or data partitioning, or the communication of data between nodes.

These properties not only simplify parallel programming but also lay the foundation for transparent adaptive parallelism. As the number of nodes is handled by the system and not the application, it becomes possible to change the number of nodes without user intervention.

In this thesis, we focus solely on the parallel loop construct available in OpenMP, shown in Figure 3.1, as other directives are not relevant for the adaptivity we are concerned with. OpenMP uses the fork-join model of parallel execution. Initially and in-between any parallel loops, an OpenMP program executes as a single thread, the master thread. At each parallel loop directive, the master forks a team of \(t\) threads (including the master thread), and work is continued in parallel among these threads. Upon exiting the parallel construct, these threads synchronize (join the master), and
-- this code is executed sequentially and only by the master --

#pragma omp parallel for
for (i=0; i<MAX; i++) {
    -- the iterations of this loop are divided among all processes --
}

-- this code is executed sequentially and only by the master --

Figure 3.1: Pseudo-code for OpenMP C parallel for construct.

only the master continues execution. A program may fork and join in this way any number of times.

The fork-join execution model suggests natural adaptation points in the sequential sections in-between parallel constructs: As the number of threads is not hardwired in the source program, the system has the opportunity to start each new parallel loop with a new number of threads. The degree of parallelism need only be constant during the execution of one parallel construct. The system automatically deals out the loop’s work among the chosen number of threads without involving the application, such that any change in the number or constellation of threads is completely transparent to the application.

3.1.1 Data parallel OpenMP applications

Data parallel OpenMP applications often have a rapid succession of many short sequential and parallel sections, such that natural adaptation points are reached frequently during execution (often one or more per second). Iterative algorithms where all processes synchronize globally - i.e. an adaptation point - and exchange data once or even several times per iteration are common.

An OpenMP join-fork sequence that constitutes an adaptation point where only the master thread is active is also equivalent to a barrier synchronization used in other parallel programming models.

3.1.2 Task parallel OpenMP applications

The task queue model is a different work sharing construct: It has one (or more) shared queue(s) where threads repeatedly enqueue new tasks and dequeue and solve tasks, until all tasks are computed and no thread is producing any new tasks. Every thread performs these operations at its own pace. There is no all-to-all global synchronization
that can stall the faster threads, so the load is well-balanced. In OpenMP, such a
scheme translates into one main parallel construct where the threads work on the task
queue.

The absence of natural adaptation points in this model due to the absence of
frequent OpenMP join-fork sequences makes the provision of adaptivity somewhat less
straightforward than in data parallel applications, as we see below in Section 3.3.2.

3.2 OpenMP on a NOW

OpenMP is designed for a shared memory environment. To run OpenMP programs on
a NOW, we compile OpenMP to the TreadMarks distributed shared memory (DSM)
system [6]. TreadMarks is a user-level software DSM system that runs on commonly
available Unix and Linux systems and on Windows NT (cf. Section 1.3). TreadMarks
provides multi-threaded parallel programming primitives similar to those used in hard¬
ware shared memory machines, namely process creation, shared memory allocation,
and lock and barrier synchronization.

Throughout this thesis, we equate the OpenMP document’s use of the term thread
with the term process. In our distributed implementation of OpenMP, these threads
execute as Unix or Linux processes, so we use these two terms synonymously.

To support OpenMP-like environments, the TreadMarks API includes Tmk_wait,
Tmk_fork, as well as Tmk_join; these primitives are specifically tailored to the fork–
join, master–slave style of parallelism expected by OpenMP and most other shared
memory compilers [4]. Tmk_fork is a one-to-all synchronization used by the master
to let all processes start executing the parallel section, and Tmk_join is the converse
all-to-one synchronization called by the master at the end of the parallel section. The
slave processes use Tmk_wait to wait for the next Tmk_fork issued by the master. After
completion of the parallel section, the slaves return to the wait-state within Tmk_wait,
waiting for the next Tmk_fork performed by the master. To improve performance, the
slave threads are thus simply blocked during the sequential sections, instead of being
destroyed and recreated at every new parallel section.

Compiling an OpenMP C program to TreadMarks is fully automated. The compi¬
er is based on the SUIF [4] pre-processor. The body of each parallel loop (or,
more generally, each parallel construct) is encapsulated into a new procedure. In the
master, the loop is replaced by a call to Tmk_fork with as argument a reference to
the procedure embodying the parallel loop. Additional code generated inside this
procedure lets each process figure out, based on its TreadMarks process
identifier and the total number of processes, which iterations of the loop it should execute. The
procedure terminates with a call to Tmk_join, and the slaves return to the waiting
Table 3.1: Loop condition code modifications needed to support adaptivity in task parallel OpenMP applications. These transformations are done automatically by a preprocessor.

More details of the OpenMP to TreadMarks translation can be found in the paper [57]. The pre-processor supports a sizable subset of OpenMP, such that many OpenMP programs can be fully transformed to TreadMarks code.

### 3.3 Transparent adaptation

#### 3.3.1 Data parallel OpenMP applications

Data parallel OpenMP programs require no code modifications whatsoever to support adaptivity [70]. The use of OpenMP join-fork sequential sections as implicit adaptation points lets the system handle all aspects of adaptivity, such that adaptations are completely transparent to the application, as shown in the previous sections. The preprocessor translates the OpenMP code to TreadMarks code in a straightforward way, as described above.

#### 3.3.2 Task parallel OpenMP applications

As task parallel applications typically have no or few OpenMP join-fork sequential sections that serve as adaptation points (cf. Section 3.1.2), we resort to a new solution outlined in this section [69].

In our task queue model, slave processes perform all work either inside tasks or, as in data parallel applications, within other OpenMP parallel sections containing no tasks.
3.3. TRANSPARENT ADAPTATION

Code executed by all threads:

```c
void _Worker_func(struct Tmk_sched_arg *my_arguments)
{
    ...
    do {
        if (PopWork(&task) == -1) {
            break;
        }
        Quicksort(task.left, task.right);
    } while (!Tmk_leave()); /* before modification: while (1); */
}
```

Code executed by master:

```c
Tmk_fork(_Worker_func, &Tmk_arguments);
...
```

Figure 3.2: Example structure of a task queue application (Quicksort) showing modification for adaptations according to rules in Table 3.1.

To allow for a transparent adaptation whenever an adapt event occurs while the application is busy with a task queue, we let each process finish its current task, then we let the system execute an extra OpenMP join-fork sequence. Having all slave processes’ work for the current OpenMP parallel section being contained in the tasks ensures that slave processes do not have any compute-relevant private process state when the adaptation is performed. We introduce a new TreadMarks primitive `Tmk_leave`, to be called by the application to indicate completion of a task. Inside `Tmk_leave`, the system has the opportunity to execute an extra OpenMP join-fork sequence, without the application having to call the join-fork routines. The system can thus create an adaptation point, where it can easily adapt. When an adaptation is due, the system only needs to wait for all processes to arrive at the next OpenMP join, i.e. the next `Tmk_leave` (unless the task queue is empty and `Tmk_leave` is not called again). When no adaptation is pending, `Tmk_leave` returns immediately, i.e. no OpenMP join-fork sequence is executed, so processes need not wait for each other there and practically no overhead is added.

A `Tmk_leave` call returns TRUE if a process is to leave, FALSE otherwise. The preprocessor inserts this call at task boundaries. More precisely, the preprocessor modifies the termination-condition of top-level loops of the functions called by OpenMP forks according to the rules in Table 3.1. If a forked function does not have any other (compute-relevant) top-level statements besides a loop which retrieves and adds tasks, as in the applications investigated, then the preprocessor can perform the correct code modifications automatically (Figure 3.2). A TRUE return value must cause the process to end without performing further computation.
Adaptations are completely *transparent* to the application, as the only application code modification is the insertion of `Tmk.leave`. There, leaving processes must return from the forked function and terminate, while continuing processes experience a slight delay while the system performs the adaptation, and joining processes begin execution of the forked function.

In the current version of the system, task parallel applications can be run adaptively by using a `-T` switch when starting the program. Omission of this switch lets the system assume that a data parallel application is running. Alternatively, the preprocessor that performs the above code transformation could be modified to set a system variable identifying the application type in the application code. A third possibility would be to let the system initially assume that the type is data parallel and set the type to task parallel upon the first call of the `Tmk.leave` routine.

### Eliminating all code modifications

In our current model, task queues are maintained by the application, as the OpenMP standard does not explicitly support task queues. However, Shah et. al. have proposed their WorkQueue model [71] as an addition to the OpenMP standard, offering two new pragmas, `taskq` and `task`, for task queues and tasks, respectively. Following the acceptance of the proposal, we may modify our system accordingly, *eliminating* the need for the `Tmk.leave` primitive, as the system will recognize task boundaries through use of the `task` pragma.

The WorkQueue model allows nested task queues. In our model, we permit adaptations only in top-level task queues, other task queues are completed non-adaptively, avoiding the complexity of dealing with compute-relevant slave process states, such as letting another process complete some half-finished task of a leaving process.
Chapter 4

A DSM system supporting adaptive parallelism

This chapter presents the functionality for transparent adaptation, and it sketches various user interfaces for the system and usage models for which adaptivity is of great importance and benefit.

4.1 Functionality

4.1.1 Join and leave events

The system may add new OpenMP threads of execution to and/or withdraw and terminate such threads from the current computation. These actions are called *join events* or *leave events* or collectively *adapt events*. Requests for adapt events may occur anytime, but the system only performs the desired actions at *adaptation points* (Section 3.1), as follows: The system receives the events via a network connection to

![Diagram of Join and Leave Events]

*Figure 4.1: Join events (a) and leave events (b).*
an external control tool (see Section 4.4). A leave event is then performed at the next adaptation point, whereas a join event first lets the system spawn the new process, and once all network connections are completed, the adaptation is performed at the next adaptation point thereafter. The exceptions are multiple join requests, which may be batched together in one adaptation, such that the first joins are delayed until all new processes are connected (see next subsection). The set-up of network connections for joining processes typically requires about 0.5-1 seconds, but this delay is of little relevance, as the on-going computation of the other threads is not affected save for perhaps a millisecond while responding to the new thread’s connection requests. An adaptation is the aggregate of all adapt events performed at one adaptation point (Figure 4.1).

More elaborate policies, such as scheduling adaptations in response to certain external conditions, such as other programs being active etc. can be implemented by some external control instance as sketched in Sections 4.4 and 4.5. As such, these issues are not dealt with inside our Adaptive TreadMarks system and are not part of our work. The system simply reacts to adapt events as they are received, irrespective of their origin.

Every OpenMP join to OpenMP fork sequential section of an OpenMP program constitutes a natural adaptation point where the number of compute threads can easily be changed, as we have seen in Section 3.1. Data parallel applications often have many such opportunities for adaptation - usually several per second in the applications we have tested - while in task parallel applications we insert an extra OpenMP join-fork sequence to this end using the Tmk.leave routine, as detailed in Section 3.3.2. As this mechanism requires every process to be at a task boundary, and as tasks may be of variable length and processes execute tasks at their individual pace, some processes may be idle for some time before an adaptation, while other processes are still finishing their current task (Figure 4.2). Note that this delay only occurs in the event of an
adaptation. `Tmk.leave` checks whether an adaptation is due. In the absence thereof, it returns immediately and therefore causes practically no overhead. The extra idle time only occurs before adaptations in task parallel applications, but not in data parallel applications, as the latter only adapt at OpenMP join-fork sequences already present in the application code.

The system does not decide on which machines it runs the OpenMP compute threads. In our distributed implementation of OpenMP, an OpenMP compute thread executes as a Unix or Linux process. The user specifies which machines to use when starting a computation and when performing a join event, and the system starts the processes on those machines. If desired, one may devise some automatic thread-to-machine allocation system - commonly, one process per CPU is run.

### 4.1.2 Multiple adapt events

The system can execute any valid combination of adapt events at one adaptation point. In fact, handling many adapt events at once is substantially cheaper than performing the corresponding adaptations sequentially.

We make use of the above observation by implementing the following policy for the handling of adapt requests: Pending adapt events are batched together as much as possible while always giving priority to any leave events. When a leave is requested, that machine is presumably needed for some other task with a higher priority, so we wish to clear the process off the machine rapidly. A join request on the other hand has the nice property that a delay does not have any adverse effects on the current computation or on any other users, the only consequence is a machine staying idle for a little longer.

Putting the above policy into practice results in a behavior as follows: Consider several leave requests arriving shortly after each other without the computation passing any adaptation point meanwhile. In this case, the requests are batched, all are executed at the next adaptation point. Join requests are somewhat more complex: Recall that new processes first have to set up all network connections, only then are they ready to join the computation, and furthermore the system sets up network connections to new processes one-at-a-time. Now, as long as further join requests arrive before the system has reached the first adaptation point following the successful connection of this pending join, i.e. before this process has had the opportunity to join the computation, the system proceeds to start and connect those additional new processes one-by-one until all are connected, and only then all the new processes join the computation in one go at the next adaptation point. Postponing some join requests and batching them with later join requests reduces the adaptation costs, as the performance analysis shows (Section 7.5.4). If however any leave requests arrive
in the meantime, they are always immediately executed at the next adaptation point, along with the joining of any fully-connected new processes at that point. New processes that are ready to start work may be included without delay, but a join of a new process which at this point is still busy setting up network connections is aborted and terminated, then the join-attempt is restarted immediately after the adaptation.

Any join request received while another new process is not yet fully-connected is deferred such that new processes are spawned and connected one-by-one as described above.

4.2 Enhancements

In this section we briefly describe two possible extensions which are orthogonal to the functionality of the base system described in the previous section. The current version of the system used throughout this thesis does not include these two extensions, but their viability has been demonstrated practically in an earlier implementation of the system which only supported data parallel adaptive parallelism. We do not consider these extensions vital for our thesis, but if desired, they may be added to the system again with reasonable effort.

4.2.1 Urgent leave events

On occasion we may wish to withdraw the computation from a certain machine as fast as possible or within a certain short interval. For such cases, we introduce the concept of a grace period $t_{\text{grace}}$: When a leave request arrives and the computation can reach the next adaptation point within the time limit $t_{\text{grace}}$ specified by the user, we let the leave event take effect there (Figure 4.3.a). This is the normal case we have seen in Section 4.1. If the computation does not encounter an adaptation point within the grace period, then the system migrates the current process to another machine.
already involved in the current computation. The process previously running there and the migrated process then have to be multiplexed until the next adaptation point is reached, at which time the system performs a leave of one of the two processes. We call this sequence of events an urgent leave, it is depicted in Figure 4.3.b. Urgent leaves cause much more data to be moved than normal leaves, because all intermediate data of the migrating process needs to be transferred. In addition, if a computation is balanced for \( t \) processes, multiplexing one node may idle the \( t - 2 \) non-multiplexed nodes for some time (alternatively, a policy which leaves one or more machines idle during regular use, keeping them for such migrated processes, may avoid the problem of multiplexing, but having available resources stand idle for most of the time is probably not desirable).

Given the application code, the number of adaptation points is usually known precisely for data parallel applications and more or less for task parallel applications. If the average distance between successive adaptation points does not vary enormously during computation, then reasonable values of \( t_{\text{grace}} \) can be estimated using some previous performance results of the same applications. Migration can be avoided by choosing values of \( t_{\text{grace}} \) that are somewhat larger than the longest expected interval between successive adaptation points.

As (1) potential adaptation points are reached fairly frequently - in our applications typically several times a second and even in the worst case after a few seconds - , and (2) the owner of a workstation is not denied service while a compute process is still active, urgent leaves are often not needed in practice.

Migration works as follows: First, the new process is spawned on the new machine and all the low-level network connections are set up, exactly as in the case of a joining process. The migration is then performed, while all processes wait for its completion. We rely on a modified version of the libckpt library to implement migration [62]. Libckpt was designed for checkpointing to disk and for recovery from that checkpoint, but the modifications let it write out the heap and the stack of the leaving process to the newly created process, then that process is started.

### 4.2.2 Fault tolerance

Checkpointing may be employed to recover from catastrophic failures such as a crash, power flicker, or a machine reboot. In addition, such a mechanism also permits a planned complete interruption of work: The system may withdraw from all machines and resume work later on, in contrast to our current model, which stipulates that a computation, once started, is always running with a minimum of one process.

A distributed computation normally requires a consistent checkpoint [22] or some form of message logging [43] to guarantee correct recovery. However, we can avoid
much of this complication by limiting checkpoints to the OpenMP adaptation points. At these points in the execution, the slave processes do not have any private process state (such as a stack) that needs to be recovered; they only have shared memory state. Only the master process has process state that needs to be recovered.

We can perform periodic checkpointing as follows: First, a garbage collection is invoked to bring shared memory into a well-defined state. Second, the master collects all pages for which it does not have a valid copy. Finally, the master uses the libckpt library to checkpoint itself to disk. No checkpointing by slave processes is required, avoiding the considerable complexity of checkpoint and recovery coordination.

4.3 Limitations

The master process cannot withdraw from the computation in the current implementation of the system. To overcome this limitation, some slave process would have to be capable of assuming the role of master process. This is not trivial because it is not only a system-internal issue, as the distinction between master and slaves is implicitly contained in the OpenMP application code, as the master executes additional code with respect to the slave processes.

Adaptivity is limited to the adaptation points. Programs with no OpenMP join-fork sequences, or task parallel programs that do not permit the extra insertion of such sequences, can only run non-adaptively.

It is also quite possible in OpenMP for the user to explicitly code the iteration partitioning in terms of the process identifiers and the number of processes. Clearly, such applications cannot benefit from adaptivity, where the system must be permitted to change process identifiers and reset the number of processes. It is furthermore possible for the user to explicitly disable adaptivity by setting the switch that OpenMP provides for this purpose.

4.4 User interface

Applications are run as in standard TreadMarks, except that the system provides a new command-line option -T to specify task parallel applications. Omission of this switch lets the system assume a data parallel application is running.

As in TreadMarks, the system reads the .Tmkrc file, specifying the machines the system may use. The file may contain multiple identical entries, in which case the system may run one Unix resp. Linux process for each such entry on the same machine.

Adaptations may be requested by any external control instance that can send the appropriate signals to the system: The master process provides a TCP/IP con-
4.4. USER INTERFACE

Figure 4.4: An example user interface for generating join and leave events in an 8-machine cluster.

nection where adaptation requests are received asynchronously, generating an I/O signal. The event handler reads two bytes, (a) the type of event (join or leave), and (b) a number specifying which entry in the .Tmkrc file is concerned by the event. At every adaptation point, the system also uses this connection to export the pool of machines currently running a process and the current process identifiers of each compute process. The difference between the former and the latter shows when a process already exists but is not a compute process yet, i.e. a new process about to join the computation.

Figure 4.4 shows a simple user interface we used for some of our experiments: After starting the application on a subset of machines in a cluster, a user can let the computation expand to and withdraw from machines using the Join and Leave buttons, respectively, and the fields with the machine names change color to indicate which machines the computation is presently using. The tool lists the machines specified in the .Tmkrc file and acts as a server waiting for Adaptive TreadMarks executions.

A more ingenious set of tools with various capabilities can be devised:\textsuperscript{1}: From all the machines making up a cluster of workstations, a subset of machines $m$ potentially available for parallel computing can first be chosen using a master tool. The simplest mode of operation now lets a user start the application on a chosen subset $n$ of the $m$ nodes, then while the application is running one can change the pool of active machines by issuing join and leave requests. Other users may also generate such requests by using slave tools running on each of the machines. These tools send their signals to the master tool that connects to the adaptive DSM system. Other modes of operation permit schedules of availability for each of the machines to be preset via the master and/or the slave tools, and application runs may also be preprogrammed.

\textsuperscript{1}Not implemented yet.
The master tool can then run the jobs on the machines specified to be available at the given times. When more than one job is outstanding, it schedules the job mix, possibly running several jobs in parallel and adapting on-going computations when another computation begins or ends. A user may specify various parameters, such as urgency of a job and approximate speedup for different numbers of processes, which the tool takes into account when scheduling the job mix. Finally, the tool can monitor the load and activity on each machine and thereby decide which machine is available or not for parallel computing and request the corresponding adaptations.

Finally, the system also adapts according to instructions given in adaptation files, where one can specify exactly which machines to add and which machines to withdraw at which adaptation points. This capability is used primarily for test purposes.

4.5 Usage models

The capacity to adapt the number of compute processes during runtime leads to a number of interesting new usage models. The list we present is not exhaustive, and some of the usage models below may be combined in various ways.

**Multi-user environment, one parallel application.** In a multi-user environment, a parallel application can use any otherwise unused machines by adapting as users come and go, and at the same time all these users retain priority. Only one machine running the application’s master process need be reserved for the parallel computation. For this scheme, adaptivity is a necessity, because a non-adaptive system cannot react to or predict the “random” arrival of other users and give them priority.

**Multi-user environment, reservation of unspecified machines.** A given percentage of machines may be reserved for parallel computing, but the exact machines used need not be specified and can vary throughout the computation, only one machine for the master process need be reserved, as above. Other users may freely choose their machines as long as enough other machines are available for parallel computing.

**Multi-user environment, variable parallel jobs mix.** Two or more parallel applications may be run simultaneously. The job mix may be altered at runtime to accommodate new application runs, as described in Section 4.4 above. This is especially valuable in situations where for example a long-running parallel program is active and another more urgent program needs to be run at short notice. Now the
system can scale down the number of nodes allocated to the first program and start the other one immediately.

**Dedicated environment, variable parallel jobs mix.** Adaptivity lets the resources of a cluster of machines reserved for parallel computing (1) be better exploited and (2) be more available for urgent or changing requirements. Taking into account the speedups, the most efficient allocation of resources to compute jobs may be found, such that several jobs may execute simultaneously, and applications expand onto or withdraw from other nodes as other applications finish or are started there. As the job mix changes over time, adaptivity permits to optimally take into account the varying ideal speedups of different applications. Further, new and perhaps urgent jobs may be started at short notice, as described in the previous paragraph.
Chapter 5

Methodology

This chapter presents the methodology we used to evaluate our adaptive DSM system. We describe the programs used for our tests, how we measured the adaptation costs, and which test scenarios we ran to get representative results.

We used nine standard application kernels and applications to assess the performance of our system: Five of these are data-parallel programs with regular access patterns: 3D-FFT, Jacobi, Modified Gramm-Schmidt (MGS), Successive Over-Relaxation (SOR) and Water, a molecular dynamics simulation. Two programs are data parallel, but with irregular access patterns: Barnes-Hut (Barnes), an n-body simulation, and Non-Bonded Force (NBF), the kernel of another molecular dynamics program. Finally, we used two task parallel programs: Quicksort and the Traveling Salesman Problem (TSP).

5.1 Experimental environment

Our primary testbed consists of 16 dual-processor 400MHz Pentium II machines running Linux 2.2.7 and with 256MB of memory each. For comparison purposes, we have two separate fully-switched full-duplex Ethernet networks with bandwidths of 100Mbps and 1Gbps, respectively, connecting the machines.

We have not exerted any special effort to optimize the setup, as the primary goal of this thesis is the demonstration of the validity of our ideas for NOW environments in general and not for highly-optimized environments. Obviously, better setups would make our system more attractive.

We have developed our system using TreadMarks version 1.1.0 as a base. We use the original TreadMarks mechanisms for communication, i.e. UDP sockets. The costs of some basic operations for this version of TreadMarks and for our system are the same, they are shown in Table 5.1 for each of the two networks. Note that the latency is very similar in both networks, the advantage of the 1Gbps network over the 100Mbps
network is the superior bandwidth. To better exploit the available bandwidth, we set the page size to 16K when using the 1Gbps network, compared to the standard 4K for the 100Mbps network. We compiled the system for a maximum of 256MB of shared memory and 32 threads, and we used a timeout for retransmissions of 100ms throughout our experiments. When the system moves pages off leaving processes, we let it aggregate a maximum of 8 4K pages or 3 16K pages in one message. All these parameters are constants whose values may be easily changed for other environments.

5.2 Classification of applications

We classify applications by their access patterns as regular, irregular or independent (Figure 5.1). Regular applications have a block or block-cyclic data distribution, a thread’s data partition is determined by the thread’s identifier \( id \) and the total number of threads \( nprocs \). In the block data layout, \( n \) processes are assigned a contiguous chunk of about \( \frac{1}{n} \) of all data each. The block-cyclic data layout is essentially a succession of block layouts put together, so the block size is much smaller, with the difference that this

<table>
<thead>
<tr>
<th>Network →</th>
<th>100Mbps</th>
<th>1Gbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page size →</td>
<td>4k</td>
<td>4k</td>
</tr>
<tr>
<td>1-byte-message roundtrip</td>
<td>219</td>
<td>217</td>
</tr>
<tr>
<td>Lock acquire minimum</td>
<td>232</td>
<td>237</td>
</tr>
<tr>
<td>Lock acquire maximum</td>
<td>369</td>
<td>383</td>
</tr>
<tr>
<td>8-node-barrier</td>
<td>588</td>
<td>429</td>
</tr>
<tr>
<td>Diff fetch minimum</td>
<td>279</td>
<td>300</td>
</tr>
<tr>
<td>Diff fetch maximum</td>
<td>940</td>
<td>620</td>
</tr>
<tr>
<td>Page fetch</td>
<td>797</td>
<td>448</td>
</tr>
</tbody>
</table>

Table 5.1: Times in microseconds for some basic operations in the two test environments.

Figure 5.1: Examples for various data distribution types and four threads.
5.3 APPLICATIONS

size is not defined by the number of processes \( n \), but by the problem set size. Threads primarily access the data in their partition, and access of other data is limited in space, e.g. to the data at the edge of the blocks, or limited in time, as such data is accessed less frequently than the own partition.

In irregular applications, a thread’s data partition is still determined by the thread’s identifier \( id \) and the total number of threads \( nprocs \), but the allocation is irregular and may change frequently during the course of computation.

Independent applications finally have access patterns that are independent of process identifiers \( id \) and dependent on \( nprocs \) only inasmuch as this value determines the share of work performed by each process. Task parallel applications are independent, as one (or more) queue(s) of tasks is maintained, from which every thread fetches a new task whenever it is done with one task. With such a setup, one cannot predict by which thread a given task will be computed. As the data is associated with the tasks, the data distribution is therefore equally unpredictable.

5.3 Applications

Barnes

Barnes-Hut (Barnes) is an application from the SPLASH benchmark suite [74] and simulates a system of bodies influenced by gravitational forces. A body is modeled as a point mass that exerts forces on all other bodies. The algorithm uses a hierarchical oct-tree representation of space in three dimensions. The space is broken into cells. The internal nodes of the oct-tree represent the cells, and the leaves represent the bodies in the corresponding cells. Each time step consists of the following key phases: A process traverses the tree to obtain a set of bodies that results in good load balancing between processes; then it computes the forces on these bodies; and finally it computes the new positions of the bodies. Barriers separate these different phases.

3D-FFT

3D-FFT comes from the NAS benchmark suite [9] and numerically solves a partial differential equation using forward and inverse FFTs. Assuming the input array \( A \) is \( n_1 \times n_2 \times n_3 \) and organized in row-major order, the array elements are distributed along the first dimension of \( A \). For any \( i \), all elements of \( A[i, *, *] \) are contained within a single process. A 1-D FFT is first performed on the \( n_1 \times n_2 \) point vectors, then on the \( n_3 \times n_1 \) point vectors. In these phases, each process computes its portion of the array without any communication. A barrier separates these first two phases from the third and final phase, which is a transpose followed by a 1-D FFT on the
During the transpose, with \( n \) processes, each process needs to read \( \frac{1}{n} \) of its data from each of the other processes.

**Jacobi**

Jacobi is a simple iterative method for solving partial differential equations. The algorithm employs two arrays - one is the data array and one is the scratch array. There are two phases in each iteration: First, each element is updated according to a four-point stencil and the new values are stored in the scratch array, then the scratch array is copied to the data array. Both loops are parallelized. The first loop requires nearest neighbor communication, in which neighboring processes exchange their boundary columns in the data array.

**MGS**

Modified Gramm-Schmidt (MGS) computes an orthonormal basis for a set of \( N \)-dimensional vectors. At each iteration \( i \), the algorithm first sequentially normalizes the \( i \)th vector, then makes all vectors \( j > i \) orthogonal to vector \( i \) in parallel. The vectors are divided over the processes in a cyclic manner to balance the load in each iteration. All processes synchronize at the end of an iteration.

**NBF**

Non-Bonded Force (NBF) is the kernel of a molecular dynamics simulation and originates from the GROMOS benchmark [81]. For each molecule \( m \), the program keeps a list of interacting “partners”, other molecules that are close enough to exert a non-negligible effect on \( m \). The program goes through all these lists of partners and updates the forces on each node based on the distance between them. At the end of each iteration, the coordinates of the molecules are updated according to the force acting on them. Each molecule has approximately the same number of partners, so a simple block-partitioning of the molecules among the processors suffices to balance the load.

**Quicksort**

Quicksort uses a centralized task-queue based approach to sort an array of integers. Initially, the entire array is inserted in the task queue. A process repeatedly dequeues a sub-array to be sorted from the queue and recursively applies the quicksort algorithm to the dequeued element. The algorithm partitions the dequeued element into two sub-arrays around some chosen pivot. The smaller partition is enqueued in the task...
queue and the process continues to work on the larger partition until its length falls below a certain threshold, then it is sorted locally.

**SOR**

Successive Over-Relaxation (SOR) is a simple iterative relaxation algorithm. The input is a two-dimensional grid. During each iteration, every matrix element is updated to the average of the four neighboring elements. To avoid overwriting an element before neighbors use it for their computations, a “red-black” approach, wherein every other element is updated during the first half-iteration and the rest of the elements are updated during the second half-iteration, is used. The work is parallelized by assigning a contiguous chunk of rows to each process. Exchange of data between processes is therefore limited to those pages containing rows on the edge of the chunks. Barriers are used to synchronize all processes at the end of each half-iteration.

**TSP**

TSP solves the traveling salesman problem using a branch-and-bound algorithm. The major data structures are a pool of partially evaluated tours, a priority queue containing pointers to tours in the pool, a stack of pointers to unused tour elements in the pool, and the current shortest path. A process repeatedly dequeues the most promising path from the priority queue, extends it by one city, and enqueues the new path, or takes the dequeued path and tries all permutations of the remaining nodes.

**Water**

Water is a molecular dynamics simulation from the SPLASH benchmark suite [74], evaluating forces and potentials in a system of water molecules in the liquid state. At each timestep, every molecule’s velocity and potential is computed from the influences of other molecules within a spherical cutoff range. Several barriers are used to synchronize each timestep, while locks are used to control access to a global running sum and to each molecule’s force sum. The main shared data structure is a large one-dimensional array of molecules. Equal contiguous chunks of this array are partitioned to each process.

**Application diversity**

Table 5.2 lists the application codes used in our evaluation, including their type according to the classification of Section 5.2. The input and the runtime characteristics
<table>
<thead>
<tr>
<th>Application</th>
<th>Input</th>
<th>Data layout</th>
<th>Shared memory (MB)</th>
<th>Iterations or tasks</th>
<th>Number of adaptation points (AP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>32768 bodies</td>
<td>irregular</td>
<td>6</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>128 x 64 x 64</td>
<td>block</td>
<td>42</td>
<td>100</td>
<td>505</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2500 x 2500</td>
<td>block</td>
<td>47.8</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>MGS</td>
<td>3072 x 3072</td>
<td>block-cyclic</td>
<td>48</td>
<td>3072</td>
<td>3072</td>
</tr>
<tr>
<td>NBF</td>
<td>131072 atoms, 80 partners</td>
<td>irregular</td>
<td>52</td>
<td>100</td>
<td>202</td>
</tr>
<tr>
<td>Quicksort</td>
<td>10'000'000 integers, bubble threshold 32768</td>
<td>independent</td>
<td>38.3</td>
<td>about 613</td>
<td>about 77</td>
</tr>
<tr>
<td>SOR</td>
<td>2500 x 5000 floats</td>
<td>block</td>
<td>58.7</td>
<td>100</td>
<td>200</td>
</tr>
<tr>
<td>TSP</td>
<td>19 cities</td>
<td>independent</td>
<td>0.4</td>
<td>about 618</td>
<td>about 77</td>
</tr>
<tr>
<td>Water</td>
<td>1728 molecules</td>
<td>block</td>
<td>1.5</td>
<td>20</td>
<td>86</td>
</tr>
</tbody>
</table>

Table 5.2: Application suite

shown are the “default” values used in all tests throughout this thesis, except where otherwise mentioned.

In Quicksort and TSP, the total number of tasks may vary somewhat from run to run, and the number of adaptation points corresponds to a 8-process-runs with each process solving about \( \frac{1}{8} \) of the tasks, whereas in the other programs the number of adaptation points does not depend on the number of processes used.

The application suite covers a wide variety of data layout and data sharing patterns and parallel programming styles, ensuring that the conclusions of our studies are applicable to a large class of applications in general.

5.4 Data sharing patterns

In this section we give an overview of the data sharing patterns of the nine programs we tested.

3D-FFT, Jacobi, SOR and Water have a block data distribution. To be precise, in 3D-FFT and Jacobi threads each have several smaller blocks of data akin to a block-cyclic layout instead of one large contiguous block, but the essential point is that these blocks’ sizes depend on the total number of processes and not on the input size, so the allocation is best described as a series of block data distributions. All conclusions we draw for block data layouts are applicable to these applications.

Jacobi and SOR have nearest-neighbor sharing, as processes use their partitions’ data and only exchange data at the edge of their partitions. The only exceptions are initial cold misses and the transfer of the slaves’ results to the master at the end.

3D-FFT has well-balanced all-to-all sharing of data, while in Water, each pro-
cess shares all data in its partition with several neighbor processes, about half of all processes, several times per iteration, producing a “banded” sharing pattern.

MGS has a block-cyclic data distribution, as the block size is defined by the problem set size. Each block contains one row of the input-matrix, and every nth row is allocated to each one of n processes. MGS typically has no data sharing among slave processes, only master-to-all and all-to-master transfers (although this property is not valid for arbitrary matrix sizes).

Barnes and NBF have more irregular data distributions. In Barnes, about 70% of all allocated shared memory pages p are accessed by the majority of processes in each iteration, whereas the other 30% are exclusively used by the master process, if at all. 50% of p experience false sharing and hence use the multiple writer protocol; their consistency updates are done mainly by diffs and page states frequently change back and forth between invalid and shared. The other 20% of p use the single-writer protocol, these pages are paged-in anew in every iteration by most slaves. In NBF, only about 5% of all allocated shared memory pages are repeatedly accessed by several processes, these pages are in most cases paged-in once per iteration by most of the slave processes. All other page transfers are one-time cold misses. Given such extensive sharing of at least some pages among processes, both Barnes and NBF have all-to-all sharing patterns.

In each OpenMP fork of n threads, the data space of the above applications is allocated anew among the n threads, using each thread’s identifier and the total number of threads n.

Quicksort and TSP have no predefined data distribution, as the allocation of tasks to processes is not predetermined. These applications are therefore characterized by quite random all-to-all data sharing patterns. In Quicksort, data is both read and written during partitioning and during sorting. As these steps are performed independently of each other, the same data is often accessed by several threads. In TSP, as one partial tour fills only a fraction of a shared memory page, different threads solving tours residing in the same page typically cause each page to be shared read-only by many threads.

Of the programs tested, only Barnes has much false sharing. All other programs have little or no false sharing, depending on the number of processes used and the problem set sizes, so in these applications very few pages use the multiple-writer protocol.

When pages used by more than one thread are both read and written and no false sharing occurs, the page state changes to or remains exclusive and the single-writer protocol is used. When this sharing behavior applies to all or most of the pages in use, each one of n threads has about \( \frac{1}{n} \) of all pages exclusively-valid. This is characteristic
for most of the programs tested, namely for Jacobi, SOR, Water, NBF and Quicksort. 3D-FFT is similar, except that the master process’ share is much larger at the expense of the slave processes’ shares. The other three programs have either much read-only sharing (MGS, TSP) or false sharing (Barnes), so each one of $n$ processes often has much more than a $\frac{1}{n}$ share of pages valid, many of which use the shared state, only few are exclusive.

### 5.5 Adaptation cost measurement methodologies

#### 5.5.1 Adaptation cost

To evaluate the performance of our adaptive system, we define and measure the cost of an adaptation. The user running a parallel application is interested in the optimal use of available resources, i.e., an application’s runtime and its resource usage. We therefore define the adaptation cost as the *difference in time between the adaptive run and an “equivalent” non-adaptive run*. We consider two runs as “equivalent” when the *same average number of CPUs* are used and all environment parameters are equal. In our tests, we always assigned each thread to a different CPU. In that case, comparing runs using equal average numbers of CPUs means comparing runs with the *same average number of threads*.

With the above approach, we have in most cases non-integer averages. Consider a single adaptation from 8 to 7 threads. Such a run uses 7.x threads on average, which we cannot immediately compare to some non-adaptive run. We have devised two methods to overcome this problem, described in the next 2 sections.

We do not quantify the benefits of the added *flexibility* our system offers over the standard non-adaptive system. For example, in a multi-user cluster where users needing only single machines are given priority, a parallel application could not be run without our system, because it might otherwise occupy some machines when new users who wish to have exclusive access to these machines arrive.

#### 5.5.2 Method A: Overview of costs

In an adaptive run, we measure separately the runtime of all non-adaptive parts during the execution and record the number of threads in each such part. With this, we calculate the average number of threads $avg_{\text{adaptive}} = n.m$ used for the whole run of duration $t_{\text{adaptive}}$. We now construct a polynomial trendline of the runtime $t(n)$ as a function of the number of threads $n$ using the runtimes of integer values of $n$ from non-adaptive runs. The curve of this function gives us “theoretical” non-adaptive
5.5. ADAPTATION COST MEASUREMENT METHODOLOGIES

runtimes $t_{\text{non-adaptive}}$ for non-integer averages $n.m$ of threads. The difference between $t_{\text{adaptive}}$ and $t_{\text{non-adaptive}}$ reflects the adaptation cost.

5.5.3 Method B: Detailed results for single adaptations

The idea of our second approach is to compare only selected parts of runs where equal work is performed by an equal (integer) number of threads in both cases, i.e. without any adaptation in-between. In these intervals, we measure many parameters such as the runtime, the number of page and diff transfers, the number of messages and bytes sent, the number of garbage collections performed, and the number of multiple-writer pages.

For an adaptation from $x$ to $y$ threads, we begin our measurements by resetting all statistics to zero during sequential execution at the beginning of the OpenMP fork performing the adaptation. At that time, the data layout is still that of an $x$-thread-run, but no more work is executed before the adaptation, rather all measured work is performed post-adaptation by the $y$ threads, and no further adaptation is performed until the measurements are stopped. These statistics thus completely contain all effects of the adaptation, as we start collecting statistics just before the adaptation is initiated. From this data we subtract the results of a non-adaptive run of $y$ threads, where we only measure statistics while exactly the same work is performed as in the adaptive run, so the difference reflects the adaptation cost.

We determine equal work as follows: Data parallel applications have the nice property that the percentage of work completed up to any given sequential section is independent of the number of threads used in any of the preceding parallel sections. We must therefore only ensure that we take measurements of exactly the same parallel sections in the adaptive and the non-adaptive case. For this, we count the number of OpenMP forks executed.

For task parallel applications, the procedure is slightly more complex, as each thread solves tasks at its own individual pace. In addition, the amount of work performed by individual tasks and the total number of tasks may vary. For simplicity, we however make do with an approximation: We assume that (1) the average size of a task during the post-adaptive part we measure, and (2) the total number of tasks for the complete run both do not vary significantly for any two runs we compare\textsuperscript{1}. This permits us to declare that equal work means (approximately) an equal number

\textsuperscript{1}Our measurements confirm that in the applications tested the total number of tasks varied only minimally (less than 1% in nearly all cases), validating assumption (2). Further, given a constant overall amount of work, (2) means that the average size of a task overall remains virtually constant for all tested scenarios, i.e. for many different numbers of processes, so an adaptation does not influence the average size of a task, so assumption (1) suggests itself.
of completed tasks. We let the application complete the same specifiable number of tasks in both runs, whereby we assign each of the threads a (more or less) equal share of the tasks, upon which we execute an extra OpenMP join and fork, where we reset the statistics as described above and perform the adaptation or continue without adapting. Assumption (2) means that after the adaptation the two runs that are compared both have an (approximately) equal number of tasks left to solve.

Obviously, with the above setup, some threads may wait a significant amount of time at the adaptation point, as not all threads are assigned the same number of tasks. For example, for an adaptation from 7 to 8 threads after 100 tasks, we let 5 threads do 14 tasks each and 2 threads 15 tasks each, as $5 \times 14 + 2 \times 15 = 100$. However, as we only begin our measurements in the OpenMP fork, any such preceding delays are irrelevant.

**Interval of measurements**

We introduce the notion of adaptation point-interval or *AP-interval* for the interval between any two successive adaptation points.

We obtained all the results for single adaptations (i.e. one or more adapt events performed at the same adaptation point) throughout this thesis by using method B, as this method provides us with much more precise timing results than method A: For method B, we collected statistics over short intervals $i_{\text{measured}}$ of typically 10-20 seconds duration, by terminating some of the data parallel applications prematurely after the adaptation. For such cases, we measured all relevant statistics between every two successive adaptation points after the adaptation and thereby determined the maximum extent $i_{\text{adaptation-effect}}$ of AP-intervals during which the adaptive and non-adaptive runs are not completely identical. By choosing an interval $i_{\text{measured}} \geq i_{\text{adaptation-effect}}$, we captured all the adaptation costs in our measurements. Runtime-variations of a half to one percent for identical successive runs are not uncommon. For longer runs, such variations can easily exceed the cost of an adaptation, so comparisons between two such runs become very inaccurate, but by using shorter $i_{\text{measured}}$ intervals, we avoid this problem.

With the above measurements, we do not only determine the length of the $i_{\text{adaptation-effect}}$ interval, but we can also precisely state in which of the AP-intervals which adaptation costs (or even benefits) occur. We use such detailed measurements in Section 6.5.

**Validation of method B**

We validated the timing results obtained by method B with results obtained by method A. As reported in Section 6.2, we performed various representative multiple
adaptations during the course of an application's run. Using method A, we observe that the average cost per adapt event in these tests is always within a few percent of the results derived via method B, such as those shown in Section 6.4, when comparing and averaging the same adaptations.

Validation of approximations for task parallel applications

We observe that in TSP tasks become shorter towards the end, as the cost of the least expensive complete tour found at any given time declines and searches are always pruned by abandoning any partial tours whose cost exceeds this value. Similarly, we observe that in Quicksort some tasks executed early on are more expensive than all others, namely those that perform splitting and sorting. Later on, when all subarrays are smaller than a given threshold and no more splitting is required, tasks only perform sorting. Further, the subarrays and therefore the task sizes vary significantly in size, as they are split up along random pivot-values. Finally, as the size is random but the threshold above which a further split is performed is fixed, the number of splits, i.e. the number of tasks, may vary as well.

However, as we only compare approximately equal parts of adaptive and non-adaptive runs, these effects are on average the same in both cases. Our measurements confirm that all of the above variations are small for any given parts of runs used in our tests. In addition, we validate these results by using method A, as stated above. Finally, our results are more than precise enough for our purposes, as our goal is to demonstrate that the order of magnitude of adaptation costs is low - small variations in cost do not affect our conclusions.

5.6 Experimental setup

5.6.1 Individual adaptations

For experimental purposes only, we extended the system with code that lets a user specify exactly at which adaptation point or after how many completed tasks (1) which adapt event(s) shall be performed, and (2) statistics are to be reset. This extension is needed for method B described in Section 5.5.3, so that the same experiments can easily be repeated many times and the correct adaptive and non-adaptive cases can be compared.
5.6.2 Periodic adaptations

To get an idea of the system’s overall performance, we added code that forced the system to perform an adaptation after every \( a \) adaptation points or after every \( a \) tasks solved on the master thread, for any \( a \).

Starting with \( n + 1 \) threads, we alternately let the system perform a leave, then a join, another leave, another join, etc., resulting in an average of about \( n.5 \) threads. Starting with a leave of thread number one (the first slave thread), for each successive leave, we let the thread with the next-higher id leave, and after the last thread, we “wrap-around” to the first slave thread again.

**Reasoning**

By letting a slave thread leave anew only after all other slave threads have left meanwhile, we maximize the minimal time \( t \) any thread participates in the computation, so the chance that part of the thread’s previous join cost has not occurred yet is minimized. In general, the longer a thread is active, the more different pages it touches, so more pages must be moved when the thread leaves. This is similarly true for the remaining threads, as a join of thread \( x \) followed by a leave of the same thread generally means the other threads return to the data layout before \( x \)’s join. The sooner this happens, the more likely it is that not all pages of the intermediate data layout (after \( x \)’s join) have actually been moved, so less restoring is needed after \( x \)’s leave. Although Section 6.5 shows that such considerations are not very relevant for the applications tested, as all data redistribution occurs very soon after an adaptation, the chosen strategy nevertheless produces upper-bound costs for any other applications for which such considerations are relevant, as our goal is to demonstrate that for any worst-case scenarios costs generally are small.

The strategy described produces realistic cost estimates for any scenario that does not predominantly have only exceptionally cheap or expensive leave events (e.g., for a given application, the leave of a middle thread may cost far less than an average leave, and only such threads leave), but for such cases, the overall costs may easily be estimated by measuring individual adaptation costs.

5.7 Test scenarios

Quantifying the cost of adaptations is not straightforward, as the costs are determined by numerous different parameters. We have identified the most important parameters where a variation is possible, and in Chapter 7 we analyse in detail each one’s influence on the adaptation cost.
5.7. TEST SCENARIOS

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Range</th>
<th>Representative or default value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of threads</td>
<td>1 to 32+</td>
<td>8</td>
</tr>
<tr>
<td>Id of leaving thread (n threads)</td>
<td>1 to (n-1)</td>
<td>around (3/4*u)-1</td>
</tr>
<tr>
<td>Id of leaving thread for n=8</td>
<td>1 to 7</td>
<td>5</td>
</tr>
<tr>
<td>Batching of adapt events (n threads)</td>
<td>1 to 2*(n-1) events</td>
<td>1 event</td>
</tr>
<tr>
<td>Progress of application</td>
<td>0-100% of execution</td>
<td>varies with application</td>
</tr>
<tr>
<td>Shared memory use</td>
<td>4K to 256MB</td>
<td>about 40 to 50MB</td>
</tr>
<tr>
<td>Network</td>
<td>100Mbps or 1Gbps</td>
<td>100Mbps</td>
</tr>
</tbody>
</table>

Table 5.3: Parameters affecting adaptation costs, the range of values each parameter may assume, and representative values used in most of the tests. N threads have ids 0 to (n-1).

We used the following strategy to obtain results for such an analysis: For each parameter, we constructed a series of tests covering a major extent of the range of values possible for this parameter while at the same time keeping the other dimensions constant at some representative, typical values. With these results, we can more or less extrapolate the performance for all the applications tested and any combination of parameters. We can also roughly predict the performance of any other applications which can be categorized according to the applications tested. As we are more interested in orders-of-magnitude statements than very precise results for a specific application run, this strategy is sufficient for our purposes.

Table 5.3 lists the parameters we varied in our tests, the potential range of values for each parameter, and the default values used while one of the parameters was varied.

5.7.1 Default test values

For practical reasons (number of available machines, speedup), we limited our tests to a maximum of 32 threads, although the theoretical maximum is much higher. We chose 8 threads as the default degree of parallelism, as we expect this to be a typical average value when executing the applications we tested: Most of these applications achieve good speedups for up to around 8 threads, but less so for significantly more threads. With 8 threads, 5 of the 9 programs achieve speedups > 5.5 or > 4.5 with the 1Gbps resp. with the 100Mbps network (Section 6.1). For a leave event in 8-thread-runs, we chose the sixth thread because it is in-between the minimum and the maximum cost for a leave, for those applications that exhibit a significant variation of the leave cost for different threads leaving (Section 7.5.2). In most tests, we performed one adapt event instead of several when adapting, as we are interested in the cost per event. If anything, this cost falls sharply with a batching of events (Section 7.5.4), so our one-event results are upper-limit results. Further, for each
application, we selected a certain adaptation point $AP_{adapt}$ or, for the task parallel applications, some number of completed tasks, where or after which we performed most of the adaptations. Most applications experience only small adaptation cost variations during execution (Section 7.5.1), so for these programs we used an $AP_{adapt}$ relatively close to the beginning, but not so close as to benefit from startup-effects. In MGS however, we adapted after exactly 50% of the iterations, and in Quicksort and TSP we likewise adapted upon completion of about half the tasks. In these three applications, the cost of a given adaptation varies considerably during execution, but the cost at the half-way point is close to the average, as Section 7.5.1 shows. For all applications, we established a reference problem set shown in Table 5.2 using less than 64MB of memory. We used this input for all tests except those for Section 7.5.6. Finally, we ran the tests by default on the 100Mbps network. The results on the 1Gbps network are much better and therefore closer to or within the margin of error of the measurements, rendering a precise analysis more difficult.

### 5.7.2 Representative subset of applications

We chose a representative subset of 4 applications for many of the experiments: Jacobi, 3D-FFT, NBF and Quicksort. Most of our applications are regular, so we selected two from this class (Jacobi and 3D-FFT), plus an irregular one (NBF) and an independent one (Quicksort). Jacobi has the highest adaptation costs of the longer-running (at least several minutes) applications, and this cost is not much less than the one of SOR, which has the highest cost overall. 3D-FFT has a different data-sharing pattern than Jacobi. NBF is the more expensive and longer-running of the two irregular applications tested, and for the independent programs the same is true for Quicksort. Selecting the more expensive applications means that the other applications, if anything, fare better, so our results are more upper-bound than average results.

Having a few representative adaptation cost results of all applications, we can more or less extrapolate the performance for other scenarios of the other applications by observing how the costs for these 4 applications vary for the scenarios tested.

### 5.7.3 Dimensions examined

Below we list the goals for our tests and the parameters we varied to quantify separately each parameter’s influence on the overall adaptation cost, and we explain what experiments we used to this end.

**A. Overview of adaptation overhead**

Using the representative set of 4 applications described in the above section and the test strategy outlined in Section 5.6.2, we performed repeated adaptations
during an application’s execution. As all slave threads left alternately, we have “average” scenarios where no thread was preferred. In successive test runs, we varied the interval between adapt events.

B. Comparison of all applications

For each of the 9 programs, we ran a series of tests with exactly one adaptation each or with no adaptation and calculated the adaptation cost according to the method outlined in Section 5.5.3. We performed 3 different adaptations: (1) a join from 7 to 8 threads, (2) a leave of the sixth thread, adapting from 8 to 7 threads, commensurate to Table 5.3, and (3) a simultaneous combination of (1) and (2), i.e. an adaptation from 8 to 8 threads. We adapted in this form at 5 different points during each application’s course of execution, choosing adaptation points or, for the task parallel applications, some number of completed tasks at which an application has done about $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$ or $\frac{5}{6}$ of its work. We average the results per application and adaptation type as a basis for comparing the different applications with each other, although most applications show little adaptation cost variations at different adaptation points.

For adaptations with no change in the number of processes, we use the term *move* to refer to one join and one leave event performed in the same adaptation, i.e. the adaptation type (3) above corresponds to one move.

C. Length of adaptation effects

Using the representative set of tests described in item B above, we measured the accumulated adaptation effects for data parallel resp. task parallel applications, at each adaptation point following the adaptation, resp. after each task executed by the master following the adaptation. With these results, we can at the same time examine how the length of the *interval between successive adapt events* has an effect on the adaptation cost.

D. Variation of cost during execution

We used the representative set of tests described in item B above to observe variations between different adaptation points.

E. Identifier of joining or leaving thread

For each of 5 representative applications (Jacobi, 3D-FFT, MGS, NBF and Quicksort), we performed series of tests with single adaptations from 8 to 7 threads with one thread leaving each time. We let each one of the 7 slave threads leave in turn, at one selected adaptation point per application.
include MGS to highlight how the issue of thread identifier choice is much less important when using block-cyclic data layouts, compared to block data layouts.

We present less results for joins for 2 reasons: (1) The data redistribution for corresponding joins and leaves is similar, except that the data flows in the opposite direction, so the costs are also very similar, as discussed in Section 7.4. (2) The identifier for the joining thread(s) is selected automatically by the system and not the user. (2) ensures that joining threads are assigned identifiers which reduce the adaptation cost whenever different possible assignments would lead to significantly different adaptation costs, so we do not evaluate other more expensive hypothetical scenarios which the system never chooses.

F. Number of threads

For each of the 9 applications we performed tests with one adaptation per run at one selected adaptation point per application, for 4, 8, 16 and 32 threads after the join or before the leave events. In each case, we adapted with either (1) one join event, or (2) one leave event of the end thread, or (3) one leave event of a middle thread. With (2) and (3) we can observe the variations between expensive and cheap leaves for those applications that have such variations. Further, we performed batched adapt events for the 8, 16 and 32-thread runs: We let either 2, 4 or 8 threads join or 2, 4 or 8 end threads or middle threads leave, all at the same adaptation point.

The results of the single join or leave events show how one adapt event becomes cheaper for more threads, whereas a comparison of the results of the 1, 2, 4 and 8-event adaptations for the 4, 8, 16, and 32-thread runs show how adaptations concerning the same share of data or threads, i.e. always one quarter of the threads, compare for different numbers of threads.

G. Batching of adapt events

For each of 4 representative applications (Jacobi, 3D-FFT, NBF and Quicksort), we performed adaptations with 1, 2, 3 or 4 joins, 1, 2, 3 or 4 leaves, and 1, 2, 3 or 4 moves. We ran the tests with 8 processes after the joins or before the leaves or before and after the moves, and we selected one adaptation point per application for all tests. For the leaves and the moves, we measured minimum and maximum values, as these are highly dependent on which of the processes are chosen. For the minimum values, we chose as much as possible an even distribution of the leaving processes within the id-space, such as every second process leaving for 4 leaves, whereas for the maximum values, we clustered all leaving processes together at the end of the id-space, such as ids 4 to 7, with
ids 0 to 3 remaining.

H. Problem set size

For each of 4 representative applications (Jacobi, 3D-FFT, NBF and Quicksort), we performed separate tests with either one join from 7 to 8 threads or one leave of the sixth thread from 8 to 7 threads, at one selected adaptation point per application. Where possible, we repeated these measurements for 4 different inputs per application, varying the total amount of shared memory use by factors of 4: Besides using the default sizes specified in Table 5.2, we reduced the sizes to $\frac{1}{16}$ or $\frac{1}{4}$, or we increased the size by 4 or by 2 for NBF.

I. Regular versus irregular applications

To study how adaptation costs in regular and irregular applications may differ (for similar problem set sizes and adaptations), we chose two regular and one irregular application (Jacobi, 3D-FFT and NBF) and performed one join or one leave event at one selected adaptation point per application, for 3 different numbers of threads each, specifically for 4, 8 and 16 threads after the adaptation. The precise adapt event is irrelevant, as we are concerned with the synchronization delay preceding the actual adaptation: We measured the idle time of each thread at the OpenMP join where the adaptation was performed.

J. Pre-adaptation delays and costs of task parallel applications

For Quicksort and TSP, we ran tests with one adaptation each (the exact type of adaptation is irrelevant for these results) at 5 different adaptation points, as in item B, using 8 threads. In each case, we measured both the pre-adaptation delay (the time from the arrival of the adapt request to the beginning of the adaptation) and the pre-adaptation cost (the average of all threads’ idle times at the adaptation’s OpenMP join).

We measured worst-case scenarios by generating the adapt event just after the master thread had started a new task. By using 5 different adaptation points, we take into account variations of task lengths and thus variations of pre-adaptation delays and costs throughout a run. As these delays and costs vary with the task lengths, we can estimate their values for other numbers of threads. Given runtime, total number of tasks and average number of threads of a run, we know the average task length.

K. Test environment

We performed most of the tests using the 100Mbps network, permitting a more precise analysis due to higher adaptation costs. Selected results from the 1Gbps
network show how the results scale for the latter network.
Chapter 6

Performance overview

This chapter provides an overview of the performance of the programs presented in the previous chapter, both without and with adaptations. We show not only the overall costs of sample adaptations, but also examine how quickly the system reacts to adaptation requests and for how long an adaptation affects the on-going computation. The following chapter then analyses adaptation costs in more detail.

6.1 Non-adaptive performance

In this section, we study the performance the selected applications achieve in the absence of any adaptations, and we show how frequently each application is able to adapt. Attractive speedups are obviously a prerequisite for any application to be considered for an adaptively-parallel execution, or else a sequential run will be chosen. Further, users preferably want the system to perform a requested adaptation with little delay, especially leave requests.

Although an adaptation request may be received anytime, adaptations can only be performed at adaptation points. To determine the responsiveness of the system to such requests, we therefore determined how many adaptation points each application encounters per second, on average, i.e. how frequently applications have an opportunity to adapt. We show results for 4, 8 and 16 threads in Table 6.1. In data parallel applications, these numbers are simply the number of OpenMP join-fork sequences executed per second. As the number of processes rises, the total runtime usually decreases, so less time is needed for each parallel section, so the response time improves correspondingly. In task parallel applications however, the opposite is true: For Quicksort and TSP, the table shows the average number of tasks solved per second by one process, as the system has the opportunity to execute an OpenMP join-fork sequence at task boundaries only. As the number of processes rises, the contention in the system rises and the average time a process spends on a task increases, i.e. the
Table 6.1: Number of adaptation points per second (data parallel programs), or average number of tasks solved per second per process (task parallel programs).

Figure 6.1: Speedups on 100Mbps Ethernet, one thread per machine (left), and two threads per machine (right).

speedup is less than ideal, so the response time deteriorates.

Figures 6.1 and 6.2 show the speedups achieved by the applications we tested, when using the 100Mbps network and the 1Gbps network, respectively. For each network, we ran two scenarios: In one case, we ran one process per machine while in the other case, we ran two processes per machine. As we have dual-processor machines, the second scenario obviously makes better use of the CPU resources. However, as each machine is connected via one network link to the switch, this case leads to network contention. A comparison of the two scenarios illustrates the vital importance of the network topology, as the speedups in the two-processes-per-machine case are much lower, although the reduction is very application-dependent.

As we do not have results for 32-process-runs with one process per machine for this environment, we omit 32-process-speedups in the graphs. Our results for 32 processes with two processes per machine in most cases show better speedups than with 16 processes and two processes per machine, but worse speedups than with 16 processes.
6.2 Overview of adaptation effects on total runtime

This section provides an idea of the overhead of adaptations, while details of individual adaptations are discussed in later sections.

We periodically caused an adapt event to occur during the execution of an application, using the setup described in Section 5.6.2. For the evaluation, we used the techniques of method A (Section 5.5.2): For adaptations between \( n \) and \( n + 1 \) threads, we first calculated the theoretical runtime of a non-adaptive run of \( n \)\( .5 \) threads, using

Figure 6.2: Speedups on 1Gbps Ethernet, one thread per machine (left), and two threads per machine (right).

and only one process per machine, due to the network contention.

For the adaptation measurements throughout this thesis, we have always used one thread per machine and network link for all runs of up to 16 threads, thereby avoiding the not easily quantifiable extra overhead induced by network collisions. Nevertheless, our conclusions in Section 7.7 are valid for different network topologies, as we do not state absolute numbers but rather make comparative statements for various scenarios in a given environment. For example, a less favorable network layout affects both the absolute cost of adaptations and the absolute runtime of an application, so the relative costs as a percentage of the total runtime remain similar. In particular, the data transfers (messages and bytes) are practically equal regardless of network topology, only the times differ.

Different execution platforms may allow better results. Cox et al. have obtained much better speedups for some of the applications we used [25], probably due to an optimized network and somewhat slower CPUs, compared to our setup. We expect adaptation costs as a percentage of total runtime to improve likewise in that environment, compared to our setup (cf. Section 7.5.10).
Figure 6.3: Execution times for different intervals between adapt events and different average numbers of threads, on 100Mbps network (left) and 1Gbps network (right).
a trendline $t(x)$ of the runtime as a function of the number of threads $x$, through $t(n)$ and $t(n+1)$. We now scaled the adaptive runtimes to the same average of $n.5$ along the same trendline. This scaling is necessary for a comparison of different adaptive runs having somewhat different averages: For example, given two adaptive runs of 7.4 and 7.6 threads on average, the second run’s measured time may be less than the first one’s time due to the better speedup with 7.6 threads, even though this run has more adaptations than the other run of 7.4 threads. A scaling of both times to an average of 7.5 typically reverses such aberrations. Finally, we calculated the average interval between successive adaptations using the scaled runtime divided by $a.5$, for $a$ adaptations. Assuming that (on average) an application ends about halfway between the last and a potential next adaptation, we add 0.5 to the number of adaptations.

Figure 6.3 shows how various frequencies of adapt events, from no adaptations to an adaptation every few seconds, affect the overall runtimes of 4 representative applications (cf. Section 5.7.2): One adaptation event about every 10 seconds causes a runtime increase of around 10-20% in the 100Mbps environment, and 5-10% in the 1Gbps environment. In general, with more threads used, the adaptation costs decrease more than the runtimes, so the adaptation overhead as a percentage of the runtime decreases somewhat. Further, we observe an approximately proportional decrease/increase in cost for a decrease/increase in adaptation frequency, as every adaptation adds about the same delay to the overall runtime, so the cost per adaptation stays constant within a few percent, regardless of the interval length between events. This observation is consistent with the results of Section 6.5.

### 6.3 Response time for adaptation requests

In this section we study the question of how responsive the system is for the applications tested, i.e. how quickly the system begins with an adaptation after receiving a request. This issue is of particular interest for leave events, as one may wish to withdraw a machine as soon as possible from the computation. Join events are generally less urgent, the inclusion of a new machine may wait a little while, as no other user is affected.

The delay $t_{react}$ between receipt of an adapt request and the actual beginning of the adaptation is at most the length of the current AP-interval or the longest task currently being solved. The maximum delay occurs when the request arrives just after the beginning of this interval or task.

Table 6.2 shows $t_{react}$ times: The first three columns (for each of the two networks) show typical values derived from measurements taken at 5 different points throughout the execution of each application, to be precise, after about $\frac{1}{6}$, $\frac{2}{6}$, $\frac{3}{6}$, $\frac{4}{6}$, or $\frac{5}{6}$ of the
<table>
<thead>
<tr>
<th>Application</th>
<th>100 Mbps network time (seconds)</th>
<th>1 Gbps network time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shorter</td>
<td>Longer</td>
</tr>
<tr>
<td>Barnes</td>
<td>1.94</td>
<td>2.02</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>0.12</td>
<td>0.14</td>
</tr>
<tr>
<td>MGS</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>Jacobi</td>
<td>0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>NBF</td>
<td>3.06</td>
<td>3.07</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.07</td>
<td>0.43</td>
</tr>
<tr>
<td>SOR</td>
<td>0.03</td>
<td>0.05</td>
</tr>
<tr>
<td>TSP</td>
<td>0.07</td>
<td>0.10</td>
</tr>
<tr>
<td>Water</td>
<td>1.12</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Table 6.2: Average and maximum delays $t_{\text{react}}$ (seconds) between the receipt of an adapt request and the beginning of the adaptation, for 8-thread-runs. The “Shorter”, “Longer” and “Average” columns show how these averages vary at 5 different adaptation points during an application’s execution. The “Maximum” column shows the worst-case delay of a request arriving at the beginning of the longest AP-interval or task of the application.

work done. The numbers represent the minimum, maximum and average $t_{\text{react}}$ times for these 5 points (averaged over many identical test runs of 8 threads). The fourth column “Maximum” shows the maximum $t_{\text{react}}$ times encountered anywhere during an application run (averaged over many 8-thread-runs). This value is therefore a worst-case delay, namely the length of the longest AP-interval or task of a complete application run.

We computed the $t_{\text{react}}$ times shown in the “Shorter”, “Longer” and “Average” columns in Table 6.2 as follows: We measured the lengths of several AP-intervals or tasks executed close to each of the 5 selected points. For the data parallel applications, we then calculated the average AP-interval length $i_{\text{avg},\text{len}}$ at each of these 5 points. For those applications where AP-intervals vary only little in length, we set $t_{\text{react}}$ equal to half of the minimum, maximum resp. average $i_{\text{avg},\text{len}}$, because on average, a request arrives halfway in-between potential adaptation points. For applications with a significant variation of AP-interval lengths, we calculated $t_{\text{react}}$ accordingly as somewhat more than $\frac{i_{\text{avg},\text{len}}}{2}$\(^1\). For the task parallel applications, we calculated the full length of the average task executed at the selected point, and set $t_{\text{react}}$ equal to the minimum, maximum resp. average value thereof: When a request arrives, the last

---

\(^1\)If the length of AP-intervals varies widely throughout the execution of a program, then the average $t_{\text{react}}$ is more than half the average length of an AP-interval: Roughly speaking, the probability that a request arrives in an AP-interval of above-average length rises with increasing length of such an AP-interval, increasing the average of $t_{\text{react}}$.\)
### 6.4. SELECTED ADAPTATION COSTS

<table>
<thead>
<tr>
<th>Application</th>
<th>100 Mbps network time (seconds)</th>
<th>1 Gbps network time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shorter</td>
<td>Longer</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.11</td>
<td>0.24</td>
</tr>
<tr>
<td>TSP</td>
<td>0.03</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Table 6.3: Pre-adaptation costs for task parallel applications, for 5 different adaptation points throughout each application’s execution, for 8-thread-runs.

process to finish its current task determines the delay, so this process is probably one that has just started a task.

We see in Table 6.2 that the system typically begins an adaptation within a second after a request is received, in many cases already after a fraction of a second. Only Barnes and NBF have a delay of on average up to 2 or 3 seconds. The “Maximum” column shows that an adaptation is always begun within about 6 seconds for all of the applications tested.

With the results of Table 6.2 and the speedups presented in Section 6.1, we can estimate $t_{react}$ for other numbers of threads. As we have already seen in Section 6.1, for data parallel applications $t_{react}$ varies proportionally with the total runtime, while for task parallel applications, $t_{react}$ varies with the average execution time of a task on one process.

### Little compute time lost

In *data parallel applications*, all threads continue to work while waiting for an adaptation request to be executed, so no overhead is added during this time, except for a few milliseconds’ interruption while setting up network connections to joining threads.

In *task parallel applications*, a smaller part of this interval is additional overhead: When receiving an adaptation request, all processes finish their current task and then execute an extra OpenMP join. As processes execute tasks at individual paces, they wait different lengths of time at the join. The average of all threads’ waiting times is the pre-adaptation cost, as this is the time threads would have been busy in the absence of an adaptation. Table 6.3 shows the lowest, highest and average pre-adaptation costs measured at 5 different adaptation points, using the same 8-thread-runs as in Table 6.2. These costs are smaller than the intervals shown in Table 6.2, as the latter include the time where processes are still finishing a task.
### Table 6.4: Average costs for one join from 7 to 8 threads, one leave from 8 to 7 threads, one move (simultaneous join and leave) from 8 to 8 threads, and the average thereof, using 8 machines.

<table>
<thead>
<tr>
<th>Application</th>
<th>100 Mbps network time (seconds)</th>
<th>1 Gbps network time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Join</td>
<td>Leave</td>
</tr>
<tr>
<td>Barnes</td>
<td>0.63</td>
<td>0.26</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>0.94</td>
<td>0.87</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.68</td>
<td>1.72</td>
</tr>
<tr>
<td>MGS</td>
<td>1.14</td>
<td>1.27</td>
</tr>
<tr>
<td>NBF</td>
<td>0.16</td>
<td>0.93</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.41</td>
<td>0.22</td>
</tr>
<tr>
<td>SOR</td>
<td>2.15</td>
<td>2.18</td>
</tr>
<tr>
<td>TSP</td>
<td>0.04</td>
<td>0.07</td>
</tr>
<tr>
<td>Water</td>
<td>-0.03</td>
<td>0.00</td>
</tr>
</tbody>
</table>

6.4 Selected adaptation costs

We present overall results for a selection of typical adapt events in this section and analyse the costs in detail in Chapter 7.

We ran the test scenarios of item B of Section 5.7.3 and measured the adaptation costs with method B (Section 5.5.3). Except for the pre-adaptation costs of task parallel applications presented in the previous section, these costs comprise the overall runtime increase caused by the adaptations.

Having results from 5 different adaptation points per application, we calculated the average adaptation cost for each type of adaptation (join, leave, move), as well as the overall average, and present the results in Table 6.4. As shown in detail in Section 7.5.1, the costs of a given adaptation vary little throughout an application’s execution: Variations are mostly within a ±10% range except for those applications with small absolute values (Barnes, TSP) or other irregularities (Quicksort), or MGS, where adaptations become progressively cheaper towards the end.

Table 6.4 shows that such joins and leaves cost around 1-2 seconds for most of the example applications, using the 100Mbps network, and well below 1 second on the 1Gbps network. A move even costs much less. The costs are also much lower in applications which use little shared memory overall (Barnes, TSP and Water, cf. Table 5.2). As the results for Water demonstrate, adaptation costs may even be negative. This happens when the application actually runs faster due to the adaptation, as compared to not adapting (cf. Section 5.5.3). In the case of Water, some of the adaptations shown in Table 6.4 increase the total number of diff transfers (due to the...
6.5 LENGTH OF ADAPTATION EFFECTS

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of adaptation points or tasks</th>
<th>100 Mbps network time (seconds)</th>
<th>1 Gbps network time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>2</td>
<td>8.50</td>
<td>6.29</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>2 to 5</td>
<td>1.20</td>
<td>0.53</td>
</tr>
<tr>
<td>Jacobi</td>
<td>2</td>
<td>1.93</td>
<td>0.76</td>
</tr>
<tr>
<td>MGS</td>
<td>1</td>
<td>1.36</td>
<td>0.50</td>
</tr>
<tr>
<td>NBF</td>
<td>1 or 2</td>
<td>6.98</td>
<td>5.11</td>
</tr>
<tr>
<td>Quicksort</td>
<td>Few</td>
<td>&lt; 1</td>
<td>&lt; 1</td>
</tr>
<tr>
<td>SOR</td>
<td>1 or 2</td>
<td>2.34</td>
<td>0.82</td>
</tr>
<tr>
<td>TSP</td>
<td>Few</td>
<td>≲ 1</td>
<td>≲ 1</td>
</tr>
<tr>
<td>Water</td>
<td>1</td>
<td>1.38</td>
<td>1.26</td>
</tr>
</tbody>
</table>

Table 6.5: Number of adaptation points and corresponding interval after an adaptation during which this event affects an on-going computation, for adaptations between 8 and 7 threads. In Quicksort and TSP, the interval is not precisely quantifiable, but it is below 1 second.

garbage collection when adapting) but decrease the total number of page transfers, resulting in less data transfers overall and a reduction in overall runtime.

Validation of results

The results presented in Section 6.2 provide experimental evidence supporting the results of this section: We periodically caused adapt events to occur for a large range of periods. Our measurements show that even for the smallest periods tested, less than 5 seconds, the average cost per adapt event is constant within a few percent of the costs reported in Table 6.4.

6.5 Length of adaptation effects

Table 6.5 shows for how long an adaptation affects an on-going computation. During this interval, denoted $i_{\text{adaptation-effect}}$, an application experiences some slowdown compared to a non-adaptive run of the same number of threads as after the adaptation. Table 6.4 shows the cost of these adaptations, i.e. how much of the intervals of Table 6.5 are costs.

The goal of Table 6.5 is twofold: Besides giving an idea of the length of $i_{\text{adaptation-effect}}$ for the selected applications and adaptations, it assists in answering the question of whether an adaptation $A_2$ following $t$ seconds after an adaptation $A_1$ may have the potential of reducing $A_1$’s adaptation cost. Such may be possible
if the data redistribution caused by $A_1$ is not completed when $A_2$ is performed, so the two adaptations’ data redistributions may be partially merged. We examine this question in the next subsection.

We used the same experiments as in the previous section (test cases of item B of Section 5.7.3) and calculated the results using method B (Section 5.5.3). The results for all adaptation types and all 5 measured adaptation points only varied within a 10-15% range per application, except for Water and MGS, therefore we only present overall averages in Table 6.5. In Water, the adaptation costs are always so small (cf. Table 6.4) that we cannot determine precisely whether observed runtime variations are due to adaptations or only measurement uncertainties. In MGS, the interval affected by an adaptation declines linearly as the computation progresses, from about double the values shown in the table to zero.

### Data parallel applications

The main result is that an adaptation only diminishes an application’s progress for a very short period after the event, namely for about 1-2 seconds for all the data-parallel applications tested except for NBF and Barnes, whose relevant intervals are 5-8 seconds. Secondly, we observe that in many cases the adaptation cost is realized in the first AP-interval after an adaptation, and in most other cases 2 AP-intervals are affected. The exception is 3D-FFT, with a maximum of 5.

Adaptations are only performed at adaptation points. *Successive* adaptations can only occur at successive adaptation points (successive adaptations are those that do not occur simultaneously, as opposed to batched adapt events). Therefore, any adaptation request $A_{2 \text{request}}$ that arrives only just after a previous adaptation $A_1$ (and is therefore not batched with $A_1$) can only be performed at the next adaptation point, no matter how soon after $A_1$ it arrives. Inasmuch as $A_1$’s adaptation cost is realized in the first AP-interval after $A_1$, the second adaptation cannot reduce the first adaptation’s cost.

All (MGS, Water) or at least 90% (Barnes, Jacobi, SOR) of an adaptation’s cost are realized in the first AP-interval after the adaptation. The exceptions are 3D-FFT and NBF. However, in 3D-FFT half the adaptation’s cost always occurs within the first AP-interval of $i_{\text{adaptation-effect}}$, so this cost cannot be reduced by any following adaptation, and often most of the other half of the cost occurs already in the second AP-interval. If a successive adaptation request arrives within about 0.6 resp. 0.3 seconds (100Mbps resp. 1Gbps network) of $A_1$, it may potentially reduce the cost from $A_1$ by up to half. We have not studied such cases in more detail. In NBF, whenever $i_{\text{adaptation-effect}}$ comprises 2 AP-intervals, the first one is very short (around 0.2 seconds for joins or 0.7 seconds for leaves), followed by a much longer second AP-
interval, giving a total of around 7 resp. 5 seconds (100Mbps resp. 1Gbps network). Therefore, any following adaptation request must be arrive and be ready for execution within the short time frame of the first AP-interval, otherwise this adaptation cannot be performed before the second AP-interval has completed. We have not tested such cases in more detail.

From the above we conclude that successive adaptations have no significant effect on the average cost per adaptation, either positive or negative, except possibly for the rare cases in 3D-FFT and NBF, as described. All or most of an adaptation’s cost occurs nearly always in the first AP-interval after the adaptation, significantly lengthening its duration, compared to non-adaptive execution.

**Task parallel applications**

In task parallel applications, the $i_{adaptation-effect}$ interval cannot be measured precisely due to the nature of such applications: The inherent randomness of task-to-thread allocations means data access patterns are independent of process identifiers. Different runs have quite different task-to-thread allocations, even for identical start parameters, so measuring adaptation effects via comparison of two runs yields quite varied results. Costs measured in such a way may be both positive or negative, i.e. an adaptation may even cause the application to finish somewhat sooner. This is especially true for TSP, where the runtime-effect of adaptations is always small.

Our measurements of tasks’ lengths and page transfers after the adaptation show no significant differences between the adaptive and non-adaptive case. Due to the variations in task size often exceeding the adaptation effects, the latter cannot be clearly identified. In any case, both the adaptation costs and the perceivable length of $i_{adaptation-effect}$ amount to only a fraction of a second for the adaptations in Quicksort and TSP shown in Table 6.5. We therefore observe that in task parallel applications successive adaptations also have no significant effect on the average cost per adaptation. The analysis in the next chapter will confirm these observations.

**Summary**

In this section, we have observed that successive adaptations in the data parallel applications we have examined do not affect the average cost per adaptation $c_{avg}$, even if the interval $i$ between the corresponding adapt requests is very small, such as one second. If $i$ is much less than 1 second, $c_{avg}$ may exceptionally be reduced a little in 3D-FFT and NBF. For task parallel applications, our measurements equally indicate that $c_{avg}$ is independent of $i$ for both Quicksort and TSP.

Note that any possible reduction in $c_{avg}$ by any following adaptations can never
include the cost component of moving pages valid solely on leaving threads off such threads.

Note further that adapt events of close proximity are often batched and executed in one adaptation, which greatly reduces the average cost per adaptation (Section 7.5.4).

In Section 7.6, we analyse the observations of this section more closely from the point of view of the data redistributions, providing explanations for our observations.

6.6 Summary

This chapter provides an overview of the costs typically associated with adaptations. We have tested the system using nine diverse applications. Our results have shown adaptation costs of on the order of 1-2 seconds when using the 100Mbps network and 0.5 seconds when using the 1Gbps network for most of the programs, and only fractions of a second for all the other programs. The system reacts quickly when receiving adaptation requests, in most of the programs evaluated adaptations are performed within a split-second after the request is issued, and in the other cases within a few seconds at most. Further, the system swiftly returns back to a “normal” state after an adaptation: At worst only a few seconds after an adaptation no more adaptation effects are noticeable in the computation, i.e. work proceeds thereafter as if there had been no adaptation. Finally, even frequent adaptations hardly increase the overall runtime: An increase of 10% for an adaptation every 10 seconds is typical, and correspondingly less for longer inter-adaptation intervals.
Chapter 7

Analysis of adaptation costs

In this chapter we analyse in detail how adaptation costs arise and how various factors contribute to and influence the costs. We first identify the main cost components, thereafter we analyse the two decisive components, the data movement by the system and by the application, individually. The main part of the chapter then deals in detail with the most important factors influencing the data movement by the application, thereby providing us with criteria for the judgment of what circumstances lead to cheaper or more expensive adaptations.

All results presented in this chapter were obtained using the 100Mbps network, unless otherwise stated.

7.1 Cost components

We measured the time for various actions performed during and after an adaptation and which together account for the adaptation cost. We identified five main elements into which the adaptation cost can be broken down, $t_{\text{pre-adaptation}}$, $t_{\text{gc-adaptation}}$, $t_{\text{system}}$, $t_{\text{fork-adaptive}}$, and $t_{\text{appl}}$, and we present results for four representative applications in Figures 7.1, 7.2, 7.3 and 7.4. These figures compare identical adaptations for various problem set sizes for each application, using the test cases described in item H of Section 5.7.3. Although we performed similar measurements for numerous others of the scenarios outlined in Section 5.7.3, a variation of the problem set sizes while keeping the other parameters constant provides the largest relative variation of each cost component’s share of the overall costs and thus illustrates best the importance of each such element, so we only present these results.

The main adaptation cost components are the following:

---

1 Various limits imposed by the system did not currently permit tests for the 4x reference size case to be executed for all applications. Further, a join could not be performed for the $\frac{1}{16}$ size Quicksort-case due to the short runtime of about one second for this input.
Figure 7.1: Breakdown of adaptation costs of Jacobi for various problem set sizes.

Figure 7.2: Breakdown of adaptation costs of 3D-FFT for various problem set sizes.

Figure 7.3: Breakdown of adaptation costs of NBF for various problem set sizes.
Figure 7.4: Breakdown of adaptation costs of Quicksort for various problem set sizes. Negative cost components reduce the overall adaptation cost. The pre-adaptation costs $t_{\text{pre-adaptation}}$ are not included in this graph. They depend on the task granularity, not the problem set size.

$t_{\text{pre-adaptation}}$ (task parallel applications only). This is the pre-adaptation cost or the compute time lost by task parallel applications, as threads are waiting at the inserted OpenMP join for the last thread to finish its task and arrive. This cost depends exclusively on the task granularity and scheduling used by the application and cannot be influenced by the system, so we do not further analyse it. When task lengths do not vary too much, $t_{\text{pre-adaptation}}$ is about half the compute-time of one task.

$t_{\text{gc-adaptation}}$. This is the time needed for the garbage collection at the adaptation.

$t_{\text{system}}$. This is the time spent by the system to move any pages $p_{\text{system}}$ valid only on one or more leaving threads to some other threads.

$t_{\text{fork-adaptive}}$. This cost covers all other with respect to the non-adaptive case extra actions performed within the adaptation’s $\text{Tmk_{fork}}$, apart from $t_{\text{gc-adaptation}}$ and $t_{\text{system}}$. In particular, $t_{\text{fork-adaptive}}$ comprises the assembling of adaptation information and of new page information (for the joining threads in case of joins or for all threads in case of leaves), the reassignment of bookkeeping data to have a contiguous thread identifier space after an adaptation, the discarding and clearing of all unused memory consistency data after the garbage collection, and three extra barriers, including the sending of all new page information.

$t_{\text{appl}}$. This component contains all adaptation overhead incurred after completion of the $\text{Tmk_{fork}}$ where the actual adaptation is executed, i.e. the sum of any extra
delays occurring during the further course of computation, in particular extra page \( p_{appl} \) transfers and page ownership requests for the data redistribution, and, where applicable, extra diff transfers. These delays are obviously scattered in-between all other communication (which would also have occurred in the absence of an adaptation), so they are difficult to distinguish from the latter, but for Sections 6.5 and 7.6 we analyse each individual OpenMP fork-join’s contribution to \( t_{appl} \) after an adaptation to determine more precisely when this cost occurs.

The results shown in Figures 7.1, 7.2, 7.3 and 7.4 are typical results, representative of all the applications tested, but individual cases may obviously vary somewhat.

The \( t_{fork-adaptive} \) costs are insignificant in all cases we tested, generally varying between 0-40ms for most runs using the default problem set sizes, or up to 70ms for large inputs using up to 256MB of shared memory: Rearranging some data structures and discarding memory consistency information is cheap. The three extra barriers likewise do not cost much, as there are practically no arrival delays, as all processes have just synchronized at the preceding OpenMP join. When large problem sets are used, the messages containing the rearranged consistency information may become large and, if the data exceeds the MTU, multiple messages are sent, impacting the \( t_{fork-adaptive} \) a little. Still, such multiple sends are executed immediately one after the other without needing any acknowledgement replies.

The garbage collection times \( t_{gc-adaptation} \) are just as small as the \( t_{fork-adaptive} \) times for all tested applications except for Barnes, ranging from 0-30ms for most runs using the default problem set sizes, and increasing to a maximum of 70ms for large inputs using up to 256MB of shared memory. Only Barnes has a significant garbage collection time of around 0.75 seconds at most adaptations. However, only a small part of this time is actually an adaptation overhead, because Barnes always performs frequent garbage collections. Either the adaptation occurs when a regular garbage collection would have been due anyway, adding no cost, or otherwise the adaptation acts like a “premature” garbage collection. In that case, our results show that the added cost of an earlier-than-necessary garbage collection are for the largest part made up for with a faster execution thereafter, so for Barnes, overall adaptation costs are actually smaller than the \( t_{gc-adaptation} \) component.

In summary, \( t_{fork-adaptive} \) and \( t_{gc-adaptation} \) together account for less than 10% of the total cost of any adaptation costing at least half a second, except for Barnes. In addition, this percentage decreases rapidly for more expensive adaptations. In the 1Gbps environment, both the absolute times and the share of the cost in percentage are even less for both of these two elements. As these two components are so small, we concentrate our analysis in the rest of this chapter on the other two elements, \( t_{system} \).
Table 7.1: 3D-FFT: Number of pages and cost in seconds of pages moved off leaving processes by the system, for \( \frac{1}{4} \) of all processes leaving (left) and one process leaving (right). The identifiers of the leaving processes are irrelevant for these costs (100Mbps network, 4K page size).

and \( t_{\text{appl}} \), which are essentially the costs for the data movement by the system and the data movement by the application. Numerous other test cases not shown here, including 32-thread-runs and a batching of up to 8 adapt events in one adaptation confirm our conclusions.

### 7.2 Data movement by the system

In this section we analyse \( t_{\text{system}} \), the time used by the system to move pages not valid on any continuing threads off any leaving threads.

Specifically, the system performs the following steps: Within the \texttt{Tmk.fork} executing the adaptation, after having completed the garbage collection, the system determines \( p_{\text{system}} \), all pages of which only leaving threads have valid copies. Next, it allocates these pages in a round-robin fashion among either all the continuing processes or, if an equal number of processes join and leave, all the joining processes. In-between the first and the second barrier after the garbage collection, still within \texttt{Tmk.fork}, all processes in parallel fetch their allocated \( p_{\text{system}} \) pages and assume ownership.

As the data movement of the \( p_{\text{system}} \) pages is performed between barriers and before any application work is resumed, its cost is equal for any type of application for a given amount of \( p_{\text{system}} \) pages allocated per process. The issues determining the cost are therefore the system’s allocation strategy, the number of \( p_{\text{system}} \) pages sent in a single message, and as far as the application is concerned, the sharing patterns which determine what share of pages are exclusively valid on one process. In contrast, the costs for the page movement by the application \( p_{\text{appl}} \) vary widely for different types of applications, even for equal \( p_{\text{appl}} \) values, as shown in detail in Sections 7.3-7.5.
Data movement varies by adaptation

Table 7.1 shows how the $t_{\text{system}}$ cost varies for different numbers of processes, for single leaves and for batched leaves of one quarter of all processes. Our measurements show that in the optimal case, a process fetches close to 1000 $p_{\text{system}}$ pages per second. For an adaptation from 4 to 3 processes with one leave, we achieve a rate of close to 3000 $p_{\text{system}}$ pages per second transferred ($\frac{1760}{0.60}$), as each of the three continuing processes is allocated one third of the pages. This rate is practically constant for all of the applications tested and for any of the processes leaving.

Figures 7.8 and 7.9 show schematically for a block data layout how the leaving process’ data partition is reallocated in a round-robin manner to continuing processes. For simplicity, the example assumes that all pages on all processes are exclusive. The left half of Table 7.1 shows how the cost is roughly halved for a doubling of the number of processes, when the share of processes leaving is the same (except for the 32-24 process-adaptation: The costs for the leaves from 32 processes cannot be compared directly to the other cases, as in this case only, two processes share one physical network link). As Figures 7.8 and 7.9 illustrate, the reason is that the number $n$ of processes fetching pages from the same leaving process is unchanged at three, so the bottleneck is the number of pages moved onto one continuing process, or the roundtrip-time $t_{\text{pages-req}}$ of a pages-request without contention. In contrast, in the one-leave case in the right half of the table, this number $n$ increases from 3 to 7, 15 or 31, as the number of processes before the adaptation is increased to 8, 16 or 32. Now the bottleneck is increasingly on the leaving process, which faces contention trying to service requests from many different processes simultaneously, dramatically increasing $t_{\text{pages-req}}$. In addition, allocating smaller chunks to a larger number of processes means more messages are not filled up to their MTU capacity with $p_{\text{system}}$ pages anymore, i.e. the average number of pages per message decreases.

These results clearly show that much room for improved $p_{\text{system}}$ pages-allocation strategies exist. We have not implemented these yet, as the need is not urgent, as the overall costs are small in all cases.

Data movement varies by application

Applications with read and write accesses to all of the data in shared memory and with no or little false sharing often have nearly all pages exclusive on one process, with each of $n$ processes having $\frac{1}{n}$ of all pages exclusive. Each process works on its partition, and when processes exchange data and such pages are written to again on the new processes, they become exclusive there. Examples of such applications are SOR, Jacobi, NBF and Quicksort. Table 7.2 shows that for one leave from 8 processes
### Table 7.2: Average number of pages $p_{system}$ moved off the leaving process for one leave from 8 to 7 threads, and $p_{system}$ as a percentage of shared memory pages allocated by the application.

<table>
<thead>
<tr>
<th>Application</th>
<th>$p_{system}$</th>
<th>% of allocated shared memory pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>707</td>
<td>6.6%</td>
</tr>
<tr>
<td>MGS</td>
<td>576</td>
<td>4.7%</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1522</td>
<td>12.4%</td>
</tr>
<tr>
<td>NBF</td>
<td>1564</td>
<td>11.7%</td>
</tr>
<tr>
<td>Quicksort</td>
<td>1045</td>
<td>10.7%</td>
</tr>
<tr>
<td>SOR</td>
<td>1872</td>
<td>12.5%</td>
</tr>
<tr>
<td>Water</td>
<td>34</td>
<td>8.8%</td>
</tr>
<tr>
<td>TSP</td>
<td>14</td>
<td>13.0%</td>
</tr>
</tbody>
</table>

often close to $\frac{1}{8}$ or 12.5% of all pages are moved off the leaving process. Whether the application is of the regular, irregular or independent type is irrelevant, only the share of work done and thus the writes by the leaving process(es) matter.

Applications where the master of $n$ processes has far more than a $\frac{1}{n}$ share of pages exclusive due to the slaves each reading and writing less than such a $\frac{1}{n}$ share obviously need to move correspondingly less pages off leaving processes. 3D-FFT and Water belong to this category, Table 7.2 shows that their percentage of $p_{system}$ pages is less. MGS is similar, although here the percentage becomes progressively less as the computation advances, as pages are read by another process after their final write, causing them to be shared, i.e. valid, on several processes, and the share of data used in the computation becomes progressively less closer to the end.

Applications with either a significant amount of the data accessed read-only or with much false sharing have many pages in the shared state, so with only one leave at least one continuing process also has a valid copy of such pages. Even with several leaves, in most cases the valid copies of such pages do not all reside on the leaving processes, so for these pages no $p_{system}$ transfers are needed. Barnes is an example of an application with extensive false sharing and no or few exclusive pages held by the slave processes, therefore a leave often does not cause any $p_{system}$ page transfers, as shown in Table 7.2.
7.3 Data movement by applications

In this section we analyse the data redistribution occurring during the course of computation after an adaptation, while in Section 7.4 we investigate in more detail where and how this data movement causes bottlenecks in the network traffic which are decisive for the cost.

In data parallel applications the data distribution is determined by the process identifiers and the total number of processes. As processes get new identifiers and/or their number changes at an adaptation, data partitions of processes are shifted or reassigned to different processes. The next subsection describes the algorithm used to reassign process identifiers at an adaptation, while the subsection thereafter outlines the data movements that occur due to these reassignments.

In task parallel applications the data distribution is independent of process identifiers, but adaptations still cause some extra data movement. We analyse the reasons for this data redistribution in the third subsection below.

The data reallocation is defined at the time of adaptation by the new constellation of processes, but the actual data transfers only occur when the application experiences page access faults and fetches the missing pages or diffs. When these requests are generated is therefore completely application-dependent, and they happen interspersed with other requests that are not performed due to the adaptation.

The cost of all extra data movement by applications is captured in the $t_{appl}$ cost component, while the total number of extra page transfers generated by application access misses is expressed by $p_{appl}$.

7.3.1 Process identifier reassignments

For data parallel applications, whose data layout is determined by process identifiers and the number of processes, the strategy chosen for the assignment of process identifiers to processes at an adaptation usually has a large impact on the subsequent data movement by applications and the associated cost.

In task parallel applications the choice of strategy should be irrelevant, as data accesses are performed independently of process identifiers.

In many cases, applications rely on a contiguous numbering of identifiers from 0 to $n-1$ for $n$ processes. The system guarantees this property and rearranges the identifiers according to the following three strategies:

1. In adaptations with no change in the number of processes, the new processes receive the identifiers of the leaving processes.

2. In adaptations with a decrease in the number of processes, any new processes get the smallest identifiers of the leaving processes. If thereafter any of the
other not-yet-assigned leaving processes’ identifiers are smaller than any identifier of a continuing process, all the latter processes’ identifiers are reassigned with smaller identifiers such that (a) the ordering is maintained, and (b) the process id space is contiguous again.

(3) In adaptations with an increase in the number of processes, the best-matching data partitions before and after the adaptation are calculated and chosen, assuming a regular block data distribution with equal-sized blocks. More precisely, the new partitions are numbered 0 to \( n - 1 \), and for each non-leaving process \( x \) the new partition \( y \) with the largest overlap of \( x \)'s partition is determined, and \( x \) gets the new identifier associated with \( y \). The joining processes then receive all remaining identifiers in the id-space 0 to \( n - 1 \).

Although this algorithm is optimized for block data layouts, it also often works well in practice for irregular layouts. For block-cyclic and independent layouts, the choice of strategy is quite irrelevant, as shown in the sections below.

### 7.3.2 Data parallel applications

**Join events.** Adding new compute processes causes the following data movements: (1) The joining processes experience cold misses and fault-in pages, as all their pages are invalid initially, and (2) data is redistributed among the “old” processes, when the total number of processes changes (the number of joining and leaving processes differs).

In many applications, all shared memory pages are accessed repeatedly many times. In such cases, joining processes generally have to page in the complete partition assigned to them, typically \( \frac{1}{n} \) of all pages for \( n \) processes. The transfers are less only if not all of the partition’s data is accessed anymore during the rest of the computation. Other processes generally page in less data extra, as a process’ pre-adaptation data
Figure 7.6: Block-cyclic data distribution: Schematic example of a join from 3 to 4 threads, for two different process identifier assignment strategies. Blocks of the same shading are allocated to the same thread. The figure illustrates how the overall amount of data redistributed often remains the same, regardless of how process identifiers are assigned.

partition often overlaps somewhat with its post-adaptation data partition.

Figures 7.5 and 7.6 are a schematic view of a join from 3 to 4 processes for a block resp. block-cyclic data layout. The shading in the first figure and the arrows in the second figure illustrate which parts of the partitions resp. which blocks are reallocated at the adaptation. Such a reallocation corresponds to the maximum theoretical extra data movement, which occurs when all such data is still accessed after the adaptation and the same page transfers do not also occur in the absence of the adaptation. In the left half of these figures, we show the effect our process identifier strategy has (cf. Section 7.3.1), in contrast to a “naïve” strategy of adding new threads at the end of the id-space, depicted on the right.

Consider the shared memory data space as one large contiguously-addressable block of memory \( sm \), with each process being allocated one resp. several smaller contiguously-addressable blocks of memory \( sm_i \) in the block resp. block-cyclic data layout, such that all \( sm_i \) blocks together comprise \( sm \).

In a block layout, the closer the address-ranges of a pre-adaptation process \( x_{old} \)’s partition \( sm_i \) and a joining process’ partition \( sm_j \) are, the more data is moved off such a process \( x_{old} \). If several processes join simultaneously, then the one whose partition-address is closest, i.e. has the most overlap, is relevant. Figure 7.5 shows how our strategy of interspersing the new processes evenly among the continuing processes reduces the overall amount of data moved and in most cases reduces the maximum amount of data moved off one process with respect to the “naïve” strategy mentioned above. With our strategy, a reallocation of around 30% of the data is typical for one or a few joins, or up to 50% when the number of processes is doubled.

However, with a block-cyclic layout both strategies result in equal or similar amounts of overall or per-process data movement. In addition, the share of data moved is usually much higher than with a block layout, it may approach \( \frac{n-1}{n} \) for a
7.3. DATA MOVEMENT BY APPLICATIONS

![Image of data movement calculations](image)

**Figure 7.7**: Example showing the *extra* page transfers between every two threads caused by one join-adaptation from 7 to 8 threads, for Jacobi (top left), MGS (top right), Barnes (bottom left) and Quicksort (bottom right). The system assigns identifier number 4 to the new thread, incrementing the previous identifiers 4 to 6 by one, so as to minimize the data redistribution.

join of $n - 1 \rightarrow n$ processes. A given block is in most cases allocated to a different process at an adaptation, and generally the more so the larger the overall number of processes.

The effect of an *irregular layout* is similar to a block-cyclic layout: When each process has several smaller data blocks of various sizes and/or scattered unevenly throughout the data space and this layout depends on the number of processes, then an adaptation typically allocates most of the data to some other process(es).

Figure 7.7 shows the extra data movements between every two processes caused by one example join of 7 to 8 processes. The system assigns the new thread the id 4 in the middle of the id-space. The measurements confirm precisely our analysis for regular layouts, showing a block data distribution (Jacobi, top left), and a block-cyclic data distribution (MGS, top right). An irregular data layout (Barnes, bottom left) and an independent data layout (Quicksort, bottom right) also let a joining process page-in more pages and/or far less pages are moved off such a process, compared to the other processes. Barnes is not typical of our description above of irregular layouts due to a very high degree of sharing such that most pages are often valid on most
CHAPTER 7. ANALYSIS OF ADAPTATION COSTS

Figure 7.8: Block data distribution: Schematic best-case data redistribution example of a leave of one quarter of 8 or 4 threads. The shading shows data partitions reallocated to a new thread.

processes, i.e. most of the data space is not partitioned (partitioning means that shared memory pages are primarily accessed by a single process). A join therefore does not cause many extra page-ins among the “old” processes, and conversely the joining process fetches many pages from all other processes. We discuss Quicksort’s independent layout below in Section 7.3.3.

**Leave events.** A leave of one or more processes generates the following data movements: (1) All pages $p_{system}$ valid only on leaving processes are moved to continuing processes, and (2) data is redistributed among the continuing processes, when the total number of processes changes, causing $p_{app}$ page fetches. The $p_{app}$ pages often include some of the $p_{system}$ pages, as the system allocates these without knowledge of any data partitioning.

We have already seen in Section 7.2 how the $p_{system}$ pages are fetched from the leaving process(es).

Thereafter, the $p_{app}$ data movement by the application occurs. In principle, the transfers are the inverse of those for a corresponding join adaptation, except that some or even most of the $p_{system}$ pages are already moved onto the correct processes, eliminating the $p_{app}$ transfers for those pages, so overall less $p_{app}$ fetches occur for leaves than for corresponding joins.
7.3. DATA MOVEMENT BY APPLICATIONS

Figure 7.9: Block data distribution: Schematic worst-case data redistribution example of a leave of one quarter of 8 or 4 threads. The shading shows data partitions reallocated to a new thread.

Figures 7.8 and 7.9 provide schematic views of various processes leaving in an application with a block data layout. The gray shading illustrates the maximum theoretical adaptation-induced data movement, including the moving of pages off leaving processes. This maximum corresponds to all pages on the leaving process(es) being exclusively-valid and all pages being accessed extra after the adaptation with respect to the non-adaptive case. Assuming this scenario, the first figure depicts a best-case choice of leaving processes, while the second figure shows a worst-case choice, as far as the amount of data moved is concerned.

In a block layout, analogously to joins, using the \( sm_i \) notation introduced there, a continuing process fetches more pages extra the closer the address-ranges of its partition \( sm_i \) and a leaving process’ partition \( sm_j \) are. If several processes leave simultaneously, then the one whose partition-address is closest, i.e. has the most overlap, is relevant. However, leaves cannot benefit from an id-reassignment strategy the way joins do: The more leaving processes are next to each other and at the edge (in the id-space), the more data is moved. When middle ids leave, the percentage of data moved may be as low as 30%, but more unfortunate scenarios may easily cause 80% or more of the data space to be shifted.

In a block-cyclic layout, the data movement overall is very similar to joins, i.e. generally most of the data is reallocated. The number of \( p_{\text{system}} \) pages placed optimally
varies considerably for different processes leaving, reducing the number of \( p_{\text{appl}} \) page fetches accordingly. If the \( p_{\text{system}} \) pages had no influence on the number of \( p_{\text{appl}} \) transfers, the latter number would be more or less equal for any process leaving, in contrast to applications with a block data layout.

The effect of an \textit{irregular layout} is also very similar for leaves as for joins, i.e. typically a high percentage of data is reallocated.

### 7.3.3 Task parallel applications

Independent applications do not assign data partitions, so an adaptation primarily only changes the percentage of data accessed per process, without predestining a specific layout. The amount of data movement depends on issues such as the number of different tasks contained by one page and whether most of the data is accessed read-only or read and write.

**Join events.** As in data parallel applications, new processes page in much data. When most pages are exclusively-valid, around \( \frac{1}{n} \) of all used pages are in this state on each one of \( n \) processes, so joining processes fetch extra this share of pages. When many pages are simultaneously valid on several processes (usually due to read-only sharing), processes each have a larger percentage of all used pages valid, so new processes page-in a correspondingly larger share.

The network traffic among the “old” processes however is generally slightly lower after a join than without any adaptation, in contrast to data parallel applications: As the number of processes increases, each process performs a smaller percentage of all work, so the share of valid pages per process as a percentage of all pages decreases. Before this reduction is completed, each process on average needs to fetch somewhat fewer pages than in the non-adaptive case.

**Leave events.** At a leave, the \( p_{\text{system}} \) pages are first distributed among the other processes, exactly as for data parallel applications, as described in Section 7.2.

The \( p_{\text{system}} \) pages increase the share of valid pages on the remaining processes such that each of these processes has on average the \textit{same} share of valid pages as in the corresponding non-adaptive case. Therefore, the \( p_{\text{appl}} \) traffic is on average around zero after leave-adaptations, so the cost of leaves is mainly determined by the cost of the \( p_{\text{system}} \) page fetches.
7.4 Key bottlenecks

In this section, we analyse the network traffic caused by adaptations and identify bottlenecks which determine the adaptation costs.

We first provide background information about network traffic issues, then we study if and how bottlenecks may occur for different types of applications. Next, we analyse analytically the bottlenecks we expect to find. Finally, we describe what measurements we made to identify the bottlenecks and we provide empirical results confirming our theories.

7.4.1 Background

This subsection describes the mechanisms of the network traffic involved when moving shared memory data.

The data repartitioning at an adaptation leads to read-faults and/or write-faults, depending on whether read or write accesses are desired.

In most of the applications studied, most processes only ever access a small portion of all shared memory pages allocated by the application (unless the program is running with very few processes such as four or less), so most pages remain unmapped (empty) on most processes. In the absence of false sharing, a page uses the single-writer protocol, so a write by a process $x$ to a page previously owned by another process $y$ later causes this page to be both invalidated and unmapped on $y$.

A read-fault by a process $dst$ on an unmapped page $p$ first of all generates a page request: $Dst$ sends an appropriate message to some process $src$ with a valid page copy of $p$, generating an asynchronous I/O signal on $src$. $Src$ immediately services the request by sending back a message with a copy of $p$, unless $src$ is already busy servicing another asynchronous request. The request is necessarily blocking on process $dst$, as $dst$ cannot proceed with its computation until the data arrives, but it still services any asynchronous requests it receives during this time.

A read-fault on an invalid but mapped page lets the system skip the above step. When or once a page is mapped, a read-fault generates diff-requests, if applicable. Diffs are created when false sharing occurs, and the page’s protocol switches to multiple-writer. Diff requests generate request and reply messages just as page requests do, but on average the cost per diff is only a fraction of the cost of a page request, for various reasons: (1) Diffs are generally much smaller than pages, so much less data is sent per diff, resulting in shorter roundtrip-times for single diffs, as Table 5.1 confirms. These measurements include the diff-creation by the sender and the diff-application by the receiver, but these operations cost little compared to the network time. (2a) Processes may need diffs from several processes when updating one page. In such
cases, all requests are sent first, and then all replies are awaited, so there is only one two-message-roundtrip-latency cost, and (2b) all diffs requested from the same process are batched in one message, as much as the MTU allows.

A write-fault by a process dst first generates the above actions for a read-fault, if the page is unmapped and/or invalid. A valid page means a read does not generate any access miss. When or once the page is valid, the system sends an ownership request which again requires a request and a reply message, as for page requests, except that not much data needs to be sent this time.

In summary, a page reallocated to another process at an adaptation often (in most of the programs we studied) leads to four extra messages, as processes often write to their new partition, causing both a page- and an ownership-request. As these requests are handled sequentially, a reallocation of data that is only read after the adaptation is significantly cheaper, typically requiring only a page-request.

7.4.2 Data parallel versus task parallel computation

In regular data parallel applications, each process typically performs the same amount of work and generates more or less the same amount of network traffic within one OpenMP parallel section, so all processes arrive at OpenMP joins nearly simultaneously, the work is well-balanced. An adaptation causing a wide discrepancy in the extra network traffic generated per process means that the process $x$ with the most extra such traffic becomes a bottleneck, as the other processes now arrive earlier at OpenMP joins and are idle until $x$ finally arrives.

In irregular applications, work is often less well-balanced, so the above observation is only partly valid. We study the implications in more detail in Section 7.5.7.

Task parallel applications have a different behavior: As processes compute tasks at their own pace without global synchronization in-between, a larger number of page fetches by one process due to an adaptation, such as a join paging-in data, only affects the requesting and the replying process, other processes are not involved. As there are no OpenMP joins, no idle time can occur there after the adaptation, as opposed to data parallel applications.

7.4.3 Analytical results

We introduce $c_{\text{max\_adapt}}$, the maximum adaptation-induced overhead incurred by any one process once the computation is resumed after an adaptation. This overhead is the sum of all extra page, diff and ownership request costs plus the cost of all replies to such requests coming from other processes. Replying to a page request costs only a fraction of a page request, it is the time spent servicing the request, but it does
not include any network-transfer time, whereas the request contains the reply-time plus the roundtrip-time of two messages. For example, for one page the cost-ratio is about $\frac{1}{6}$ (130\,\text{us} / 800\,\text{us}, 100Mbps network). The $c_{\text{max,adapt}}$ overhead may even be negative, as in some cases less pages and messages are transferred in the adaptive case, compared to the non-adaptive case. With the help of next section’s measurements, we can approximately determine $c_{\text{max,adapt}}$ and verify the current section’s analysis in the ensuing sections.

In regular applications (block or block-cyclic data layout) with a well-balanced computation, we have seen in Section 7.4.2 that the process with $c_{\text{max,adapt}}$ constitutes the bottleneck and thus determines $t_{\text{appi}}$.

In a block layout, we have already seen in Section 7.3.2 how the number of pages fetched and sent by one process varies widely for different processes, so we expect the adaptation-induced overhead $c_{\text{adapt}}$ to vary accordingly and the processes with lower $c_{\text{adapt}}$ values to be idle for some time.

As we have seen in Sections 7.3.2 and 7.3.3, with the process identifier strategy we use, joining processes usually fetch more data than any other process, regardless of the number of joining processes, but they do not receive page requests from other processes. In contrast, processes whose partition is next to a new process’ partition not only fetch many pages, they also send many pages to other processes. We therefore find $c_{\text{max,adapt}}$ either on new processes or on such neighbor processes.

For leaves, $c_{\text{max,adapt}}$ varies much depending on which process leaves. As Figures 7.8 and 7.9 illustrate, with only one leave, all processes at most need to page-in only a part of their new partition, but with two or more leaves, in unfavorable scenarios such as two “edge” processes leaving, some processes may need to page in all of their new partition just as joining processes do, and in addition all of their pre-adaptation partition may be requested by other processes. In such cases, we expect $c_{\text{max,adapt}}$ to be somewhat higher for leaves than for corresponding joins.

In a block-cyclic layout, all processes generally page in a similar share of data, as opposed to applications with block data layouts. Figure 7.6 shows how typically most of the blocks are reallocated at adaptations, so not only joining but also continuing processes theoretically page in most or all of their new partition. Another difference also confirmed by Figure 7.7 are the all-to-all data movements, as opposed to transfers mainly among neighboring processes for block data partitioning. Every non-joining process receives page requests from every other process, causing many of these requests to be delayed a little, as a process can only service one page request at a time. Still, the maximum page movements for any process are similar for similar adaptations in both data layouts, so we expect similar costs, even though the percentage of data moved is usually far higher in the block-cyclic layout.
In an irregular layout, if the computation is not well-balanced the \( c_{\text{max\_adapt}} \) overhead is not necessarily on the bottleneck process, so we cannot predict the adaptation cost as straightforwardly as for balanced regular computations. When \( c_{\text{max\_adapt}} \) is on a process which in the non-adaptive case would be idle for some time, this adaptation overhead is partly overshadowed by another process' work and thereby reduces the cost.

In an independent layout, there are no global bottlenecks because each process works at its own speed, so high \( c_{\text{adapt}} \) overheads do not slow down other processes, so we again expect comparatively lower adaptation costs than in regular applications.

### 7.4.4 Measurements

This section describes what measurements we performed to identify bottlenecks.

In our tests we used a fully-switched full-duplex network and let each process run on a different machine with its own physical network link (except for tests with more than 16 processes), so network contention is only possible where several processes simultaneously send a message to the same destination process.

Every extra page request caused by an adaptation adds some overhead on the requesting and on the responding process. We measured \( p_{\text{dst}} \), the total number of pages requested by any one process, \( p_{\text{src}} \), the total number of such requests replied to by any one process, and \( p_{\text{dst\_src}} \), the total of \( p_{\text{dst}} \) and \( p_{\text{src}} \) on the same process. We performed such measurements for each individual OpenMP fork-join parallel section separately, starting at the adaptation and continuing for a sufficient number of fork-joins, until all adaptation effects were contained in the measurements. For each of these values, we then determined the maxima \( p_{\text{max\_dst}} \), \( p_{\text{max\_src}} \) and \( p_{\text{max\_dst\_src}} \) over all processes, and for each type of measurement we summed up the maxima from each of the relevant join-fork intervals, giving us the cumulative maxima \( p_{\text{cum\_max\_dst}} \), \( p_{\text{cum\_max\_src}} \) and \( p_{\text{cum\_max\_dst\_src}} \). Each such cumulative maximum may be the sum of maxima from different processes for different join-fork intervals: As all threads synchronize globally and must wait for each other at every join, any bottleneck on a thread \( x \) in interval \( A \) has no effect on the bottleneck in the next interval \( B \), so we determine the maxima separately for every interval.

### 7.4.5 Empirical results

Figure 7.10 shows how the \( p_{\text{cum\_max\_dst}} \) and the \( t_{\text{appl}} \) values correlate for the same test cases as those used for Table 6.4, i.e. adaptations performed at five different moments for each of the nine applications. Analogously, Figure 7.11 shows the correlation between \( p_{\text{cum\_max\_dst\_src}} \) and \( t_{\text{appl}} \). All the per-process \( p_{\text{dst}} \) and \( p_{\text{src}} \) or the overall \( p_{\text{appl}} \)
Figure 7.10: Correlation of $p_{cum\_max\_dst}$, the maximum number of extra application page requests issued by one process, and $t_{appl}$, the adaptation cost (application part), at different points during each application’s execution.

Figure 7.11: Correlation of $p_{cum\_max\_dst\_src}$, the maximum number of extra application page requests issued plus page sends handled by one process, and $t_{appl}$, the adaptation cost (application part), at different points during each application’s execution.
transfers occur once the computation resumes after an adaptation, so the overall cost they add is captured exactly by the \( t_{\text{appl}} \) cost component.

Figure 7.12 provides another view, showing the \( t_{\text{appl}} \) cost in relation to \( p_{\text{appl}} \), i.e. it shows how the average cost per redistributed page varies for the various cases.

Water and TSP use little shared memory overall, so all values are close to zero and therefore too small for a meaningful analysis, given an inherent uncertainty of measurement, so we concentrate on the other applications below.

**Block data layout.** The results for the 3 regular codes 3D-FFT, Jacobi and SOR are all close to a diagonal trendline in Figure 7.10. Further, the leaves are slightly more expensive per \( p_{\text{cum\_max\_dst}} \)-page fetch than the joins (on imaginary lines perpendicular to the dotted diagonal, the leaves are somewhat further in the upper left half of the graph). Here, the processes whose \( p_{\text{cum\_max\_dst}} \) values are shown also handle a similar amount of \( p_{\text{src}} \) page replies not included in this figure, in addition to the \( p_{\text{dst}} \) fetches, in contrast to the joins. This is evident from Figure 7.11, where the \( p_{\text{src}} \) replies are also counted. As each \( p_{\text{src}} \) reply typically costs only \( \frac{1}{6} \) of a \( p_{\text{dst}} \) request, it adds little to the cost, so leaves appear somewhat above the trendline in Figure 7.10 but much below the trendline in Figure 7.11. These observations confirm precisely our analytical results from the previous sections. There we have seen how for joins the \( p_{\text{cum\_max\_dst}} \) values are found on the new processes and how for leaves the processes having the \( p_{\text{cum\_max\_dst}} \) values also handle many \( p_{\text{src}} \) replies (Figures 7.5, 7.7, 7.8, and 7.9).
Figure 7.12 shows how the $p_{appl}$ data redistribution is more expensive for joins than for leaves. Figure 7.10 explains this observation: The $p_{cum\_max\_dst}$ values are much higher for joins, so the data movement is less balanced among the processes than for leaves, as the total $p_{appl}$ values are very similar. This observation matches our analysis, which has shown how joins often page in significantly more data than any other processes.

**Block-cyclic data layout.** In MGS, the $p_{cum\_max\_dst}$ fetches in Figure 7.10 are significantly more expensive than in all other programs. Recall that MGS is the only application we tested where an adaptation causes all-to-all data movements. Every non-joining process sends $p_{src}$ replies for requests coming from every other process, causing contention on the responding process, which can only handle the requests sequentially. These extra delays increase the average cost per $p_{cum\_max\_dst}$ and $p_{cum\_max\_dst\_src}$ page request. In Figure 7.11, MGS’ values for joins, which also include many $p_{src}$ replies (which in the absence of contention would be cheap and would lower the average), therefore seem nearly as expensive as the other applications’ values for joins, which count none or few $p_{src}$ replies. The effect of the contention is therefore clearly visible in both figures.

However, when regarding the average per-page data redistribution cost in Figure 7.12, MGS fares best among the regular applications, because the data movement is much better balanced. In MGS, all processes handle a very similar amount of page redistribution work, whereas a block layout causes very wide discrepancies among the processes, as seen in previous sections. Therefore, MGS’ generally higher percentage of redistributed data is partially compensated for by a more efficient redistribution, so overall costs are similar among regular applications.

**Irregular data layout.** The results for Barnes are inconclusive in Figures 7.10, 7.11 and 7.12. These figures concentrate only on page transfers, but in Barnes the $diff$ transfers have a decisive effect on adaptation costs. In our tests, the difference in the number of $diff$-transfers between the adaptive and the non-adaptive case was often on the order of 20000 with both positive and negative values, i.e. an adaptation can either increase or decrease the $diff$-traffic, depending on when the adaptation occurs: Barnes performs frequent garbage collections (in 8-process runs after every 3rd iteration), so an adaptation, which always includes a garbage collection, also has the effect of a premature garbage collection whenever the adaptation occurs before a regular garbage collection is due. As the frequency of garbage collections remains the same after the adaptation, an adaptation sometimes increases by one the total number of garbage collections performed, increasing the overall $diff$-traffic. In other
cases this number remains equal despite a premature \( g_{\text{adapt}} \) garbage collection at the adaptation, so the overall diff-traffic decreases due to less diff-transfers at or before the early \( g_{\text{adapt}} \), compared to a regular garbage collection.

However, the number or frequency of garbage collections only has a small overall influence on performance, because a garbage collection speeds up work thereafter, as pages which otherwise would be invalid are updated during garbage collection instead of when actually needed, or potentially long diff histories are discarded, so page updates cause much less transfers thereafter. This compensation lets adaptations in Barnes cost less than the cost of the adaptation’s garbage collection alone, even though extra page transfers add to the cost. Joins in Barnes are much more expensive than leaves due to the high percentage of valid pages per process, as we have seen in Section 7.3.2: A join pages in its partition, in this case about 750 pages as reflected by the \( p_{\text{cum,max.dst}} \) value, but for leaves this value is only around 200, and the \( t_{\text{system}} \) cost for leaves is often negligible.

In NBF, an average \( p_{\text{cum,max.dst}} \)-page fetch is only a fraction as expensive as in the regular codes. In this case, there is much overlapping of adaptation-induced communication with computation, as discussed in more detail in Section 7.5.7, so the overhead of most of the extra transfers does not add any extra cost, greatly reducing NBF’s costs as shown in Figures 7.10, 7.11 and 7.12.

Another factor lowering adaptation costs in NBF is the read-only use of most pages after an adaptation, so many pages in exclusive state on one process \( x \) before an adaptation change to the shared state on two processes \( x \) and \( y \), as \( y \) pages in part of a new partition while \( x \) may keep its copy valid. Such a read-fault by \( y \) requires only one message roundtrip, so it is much cheaper than a write-fault which needs two message roundtrips, if the page is empty before (cf. Section 7.4.1). In most of the other programs studied, processes write to their new partitions after adaptations, generating write-faults.

To sum up, the data redistribution of the \( p_{\text{appl}} \) pages is comparatively very cheap in NBF. As a consequence, leaves are much more expensive than joins in NBF, because the \( p_{\text{system}} \) page transfer costs are similar in all applications, they cannot in any way gain from the factors reducing the \( p_{\text{appl}} \) transfer costs described above. As processes initially write to their data partition, the share of \( p_{\text{system}} \) pages is about \( \frac{1}{n} \) for \( n \) processes.

Independent data layout. In Quicksort, we observe large variations between more than +1000 and less than -1000 pages for \( p_{\text{cum,max.dst}} \), with no clear correlation to the \( t_{\text{appl}} \) times, as applications with no global synchronization do not have such bottlenecks. The total number of extra transfers is much more decisive, although the
average cost is significantly less per transfer, so the absolute costs are small. These are the results we expected in Sections 7.4.2 and 7.4.3.

Summary

In conclusion, for regular applications the bottleneck value determining the adaptation cost is the sum of per-interval maxima of per-process adaptation overheads, over all join-fork intervals affected by the adaptation. These maxima are essentially the largest $p_{dst} \cdot f_{req} + p_{src} \cdot f_{rep}$ value of any process for that join-fork interval. In a block layout, we set $f_{req} = 1$ while $f_{rep}$ is about $\frac{1}{6}$ and reflects how much cheaper a page request is for the replying versus the requesting process. In a block-cyclic layout, both factors are higher due to contention when servicing page requests, but the factors vary with the number of processes. However, these maximum adaptation overheads are similar for block and block-cyclic layouts, so overall adaptation costs are also similar.

With irregular data layouts, much adaptation-induced data movement may be overlapped by computation, often greatly reducing adaptation costs in comparison to regular data layouts. Independent data layouts also generate much lower adaptation costs, as one process’ adaptation overhead does not slow down another process due to their independent computation.

7.5 Factors determining the adaptation cost

After seeing in the previous sections how the adaptation costs originate and where the key bottlenecks are, in this section we analyse in detail how variations of the most important parameters influence these bottlenecks and the costs.

We study the importance of each of the following parameters:

- Timing of adaptation request
- Identifier of joining or leaving thread
- Number of threads
- Batching of adapt events
- Interval between adapt events
- Problem set size
- Type of application and data layout
- Data sharing patterns
Figure 7.13: Variation of adaptation costs during runtime of applications.

- False sharing
- Execution platform

### 7.5.1 Timing of adaptation request

We use the tests of Section 5.7.3, item B, to assess how adaptation costs may vary throughout the execution of a program.

Figure 7.13 shows that for most of the applications tested adaptation costs stay more or less constant during the course of computation. Only MGS and to a much lesser degree Barnes and Quicksort have evident variations. The reason is that most applications repeatedly access the same data numerous times throughout their work, and for the data redistribution there is no difference whether an application is close to the end and only needs a given page once more, or whether it still needs to access the page hundreds of times.

Only in MGS adaptation costs decrease linearly as the computation progresses, as in every iteration $i$ only the sub-matrix below right of the corresponding pivot-element is needed, so the amount of data needed shrinks constantly in proportion to the number of iterations left.

Generally, joins may be cheaper close to the start or end of a computation. At the beginning, many cold misses have not occurred yet, so whether a process is paging in its partition due to its joining the computation or due to initial cold misses makes no difference, and when finishing, a joining process may not need all data of its partition.
Figure 7.14: Variation of adaptation costs for one of 8 processes leaving, depending on which process leaves, and corresponding variation of overall extra page fetches (by system and application).

7.5.2 Identifier of joining or leaving thread

We use the tests of Section 5.7.3, item E, to assess the influence of the thread identifiers of joining or leaving threads on the adaptation costs: At a given adaptation point, we measured the cost of a leave alternately for each slave and present the results in Figure 7.14. For leaves the system also moves pages $p_{system}$ off the leaving thread(s), resulting in two page movements for some pages instead of only one for joins. However, when for any page included both in the $p_{system}$ and the $p_{appl}$ page transfers we consider only the source of that $p_{system}$ transfer and the destination of that $p_{appl}$ transfer, then
we observe that the overall data movements for joins and leaves are similar, except that the data is transferred in opposite directions. In other words, more or less the same page transfers occur, but for each such transfer, the source and destination threads are exchanged. For this reason, we show results for leaves only instead of for leaves and joins, but the conclusions below are similarly valid for joins. In addition, we let the system automatically assign approximately optimal identifier(s) to joining thread(s), so a user need not deal with this issue.

The $p_{system}$ page transfers and their cost are included in Figure 7.14. However, if they had been omitted, the appearance of the curves in the figure would hardly change. The curves would only be shifted down vertically, because, for a given application, the overhead of the $p_{system}$ data movement is nearly constant for all the cases shown in the figure. This observation matches the statements made in Section 7.2.

In 3D-FFT and Jacobi, adaptation costs are higher the closer to the edge of the id-space a leaving thread’s id is. Not only the total extra page transfers vary similarly, but also the maximum extra adaptation overhead per process, which is largely proportional to $p_{cum \_max \_dst}$. Figure 7.15 shows a close correlation of $p_{cum \_max \_dst}$ and $t_{appl}$. The $p_{system}$ transfers and their cost $t_{system}$, included in Figure 7.14, only add a near-constant vertical offset to each of the series in Figure 7.14. 3D-FFT and Jacobi have a block data distribution, so these results confirm our analysis in Section 7.3.2.
which shows how both the overall and the bottleneck transfers become less for such applications the more evenly the leaving threads are distributed in the id-space, i.e. for one leaving thread, the more in the middle it is.

In NBF, the total cost in seconds varies little, although the total extra page transfers vary as in 3D-FFT and Jacobi. The reason is a significant overlap of adaptation overhead with computation, made possible by the unbalanced computation of the irregular NBF code. This effect is detailed in Section 7.5.7. Suffice it to mention here that the overall and per-page costs of the $p_{cum \_max \_dst}$ transfers in NBF are only a fraction of what they are for 3D-FFT and Jacobi, as seen in Figure 7.15: The curve for NBF is close to the zero-cost dotted line, and this curve has a much smaller variation between minimum and maximum cost in seconds than the curves for 3D-FFT and Jacobi.

In MGS, both the overall values in Figure 7.14 and the bottleneck values in Figure 7.15 are practically the same for any thread leaving. MGS has a block-cyclic data layout, and in Section 7.3.2 we have seen how these values are largely independent of the leaving or joining threads’ ids for such applications.

In Quicksort, the variations between the individual threads are relatively small and to a large degree “coincidental”, as its data layout and therefore also adaptation costs are largely independent of thread identifiers. In independent applications there is no relevant bottleneck thread (cf. Section 7.4.2), but every page fetch adds a similar overall overhead to the computation: In Figure 7.14 we observe that the adaptation cost series and the page transfer series follow each other closely for this application. The negative $p_{cum \_max \_dst}$ values in Figure 7.15 simply mean that the “peak” transfer values, the maximum page fetches by one thread, become less due to the adaptation. As these maxima become less, transfers on other threads decline similarly, leading to negative $p_{appl}$ overall extra application page transfers and negative $t_{appl}$ times.

### 7.5.3 Number of threads

We use the tests of Section 5.7.3, item F, to assess how for different numbers of processes the cost of one join or one leave event and the cost of the same share of joining or leaving threads varies. We omit the results for 32 processes, as they (1) do not provide any other conclusions than those from the 4, 8 and 16-process tests, and (2) cannot be directly compared to the results for up to 16 processes: In our experimental platform, we have 16 physical network links, so with 32 processes each link is used by two processes, leading to network contention, but for up to 16 processes we allocate one link to one process. Despite this limitation, our 32-process results are similar to the 16-process results.

Figure 7.16 shows how the adaptation cost decreases as the number of processes
Figure 7.16: Overall adaptation costs (except pre-adaptation costs in Quicksort) for various numbers of processes, for 3D-FFT, Jacobi, NBF and Quicksort (top to bottom).
Figure 7.17: Total number of extra page transfers (by system and applications) for various numbers of processes, for 3D-FFT, Jacobi, NBF and Quicksort (top to bottom).
Figure 7.18: Correlation of $p_{cum_{max, dst}}$, the maximum number of extra application page requests issued by one process, and $t_{appl}$, the adaptation cost (application part), for 3D-FFT, Jacobi and NBF (top to bottom). The data point labels indicate the number of processes (after joins resp. before leaves).
increases. In many cases, especially in regular applications such as 3D-FFT and Jacobi, the cost of one adapt event is halved as the number of processes is doubled. Furthermore, the batched adaptations of two or four joins or leaves are often only a few percent more expensive than single adapt events, and on occasion even cheaper, illustrating the great reduction of the cost per event achieved by batching, as described further in Section 7.5.4.

Figure 7.17 shows the overall data redistribution, the total number of extra system and application page transfers $p_{\text{system}} + p_{\text{app}}$ for each of the adaptations. For joins, this data movement is practically constant for an equal share of joined processes, and for one join it even decreases slightly for more processes. Leaves have larger variations between single and batched events, and the total data movement is generally substantially higher, in some cases even by a factor of three, such that actually more pages are moved than are in use, as some pages are moved twice. These observations are explained by the data redistribution patterns analysed in Section 7.3: The system assigns approximately optimal ids to the joining processes, interspersing them evenly in the pre-adaptation id space. When the ids of leaving processes are similarly optimal, as depicted in the “Minimum” cases, the number of pages redistributed by the application $p_{\text{app}}$ is actually nearly always less than the number in the corresponding join cases. However, the extra $p_{\text{system}}$ transfers by the system increase the total data movement to the values shown in Figure 7.17. Most of the $p_{\text{system}}$ pages are in such cases already put on the correct destination process, so the $p_{\text{app}}$ is smaller than for joins. Any other choice of leaving threads causes far more data to be moved. Figures 7.8, 7.9 and 7.17 show for a block data layout (1) how the total data movement increases by multiples between best and worst case, and (2) how for an equal percentage of leaving processes the data movement increases as the number of processes increases, for any other than an “optimal” choice of leaving processes: With more processes, each process’ data partition is smaller and is more often shifted over to another process at adaptation.

Figure 7.18 shows for the same adaptations how the $p_{\text{cum}}_{\text{max}}_{\text{dst}}$ maximum extra per-process page fetches and the $t_{\text{app}}$ time for the data redistribution by the application correlate closely. We do not include Quicksort in the latter figure, as independent applications do not have one bottleneck process. The labels at the data points show the number of processes after the joins or before the leaves (to avoid clutter, not all series are labeled): The costs decrease the more processes are used. Leaves are always somewhat more expensive per $p_{\text{cum}}_{\text{max}}_{\text{dst}}$ transfer due to additional $p_{\text{src}}$ page sends not included in the figure, as explained in Sections 7.4.3 and 7.4.5.

The $p_{\text{cum}}_{\text{max}}_{\text{dst}}$ transfers, not the total data movement, are largely relevant for the adaptation costs: For joins, this bottleneck occurs on the new processes, so as every
doubling of the number of processes means a halving of each process’ data partition, the \( p_{\text{cum\_max\_dst}} \) transfers and the costs also decrease accordingly. For leaves, the equivalent bottleneck and costs are often very similarly halved for every doubling of the number of processes. Only the \( t_{\text{system}} \) cost to move pages off leaving processes currently does not improve as much with an increase of processes, as analysed in Section 7.2, therefore the overall costs for leaves do not always decrease as much as for joins, as the number of processes grows. In fact, in NBF the overwhelming part of the adaptation cost with 16 processes is due to \( t_{\text{system}} \). In Figure 7.18, \( t_{\text{appl}} \) is very small, but in Figure 7.16 the overall adaptation cost for the same adaptations is not so small, and our measurements confirm that the difference is approximately equal to \( t_{\text{system}} \).

Actually, for corresponding join and leave adaptations (comparing joins from \( a \) to \( b \) processes with leaves from \( b \) to \( a \) processes for the same applications), both the \( p_{\text{cum\_max\_dst}} \) values and the \( t_{\text{appl}} \) times are nearly always worse for joins than even for worst-case leaves. However, the fact that the overall adaptation costs for such leaves is nevertheless often worse than for the joins is due to the additional \( t_{\text{system}} \) cost of moving pages off leaving processes. These pages are frequently moved a second time, increasing \( p_{\text{appl}} \), as the system cannot know how to optimally place them, in the application’s view.

Whereas in the regular applications the bottleneck transfers and the \( t_{\text{appl}} \) costs are roughly halved as the number of processes is doubled, in NBF the \( t_{\text{appl}} \) costs but not the bottleneck transfers actually improve much more as more processes are used: In most 16- and 8-thread-cases in Figure 7.18 \( t_{\text{appl}} \) is nearly zero. This result is due to a larger adaptation-overhead-to-computation-overlap (Section 7.5.7) for more processes, so although the adaptation overhead is more or less halved as the number of processes is doubled, the increase in the overlap means the cost is more than halved. We identify 2 reasons for this phenomenon: (1) Doubling the number of processes does not quite double the speedup and the idle times at OpenMP joins are not quite halved, but the extra transfers per process and their cost are roughly halved, and (2) with more processes, the maximum number of extra transfers in fewer cases occurs on the process which in the non-adaptive case arrives last at the OpenMP join concerned. Therefore, a larger share of the extra transfers can take place while other processes are still active with work.

Quicksort has significant variations due to its independent data layout type, but as the absolute costs are always much smaller than in most other applications tested, the large variations-to-cost ratio precludes a clear analysis of the costs. Nevertheless, even though the percentages of data moved are smaller in Quicksort than in other programs, they vary similarly in proportion. Larger numbers of processes mean less
7.5. FACTORS DETERMINING THE ADAPTATION COST

In summary, the costs for one or for several batched joins are often nearly equal and are more or less halved for every doubling of the number of processes. For leaves, the improvements in cost for more processes are less dramatic only because the $t_{\text{system}}$ cost does not scale down accordingly in our current implementation. With leaves, although total data movement values vary much, bottleneck values and therefore costs vary less: Extra page transfers in worst-case leaves occur much more in parallel than in best-case leaves, so the cost-difference is less.

7.5.4 Batching of adapt events

We use the tests of Section 5.7.3, item G, to assess the effect of batching of events on adaptation costs. We batched up to 4 joins, 4 leaves, or 4 moves in one adaptation, where a move is actually a simultaneous join and leave event.

Figure 7.19 shows the overall adaptation costs for such adaptations, as well as...
98 CHAPTER 7. ANALYSIS OF ADAPTATION COSTS

Figure 7.20: 3D-FFT: Correlation of $p\text{cum\_max\_dst}$, the maximum number of extra application page requests issued by one process, and $t\text{appl}$, the adaptation cost (application part).

Figure 7.21: Jacobi: Correlation of $p\text{cum\_max\_dst}$, the maximum number of extra application page requests issued by one process, and $t\text{appl}$, the adaptation cost (application part).
7.5. FACTORS DETERMINING THE ADAPTATION COST

the average cost per join, leave or move as the overall cost divided by the number of simultaneous such events. Figures 7.20, 7.21 and 7.22 show how the $p_{\text{cum\_max\_dst}}$ and the $t_{\text{appl}}$ values correlate for 3D-FFT, Jacobi and NBF - we omit the results for Quicksort, as these are (1) often negative, and (2) vary widely, and in addition, the correlation in independent applications is not evident (cf. Section 7.4.5). These last 3 figures give insight to the results in Figure 7.19.

A batching of join events is very attractive: In all cases the overall costs for 3 or 4 joins are only a few percent higher than for one join. The reason is that the bottleneck determining the cost is located on the joining process(es) which need to page in all (or most) of their data partition. Whether one or several processes join therefore hardly changes the bottleneck, the correlation figures show that the $p_{\text{cum\_max\_dst}}$ values are very similar in these cases. Only the processes replying to the page requests experience more contention with more joins, therefore the costs do rise a little with additional joins.

A batching of leave events is also advantageous, although less so: A doubling of overall adaptation costs for 4 versus 1 leave is typical. First of all, the $p_{\text{system}}$ number of pages valid only on leaving process(es) is proportional to the number of leaving processes. The associated cost $t_{\text{system}}$ is only little higher for 2 versus 1 process leaving, but for 4 leaves it rises substantially due to the currently-implemented allocation strategy for the $p_{\text{system}}$ pages, as discussed in Section 7.2. The $p_{\text{appl}}$ extra page transfers by the application present similar scenarios: With one leave out of 8 processes, in applications with a block data layout and about a $\frac{1}{n}$ share of all pages
valid per process, one process fetches at most about \( \frac{1}{8} \) of all data, as this process still has a slight overlap of the pre-adaptation and post-adaptation data partitions. With 4 leaves and only 4 processes remaining, a process at most has to page in its complete partition of about \( \frac{1}{4} \) of all data. In any type of application, with less processes remaining, each process on average pages in a larger share of the data. The correlation figures show clearly how the \( p_{\text{cum\_max\_dst}} \) values rise for more leaves, and the \( t_{\text{appi}} \) cost increases proportionally. NBF profits from the overlap of adaptation overhead with computation, as described in Section 7.5.7, therefore NBF’s results do not all correlate as well as the other applications’.

Obviously, comparisons are also somewhat a matter of perspective: Here, we performed joins from 4→8 and leaves from 8→4 processes. If we would however compare adaptations with an equal number of processes after the adaptation, we would perform an adaptation from 12→8 processes, and leaves would seem much cheaper - on the other hand, we could compare adaptations with an equal number of processes before the adaptation - then leaves would seem even more expensive in relation to joins.

A batching of move events is quite similar to a batching of leave events, except for single move events: Whenever the system detects a move adaptation (an equal number of leave and join events), it tries to optimize the adaptation by transferring the \( p_{\text{system}} \) pages valid only on the leaving process(es) directly to the joining process(es), instead of distributing them among the continuing processes as usual. This optimization works very well for one move in data parallel applications, as the new process gets the leaving process’ data partition and the \( p_{\text{system}} \) pages contain a significant part of this partition. However, in the current implementation, which may be optimized later, when several processes leave, the system does not know which new process should get which of the leaving process’ \( p_{\text{system}} \) pages and therefore allocates many of these pages to the wrong process, so they arepaged in again. The correlation figures reflect these scenarios, showing how in the one-move case the \( p_{\text{cum\_max\_dst}} \) values are very small. Considering that each move event is actually a combination of 2 events (leave and join), move events are especially cheap.

### 7.5.5 Interval between adapt events

In theory, an adaptation \( A \)’s cost may potentially be reduced by an ensuing adaptation \( B \), given a sufficiently short interval \( A - B \), namely when the data redistribution provoked by \( A \) is not completed yet when \( B \) is executed and \( B \)’s data redistribution is merged with the outstanding part of \( A \)’s data redistribution such that overall less data is moved. Such a scenario is especially likely for reciprocal adaptations. Consider a join followed by a leave. Thereafter, the number of processes and consequently the
overall data allocation is restored to before the join (although usually not the per-process data allocation). The leave may partially undo the join’s data redistribution. However, we have seen that most of a join adaptation’s cost is usually caused by the new process itself, so the chance of B reducing A’s cost is very limited at best for such a case.

In Sections 6.5 and 7.6 we see how short intervals \( i \) typically are during which data is redistributed and why ensuing adaptations often cannot reduce the costs of any previous adaptations in data parallel applications, regardless of \( i \) (except of course when a batching of events occurs). In practice, in none of all the data and task parallel applications we evaluated did our tests show any significant dependence between the cost per adaptation and the length of the interval between adapt events. Among others, our experiments from Section 6.2, where we performed periodic adaptations at various frequencies, confirm our results.

### 7.5.6 Problem set size

We use the tests of Section 5.7.3, item H, to assess the scalability of the system for more or less shared memory usage. Figure 7.23 shows on a logarithmic scale how the overall adaptation cost generally increases linearly with an increase of the problem set size, as expected. Only for small sizes leaves may be somewhat disproportionally more expensive because the moving of pages off leaving processes is less efficient, as too few pages are transferred to fully take advantage of the capacity to batch many pages in one message. Figure 7.24 shows how the total number of extra page transfers caused by adaptations also scales linearly with different problem set sizes. Small sizes may in
Figure 7.24: Total number of extra page transfers (by system and applications) scales with problem set size.

Figure 7.25: Correlation of $p_{cum_{max_{dst}}}$, the maximum number of extra application page requests issued by one process, and $t_{appl}$, the adaptation cost (application part).
7.5. FACTORS DETERMINING THE ADAPTATION COST

7.5.7 Type of application and data layout

In this section we compare adaptation costs for regular, irregular and independent applications.

We have already studied in detail the implications of block and block-cyclic data layouts in Sections 7.4.3 and 7.4.5, among others, and we have concluded that adaptation costs are similar (for similar problem set sizes and adaptations).

When applications with an irregular data layout nevertheless have a balanced computation as regular applications do (cf. Section 7.4.2), then the data movement and adaptation costs may be similar to programs with a block-cyclic data layout (cf. Section 7.3.2).

In the rest of this subsection, we study the implications of a less well-balanced computation for the adaptation costs. We use the tests of Section 5.7.3, item I, to assess how the adaptation overhead may overlap with computation, thereby reducing the overall adaptation cost. Figures 7.26, 7.27 and 7.28 show results for one join event and one leave event, for three codes with different data sharing patterns. To reveal the effect of these adaptations, each of these figures also shows results for a non-adaptive comparison case. We measured the time each process spends waiting

Figure 7.26: Network traffic in first fork after adaptation point for Jacobi, sorted by number of pages. The data point labels represent process identifiers.

some cases lead to slightly disproportionally more page transfers, as the page size used by the system remains the same (4K), so the sharing becomes more coarse-grain with respect to the overall use of shared memory. Figure 7.25 shows how the $p_{cum_{max_{dst}}}$ page transfers and the $t_{appl}$ costs also correlate quite linearly for the different problem set sizes.
Figure 7.27: Network traffic in first fork after adaptation point for 3D-FFT, sorted by number of pages. The data point labels represent process identifiers.

Figure 7.28: Network traffic in first fork after adaptation point for NBF, sorted by number of pages. The data point labels represent process identifiers.
at the first OpenMP join following the adaptation or at the same join in the non-adaptive case, and the number of pages fetched by each process during the preceding AP-interval. The labels in the figures represent the process identifiers. For the leave, we chose the “end” process (with id 8 of 0-8), whereas the system inserted the joining process in the “middle” as id 4 of 0-7 (The choice of processes for the leaves/joins is irrelevant for our conclusions).

Without any adaptation, the figure for Jacobi shows a very well-balanced computation (for this parallel section), as all processes arrive at the barrier practically simultaneously. Also in 3D-FFT, little compute time is lost by the global synchronization, as all processes arrive well within 0.1 seconds of each other, but in NBF, the irregularity of the computation lets processes wait up to 2.9 seconds (in this case), while other processes are still working. The communication is however well-balanced for all three programs, as each process fetches a nearly equal number of pages.

With adaptations, the data redistribution causes different amounts of pages to be moved among the processes. We observe a close connection between the extra page fetches and the extra time needed for this, as each request adds a similar overhead: The more pages a process fetches, the more time it needs or the less time it spends waiting idly at the next join. In all cases, the process fetching the most pages arrives last and therefore determines the overall adaptation cost. However, while in Jacobi and in 3D-FFT only the adaptive cases exhibit significant delays between the first and the last process’ arrivals, in NBF this delay is high in the adaptive and the non-adaptive case, varying between about 2.9 and 3.1 seconds in Figure 7.28. As the overall adaptation cost is the difference between the adaptive and the corresponding non-adaptive case, the $t_{appd}$ component of the adaptation cost is often very small for NBF, in line with the results shown in Tables 7.3 and 7.4. The reason is that in NBF by far most of the extra page fetches are performed by processes which would otherwise have significant idle time to spare, resulting in a large overlap of adaptation overhead with (other processes’) computation. In this case processes 4 and 7 wait about 2 seconds in the non-adaptive case, while they arrive last when adapting (We can however not simply compare adaptive and non-adaptive results for each process, as results vary widely between individual test runs, even for identical test cases, due to the irregularity of the program, but the general conclusion remains valid).

All three figures further demonstrate that the bottleneck in terms of extra page transfers is far more pronounced in joins than in leaves, as we have seen in Section 7.4, as there is a large difference between the joining process with id 4 and the process with the next-most number of page fetches. The new process generally pages in its complete data partition, while in the case of leaves the data redistribution is somewhat more balanced.
While these figures show the advantage of irregular applications over regular applications, independent applications have a different behavior still. We have only studied task queue-type independent applications, although other types are conceivable. As task queue applications do not have (frequent) OpenMP joins, processes do not lose compute time waiting for other processes to complete the data redistribution, each redistributed page causes overhead only on the requesting and the replying process. In addition, the share of data moved is much smaller, so adaptations are much cheaper than in regular applications (cf. Section 7.3.3).

7.5.8 Data sharing patterns

The issue of which processes exchange data with each other (all-to-all sharing, nearest-neighbor sharing, master-to-all sharing or vice versa, etc.) is not very relevant for the adaptation cost. One essential issue we have studied in this chapter so far is the data layout and how it is rearranged after an adaptation - the data exchange patterns have little influence on the initial page-in of a new post-adaptation partition.

Important is further the question of read-only sharing of data, where two or more processes perform read-accesses to the same shared memory pages without any write-accesses in-between, so these pages assume the “shared” state on all those processes. As every page is valid on at least one process, when pages become shared the average share of valid pages per process for each one of $n$ processes exceeds $\frac{1}{n}$ of all used pages. As this fraction increases, joins become more expensive, but leaves may become cheaper: Joining processes normally page in this fraction of all pages, and we have seen that in most cases these processes’ page fetches determine the adaptation cost. Leave-only adaptations however benefit twofold from more read-only sharing: First, more sharing means fewer pages are exclusively-valid, so the number of $p_{\text{system}}$ pages moved off leaving processes decreases. Second, a larger share of valid pages on a process may cause a smaller increase of this share at the adaptation, so, if the data allocation is regular, fewer $p_{\text{appl}}$ pages are fetched: Consider e.g. 8 processes each sharing the same $\frac{1}{2}$ of all pages, while the other $\frac{1}{2}$ of the pages are exclusively-valid on some processes, so each process’ valid share is on average $\frac{1}{2} + \frac{1}{2} \times \frac{1}{8}$. In this scenario, a leave of one process increases each continuing process’ share to $\frac{1}{2} + \frac{1}{2} \times \frac{1}{2}$. Contrast this with a regular block data layout, where such an adaptation increases this share from $\frac{1}{8}$ to $\frac{1}{7}$. Obviously these effects are highly application-specific, other scenarios where more read-only sharing does not reduce leave-costs are also possible.

The common sharing modes are read-write sharing and write-write sharing: When a process $x$ writes to a page $p$ valid on other processes, $x$’s copy becomes the only valid copy once the other processes’ copies are invalidated, unless there is false sharing, which we analyse in the next section. The common case means pages are generally
valid only on one process at a time, and each one of \( n \) processes has on average a \( \frac{1}{n} \) share of valid pages. This is typical for most of the programs we studied, the notable exceptions are MGS and TSP (read-only sharing), and Barnes (false sharing).

MGS has significant read-only sharing, yet the simplified conclusions above for such a sharing pattern do not apply to MGS because much of the data is never accessed anymore after an adaptation, so page-ins do not happen. MGS works with progressively less data as its computation proceeds. In TSP, the small numbers and large variations between successive runs preclude clear observations confirming our analysis. However, Barnes supports our theories well, because false sharing and read-only sharing have very similar effects, as we see in the next section.

### 7.5.9 False sharing

When several processes write to different parts of the same shared memory pages within the same synchronization intervals, false sharing occurs and these pages switch to the multiple-writer protocol and the "shared" state. For adaptations, the effect is similar to the read-only sharing described in the previous section, in both cases pages are valid "shared" on several processes simultaneously, so joins become more expensive and leaves may become cheaper as more false sharing happens.

Barnes has false sharing of around half of all used pages among all or most processes, i.e. processes exchange data of the same pages. A joining process typically fetches many of these pages extra. Besides this high degree of false sharing and some read-only sharing, Barnes has little of the other sharing types (read-write or write-write) common to the other applications, so few (if any) pages are exclusively-valid on one process, so a leave of one (or a few) processes generates hardly any \( p_{\text{system}} \) page transfers. However, a leave in Barnes leads to a significant amount of \( p_{\text{appl}} \) transfers, as opposed to the previous section’s theory, because the latter analysis applies to regular data layouts, while Barnes has irregular access patterns and data distributions, which may cause much smaller overlaps in pre- and post-adaptation data partitions of a process (cf. Section 7.3.2). In such cases, although the partitions’ size may not increase much, many pages are still transferred. Tables 7.3 and 7.4 confirm this analysis, showing how the \( p_{\text{appl}} \) transfers for both one join and one leave amount to about 50% of all data being moved, much more than for the other applications, and the leave has no \( p_{\text{system}} \) transfers.

### 7.5.10 Execution platform

As we see in Sections 7.4.3 and 7.4.5, the adaptation cost is in most cases determined by the maximum extra overhead experienced by any process, and this overhead usu-
ally for the largest part consists of page requests and page ownership requests, and a smaller part is the servicing of requests from other processes. Even where this overhead and the cost do not clearly correlate, we nonetheless see in Section 7.1 that the $t_{appl}$ cost component accounts for most of the adaptation cost, and $t_{appl}$ is precisely caused by the extra actions performed in the adaptive case, namely such requests and request servicings. Further, our measurements show that the network time is the dominant cost factor in a two-message request roundtrip. The servicing of a page request on the replying side, which does not include any network transfers, costs around $\frac{1}{6}$ of the total request time.

From the above, we expect adaptation costs to vary more or less proportionally with (1) network latency times and (2) average end-to-end message transfer times. The latter may be significantly influenced by network contention due to non-ideal topologies, e.g. several processes or logical connections sharing one physical network connection. We expect environment factors which determine the time to assemble and to service a message to have a smaller but not negligible influence on adaptation costs.

For adaptations, the most important system-internal parameter which can be changed is the page size. The larger the page size, the less pages and messages are transferred for the same data. Moving from the 100Mbps network to the 1Gbps network, we increased the page size by a factor of 4. Due to the coarser page granularity, the number of extra page-requests and messages is reduced by somewhat less than 4, and the adaptation costs are typically improved by a factor of about 3. The latency is very similar in both networks, and the bandwidth difference allows for similar page transfer times of the different-size pages, so the reduction in transfers caused by the larger page size is truly the decisive factor.

### 7.5.11 Summary

In this section, we have analysed how the adaptation costs are determined by various factors. The main results are:

- Adaptation costs often vary little throughout a program’s execution
- Leave costs vary somewhat, depending on which thread is leaving, for applications with a regular block distribution. Programs with other data distributions show little such variations
- Adaptations are cheaper as the number of processes running is larger, or vice versa
• Batching several adapt events in one adaptation significantly reduces the cost per event

• The length of the interval between successive adaptations has no effect on the adaptation costs

• Adaptation costs usually increase in proportion to an increase of the computed problem set size

• Independent applications generally have the lowest adaptation costs, followed by irregular applications, while regular applications have the highest costs (given equal shared memory usage)

• Extensive read-only sharing or false sharing of data can both increase join costs

• The network latency is the primary environment factor determining adaptation costs

### 7.6 Length of adaptation effects

In Section 6.5 we have observed that an adaptation typically affects an on-going computation only for a very short interval of a few seconds at most - in this section we analyse these observations.

#### Data parallel applications

*Data parallel applications* often access all or most of the shared memory pages used by a given thread in every iteration. In such cases, the data redistribution caused by an adaptation is mostly or even completely performed within the first AP-interval after an adaptation. Our measurements confirm that the AP-intervals where the data redistribution occurs correspond completely to the intervals shown in Table 6.5 (except maybe for Barnes leave events, where diff transfers influence the results). In other words, the extra page transfers and the adaptation cost occur in exactly the same AP-intervals, confirming that this data movement is overwhelmingly responsible for the adaptation cost.

#### Task parallel applications

In *task parallel applications*, even successive runs with identical start parameters lead to highly different data access patterns due to the task queue model of tasks’ execution not being bound to specific threads. The lengths of adaptation-effects intervals are therefore not precisely quantifiable empirically. However, our measurements show that
a join-adaptation hardly affects the progress of the other, “old” processes, only the new process initially computes tasks somewhat slower. We further observe no visible slowdown of any process’ computation after a leave-adaptation. Rather, the cost \( t_{system} \) to move pages off leaving processes seems to be approximately equivalent to the overall adaptation cost \( t_{total} \), and in some cases \( t_{system} \) even exceeds \( t_{total} \), meaning that once the computation resumes, it actually progresses faster after the adaptation than in the non-adaptive case.

We use the results of Section 7.3.3 as a basis for our analysis. In task parallel applications each process proceeds at its own pace. After a join adaptation, there is no data redistribution among the “old” processes, so they continue without any slowdown. Only the new process(es) need to fetch extra pages, so new process(es) experience a slowdown until they have paged-in all these extra pages. We can estimate the length of this adaptation-effects interval \( i_{adaptation-effect} \) as follows: Consider applications where most pages are exclusively-valid, i.e. where each one of \( n \) processes has about \( \frac{1}{n} \) of all used pages in this state, and where each page is accessed on average \( m \) times throughout the execution. Then, after about \( \frac{1}{m} \) of the application’s total runtime, each page is on average accessed once, i.e. after roughly \( \frac{1}{m} \) of the application’s runtime a new process will have paged in its \( \frac{1}{n} \) share of pages. This interval should therefore roughly correspond to the interval during which a new process computes slower after joining than comparable non-joining processes. To be precise, considering that all first-time accesses on joining processes are cold misses, requiring a page transfer, while some of the accesses on the “old” processes will be to valid pages, the interval is probably a little longer. Also, much read-only sharing increases the interval length: If on average many more than \( \frac{1}{n} \) of all pages are valid on each process, a new process obviously has to page in many more pages to attain the same share of valid pages. Our measurements suggest \( i_{adaptation-effect} \) interval lengths of about \( \frac{1}{20} \) of the application’s runtime for TSP and about \( \frac{1}{3} \) for Quicksort.

After a leave adaptation, the share of valid pages on the continuing processes is about equal to the corresponding non-adaptive case, so processes do not page-in more pages after the adaptation than without an adaptation, as there is no data redistribution. Therefore, there is also no slowdown, once work is resumed. Explicitly moving pages off the leaving processes is by far the major cost component.

7.7 Evaluation

In this section we attempt to establish some general criteria by which we may judge whether adaptations for given applications and scenarios are cheap or expensive. For this, we use results from this chapter, and we also compare the results of the different
### Table 7.3: Average costs for one join from 7 to 8 threads, in seconds resp. number or percentage of pages transferred.

<table>
<thead>
<tr>
<th>Application</th>
<th>$t_{total}$</th>
<th>$t_{appl}$</th>
<th>$p_{appl}$</th>
<th>$p_{share_{appl}}$</th>
<th>$t_{scaled}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>0.63</td>
<td>-0.11</td>
<td>773</td>
<td>50.3%</td>
<td>4.19</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>0.94</td>
<td>0.91</td>
<td>2069</td>
<td>19.2%</td>
<td>0.90</td>
</tr>
<tr>
<td>MGS</td>
<td>1.14</td>
<td>1.11</td>
<td>4036</td>
<td>32.8%</td>
<td>0.95</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.68</td>
<td>1.64</td>
<td>3480</td>
<td>28.4%</td>
<td>1.40</td>
</tr>
<tr>
<td>NBF</td>
<td>0.16</td>
<td>0.13</td>
<td>3559</td>
<td>26.7%</td>
<td>0.12</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.41</td>
<td>0.35</td>
<td>235</td>
<td>2.4%</td>
<td>0.43</td>
</tr>
<tr>
<td>SOR</td>
<td>2.15</td>
<td>2.12</td>
<td>4258</td>
<td>28.4%</td>
<td>1.47</td>
</tr>
<tr>
<td>Water</td>
<td>-0.03</td>
<td>-0.06</td>
<td>-60</td>
<td>-15.5%</td>
<td>-0.80</td>
</tr>
<tr>
<td>TSP</td>
<td>0.04</td>
<td>0.02</td>
<td>21</td>
<td>19.4%</td>
<td>3.79</td>
</tr>
</tbody>
</table>

### Table 7.4: Average costs for one leave from 8 to 7 threads, in seconds resp. number or percentage of pages transferred.

<table>
<thead>
<tr>
<th>Application</th>
<th>$t_{total}$</th>
<th>$t_{system}$</th>
<th>$p_{system}$</th>
<th>$p_{share_{system}}$</th>
<th>$t_{appl}$</th>
<th>$p_{appl}$</th>
<th>$p_{share_{appl}}$</th>
<th>$p_{share_{system and appl}}$</th>
<th>$t_{scaled}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>0.26</td>
<td>0.00</td>
<td>0</td>
<td>0%</td>
<td>-1.11</td>
<td>770</td>
<td>50.1%</td>
<td>50.1%</td>
<td>1.73</td>
</tr>
<tr>
<td>3D-FFT</td>
<td>0.87</td>
<td>0.27</td>
<td>707</td>
<td>6.6%</td>
<td>0.57</td>
<td>2011</td>
<td>18.7%</td>
<td>25.3%</td>
<td>0.83</td>
</tr>
<tr>
<td>MGS</td>
<td>1.27</td>
<td>0.23</td>
<td>576</td>
<td>4.7%</td>
<td>0.98</td>
<td>3952</td>
<td>32.2%</td>
<td>36.8%</td>
<td>1.06</td>
</tr>
<tr>
<td>Jacobi</td>
<td>1.72</td>
<td>0.56</td>
<td>1522</td>
<td>12.4%</td>
<td>1.11</td>
<td>3704</td>
<td>30.2%</td>
<td>42.7%</td>
<td>1.44</td>
</tr>
<tr>
<td>NBF</td>
<td>0.93</td>
<td>0.57</td>
<td>1564</td>
<td>11.7%</td>
<td>0.31</td>
<td>3694</td>
<td>27.7%</td>
<td>39.5%</td>
<td>0.72</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.22</td>
<td>0.52</td>
<td>1045</td>
<td>10.7%</td>
<td>-0.36</td>
<td>-424</td>
<td>-4.3%</td>
<td>6.3%</td>
<td>0.23</td>
</tr>
<tr>
<td>SOR</td>
<td>2.18</td>
<td>0.70</td>
<td>1872</td>
<td>12.5%</td>
<td>1.48</td>
<td>4256</td>
<td>28.3%</td>
<td>40.8%</td>
<td>1.49</td>
</tr>
<tr>
<td>Water</td>
<td>0.00</td>
<td>0.01</td>
<td>34</td>
<td>8.8%</td>
<td>-0.05</td>
<td>-17</td>
<td>-4.4%</td>
<td>4.4%</td>
<td>0.00</td>
</tr>
<tr>
<td>TSP</td>
<td>0.07</td>
<td>0.00</td>
<td>14</td>
<td>13.0%</td>
<td>0.04</td>
<td>117</td>
<td>198.3%</td>
<td>121.3%</td>
<td>6.64</td>
</tr>
</tbody>
</table>

Applications with the help of four new quantities we introduce, $t_{scaled}$; $p_{share_{system}}$; $p_{share_{appl}}$; and $p_{share_{system and appl}}$.

$T_{scaled}$ expresses the total adaptation cost $t_{total}$ in relation to an application’s overall use of shared memory $m$. We calculated $t_{scaled}$ as $t_{total} * \frac{mMB}{40MB}$, scaling the results of the various applications to a hypothetical default use of 40MB of shared memory\(^2\).

$P_{share_{system}}$, $P_{share_{appl}}$ and $P_{share_{system and appl}}$ express the number of $p_{system}$, $p_{appl}$ resp. $p_{system} + p_{appl}$ shared memory pages moved extra due to the adaptation as a percentage of the overall number of shared memory pages used by the application.

Tables 7.3 and 7.4 repeat some of the results shown in Table 6.4, but in addition they contain the $t_{appl}$, $p_{appl}$, $P_{share_{appl}}$ and $t_{scaled}$ values, plus for leaves the $t_{system}$, $P_{system}$, $P_{share_{system}}$, and $P_{share_{system and appl}}$ values.

We can estimate adaptation costs using the following criteria. We first list five categories characterizing different applications, thereafter we list four runtime-criteria applicable to any application in general:

\(^2\)40MB is close to actual values we used for many of the evaluated applications (Table 5.2).
• Adaptation costs are generally the lowest in applications with an independent data layout, followed by applications with irregular data layout. Applications with a regular data layout have significantly higher adaptation costs. In this category, a block-cyclic data layout is maybe somewhat less expensive for adaptations than a block data layout, even though the former transfers more data due to an adaptation.

Tables 7.3 and 7.4 show that $t_{\text{scaled}}$ is lowest for Quicksort (independent). When averaging joins and leaves, NBF (irregular) has a slightly higher $t_{\text{scaled}}$, and MGS (block-cyclic) has much higher $t_{\text{scaled}}$ values still. Jacobi and SOR (block) have the most expensive $t_{\text{scaled}}$ results (Barnes is more expensive for other reasons as stated below, and TSP’s $t_{\text{scaled}}$ values are not representative due to the very small times actually measured).

• Regular applications have similar overall adaptation costs for joins and leaves. Although the data redistribution by the application is cheaper for leaves, the addition of the cost to move pages off leaving processes leads to similar overall costs.

Tables 7.3 and 7.4 show how the costs of joins and leaves are similar per application for each of the regular programs 3D-FFT, Jacobi, MGS and SOR.

• Adaptations are more expensive the larger the share of data accessed by the slave processes.

Tables 7.3 and 7.4 show how the $t_{\text{scaled}}$, $p_{\text{share.system}}$, $p_{\text{share.appi}}$, and $p_{\text{share.system.and.appi}}$ values are significantly lower for 3D-FFT than for Jacobi or SOR. Our measurements show that about one third of all data in 3D-FFT is used only by the master, whereas the other two programs have a balanced use of data by all processes. Therefore, in 3D-FFT proportionately less data is moved at or after an adaptation, as the master never joins or leaves.

• Adaptations are more expensive the more data is shared among processes, especially for joins. The share of data repeatedly accessed by one process may be far greater than $\frac{1}{n}$ for $n$ processes.

In Barnes, all processes often repeatedly access about half of all data. Tables 7.3 and 7.4 show that $p_{\text{share.system}}$, $p_{\text{share.appi}}$, and $p_{\text{share.system.and.appi}}$ are very high for both joins and leaves, and therefore also $t_{\text{scaled}}$. Joins are especially more expensive due to the bottleneck of paging-in half of all data by each joining process.
• Adaptations are cheaper the more of the data space is only read and not written anymore after the adaptation.

This is one of the reasons for lower $t_{\text{scaled}}$ values for 3D-FFT and NBF in Tables 7.3 and 7.4.

• Adaptations are cheaper the more processes are used. For joins, each doubling of the number of processes roughly halves a join’s cost. For leaves, the cost scales down somewhat less than for joins.

• Adaptations with multiple adapt events are typically only a fraction more expensive or in some cases up to double as expensive as single adapt events. A batching of adapt events therefore greatly reduces the average cost per event.

• Adaptation costs scale proportionally with an application’s problem set size’s use of shared memory.

• Adaptation costs scale more or less proportionally with the network latency, as the data redistribution cost determines the adaptation cost.

7.8 Summary

This chapter has shown that adaptation costs are overwhelmingly determined by \textit{data redistribution costs}. In most cases, the $p_{\text{appl}}$ extra data movement by the application after resuming the computation costs substantially more than the $p_{\text{system}}$ data movement by the system during the actual adaptation. However, the latter varies little for different application types, whereas the former’s cost for independent and irregular applications may be only a fraction of the cost experienced by other application types.

The data redistribution costs for regular applications are for their part largely determined by the maximum extra page requests and replies handled by any one process due to the adaptation. In irregular and independent applications, the correlation is less strict, but a trend applies nonetheless. The network latency directly determines the cost of a page transfer and therefore the adaptation cost.
Chapter 8

Implementation

This chapter provides details about the implementation of the Adaptive TreadMarks system. We have used TreadMarks version 1.1.0 as a base for our implementation and describe some of its mechanisms in the first section below, focusing on the issues relevant for the support of adaptivity. We assume that the reader is already familiar with the basic TreadMarks mechanisms used to ensure memory consistency, as described in various publications [48, 46, 6, 3, 5]. The remainder of the chapter then deals with the enhancements that allow adaptive parallelism for data parallel and task parallel applications.

8.1 TreadMarks

TreadMarks implements the lazy release consistency (LRC) protocol and uses diffs to avoid transferring whole pages back and forth among processes when multiple processes write to the same page\(^1\). Write notices are used to determine the correct order when applying several diffs. Although a process may in some cases need to fetch diffs for one page from more than one process, it can always determine the location of needed data on the basis of local information. With time, the number of diffs for a given page, as well as other memory consistency information, may grow. If the memory used exceeds some set limits, the system may free memory again by performing a garbage collection, as described below.

TreadMarks supports the OpenMP fork-join style of parallelism with the \texttt{Tmk.fork} and \texttt{Tmk.join} routines for the master process and the \texttt{Tmk.wait} routine for slave processes (Figures 8.1, 8.2 and 8.3). Their semantics is very similar to normal barriers - there are only two important differences: (1) \texttt{Tmk.join} corresponds to a barrier

\(^1\)Version 1.1.0 actually uses diffs only for pages using the multiple-writer protocol, i.e. pages with false sharing, while other versions also use diffs to encode writes to pages using a single-writer protocol.
create interval
wait for all slave processes to arrive, incorporate each slave’s consistency information

Figure 8.1: Tmk_join: Non-adaptive execution.

arrival and Tmk_fork corresponds to a barrier departure. Splitting up a barrier into two separate routines lets the master process optionally execute sequential program sections in-between parallel sections and at the start and end of a program. During such times, all the slave processes are simply idle, waiting to receive the next departure message. (2) Slave processes perform work only as specified by the master in the preceding Tmk_fork, instead of continuing with some previous work as in the case of normal barriers. In every Tmk_fork, the master sends all slave processes an application function pointer, all arguments for this function, and virtual shared memory addresses. The slave processes then call that function and use the shared memory addresses to access the correct shared memory pages, as all processes use the same virtual addresses for this.

Just as after a barrier, all processes have up-to-date memory consistency information after having incorporated the information received in the departure message. At this point, the system may therefore perform a garbage collection. This operation either updates pages, fetching and applying any diffs as needed, or discards pages, such that all pages end up either valid or discarded. From here on, any process trying to access a discarded page issues a page request to get a valid copy, so it suffices to keep all page state information, including the location of a valid copy — all other memory consistency information accumulated before the garbage collection is not needed anymore. All diffs, twins, interval records and write notices are discarded and all vector timestamps reset to their initial zero values upon completion of a garbage collection. As all processes have up-to-date memory consistency information when initiating a garbage collection, they all perform identical decisions on which processes keep valid copies of which pages. In general, the process with the newest write notice for a page is the one keeping a valid copy of a page, the other processes discard invalid copies.

8.2 Supporting adaptivity

The system distinguishes between data parallel and task parallel applications through omission or addition of the -T command-line option (cf. Section 3.3.2). The system requires this information to decide whether to send an extra message to all processes to inform them about an impending adaptation. Task parallel applications need such a notification, so that processes prepare to adapt upon completion of their current task, whereas data parallel applications include this notification in the next OpenMP
create interval
send departure messages to all processes, containing:
  missing consistency information + function pointer and arguments + shared memory addresses
if (little memory left on heap)
  perform garbage collection
  barrier #1
  reset and discard consistency data
  barrier #2
end
if (application function pointer was passed)
  call application function
end

Figure 8.2: Tmk.fork: Non-adaptive execution.

for (;;) {
  create interval
  send arrival message with slave's consistency information
  receive departure message
  incorporate received consistency information
  if (little memory left on heap)
    perform garbage collection
    barrier #1
    reset and discard consistency data
    barrier #2
  end
  if (application function pointer was received)
    call function
  else
    return 0
  end
}

Figure 8.3: Tmk.wait: Non-adaptive execution.
fork messages.

8.2.1 Before an adaptation

Join and leave requests may be sent to the system anytime from any external source via a TCP/IP connection.

For a join event, the master spawns a new process \( p_{new} \) on the designated machine and \( p_{new} \) sets up network connections, first to all other slave processes and finally to the master process. In the meantime, all old processes continue with their work, so any slow low-level process initializations do not affect the on-going computation, except for a very brief interruption (less than a millisecond) while the connection requests are being serviced. Once the master is connected, it knows that all connections are in place, and the new process may join the computation at the next opportunity.

Once a new process is fully connected, or for a leave event, immediately after receipt of the request, the system may perform the adaptation at the next adaptation point. In data parallel applications, this means processes continue until completion of the current application function, upon which they reach an adaptation point, whereas in task parallel applications, the master process now sends a ‘prepare-to-adapt’ message to every slave process, such that these begin an adaptation at the next \texttt{Tmk.leave}, which marks task boundaries. Exceptionally, if a process does not encounter a \texttt{Tmk.leave} but instead finishes the application function it was executing, it begins the adaptation there, as in the case of data parallel applications (cf. Section 8.2.4, “Special cases”).

When several adapt requests are pending, the system may in certain cases also postpone some of them and execute them at a later adaptation point, as described hereafter.

Multiple adapt requests

The system registers any adapt requests it receives and usually executes them as soon as possible according to the policy described in Section 4.1.2, such that several processes may leave and join in one go.

As the current policy always favors leave requests over join requests whenever there is a conflict, a pending leave request at an adaptation point causes a new process with as-yet incomplete network connections to be abandoned and terminated there, such that all pending leaves and any ready-to-join new processes can be handled there. Immediately after the adaptation, the system starts a fresh join attempt by spawning the new process again on the same machine as before.

The reason for the above is given by the implementation. Performing an adapta-
tion while low-level connections are being set up would be tricky: Consider, e.g., a process $P_0$ leaving while the new process attempts to connect to $P_0$, not knowing yet about $P_0$’s withdrawal.

Also for the sake of implementation simplicity, currently only one process at a time is permitted to set up network connections: Consider, e.g., a second new process $P_2$ trying to set up connections to a first new process $P_1$ while $P_1$ has no knowledge about $P_2$ yet, as it has not been able to receive the master’s notification thereof. Handling new process connections sequentially avoids such complications.

Any join requests arriving while another new process is still attempting to set up network connections is deferred. The next new process is spawned as soon as the preceding new process is fully connected, whereby an abandoned new process is handled first, followed by any deferred join requests. Multiple join requests arriving while any other join requests are still pending are thus batched and all join the computation in one go, unless interrupted by leave requests, as stated above. For data parallel applications, this means that any other adaptation points passed in the meantime are not used for an adaptation, even if some of the new processes could already join the computation there, and for task parallel applications, this policy means that the master only asks the slave processes to prepare for an adaptation once all the requested new processes are fully connected. The exceptions are leave requests arriving in-between - they are handled immediately at the next adaptation point.

### 8.2.2 Performing the adaptation

This section provides an overview of the major actions comprising the actual adaptation. The next section then examines in more detail some of these actions, namely the specific steps performed to guarantee the memory consistency. The current section thus shows how and where those memory consistency operations fit in in the sequence of events performed at an adaptation.

Section 8.2.4 shows how task parallel applications additionally employ `Tmk.leave`, from where the other routines `Tmk.join`, `Tmk.fork` and `Tmk.wait` are called.

**Master thread**

The master thread reaches an adaptation point when it arrives at a call of the `Tmk.join` routine (Figure 8.4), which implements an OpenMP join. Now the master updates its memory consistency data with all the missing information sent by the slave processes, as in a normal barrier. New processes that have all network connections in place are registered such that the master also expects an arrival message from these
if (no partially-connected new process exists and new processes exist)  
    let new processes join computation  
end  
create interval  
register processes with leave requests for withdrawal  
wait for all slave processes to arrive, incorporate each slave’s consistency information  
if (a partially-connected new process exists)  
    abort that process  
end  
set adaptation state variables  
calculate new process ids

Figure 8.4: Tmk_join when adapting.

processes, indicating their completion of any initializations. A partially-connected new process is aborted and taken note of, such that it can be restarted immediately after the adaptation. The system then sets various variables used to capture the details of the current adaptation, and new process identifiers are chosen according to the algorithm outlined in Section 7.3.1.

The master now executes any present sequential section application code, until it arrives at the next call of the Tmk_fork routine (Figure 8.5), which implements an OpenMP fork. Here, the master sends departure messages with information about joining and leaving processes and process identifier reassignments. All non-joining processes also get any missing consistency information, so that they can participate in the ensuing garbage collection. If the application is of the data parallel type, all non-leaving processes receive the address of the function they are to execute in the next parallel section, plus any function arguments. In task parallel applications, the current OpenMP join-fork sequence is typically called from within Tmk_leave and is therefore not present in the application code, so after the adaptation all continuing processes must resume work using the same function as before the adaptation, and all joining processes must begin work using that same function. In that case, Tmk_fork is invoked with NULL parameters, as we see later on, thereby continuing processes do not receive a new function pointer. Joining processes however are sent the function pointer of the previous Tmk_fork call by the application, which has been stored temporarily. In all cases, the NULL function pointer passed to leaving processes lets these return and exit gracefully upon completion of the adaptation.

All non-joining processes now perform a garbage collection, which greatly reduces the amount of memory consistency information, as most of it is discarded (cf. Section 8.1).

The system then performs three barrier synchronizations. In the first one, all processes update their memory consistency information in accordance with the adap-
create interval
send departure messages:
  send to continuing processes:
    adaptation data + missing consistency data + function pointer and arguments + shared memory addresses
  send to leaving processes:
    adaptation data + missing consistency data + NULL pointer
  send to joining processes:
    adaptation data + function pointer and arguments + shared memory addresses
perform garbage collection

barrier #1:
  if (any process is leaving)
    determine all pages only valid on leaving processes and select new owners
    for any pages for whose consistency any leaving process would have been used (Figure 8.8)
    send to each continuing slave q a message with designated new owners for all pages
    for which q required any leaving process for consistency (Figure 8.10)
    send default departure message to leaving slaves
  else
    send default departure message to continuing slaves
end
if (any process is joining)
  send all non-default page state information to joining processes (Figure 8.9)
end
if (any process is leaving)
  fetch the share of pages only valid on leaving processes assigned to local process, if any (Figure 8.13)
end

barrier #2
if (local process not leaving)
  close all connections to leaving processes
  perform process-id and associated reassignments
end
if (local process not joining)
  reset and discard consistency data
end
if (local process not leaving)
  barrier #3
end
if (aborted join process or deferred join process exists)
  spawn new process on designated machine
end
if (application function pointer was passed)
  call function
end

Figure 8.5: Tmk_fork when adapting.
More precisely, if the adaptation includes any leaves, all pages that are only valid on leaving processes must be transferred to some continuing process, so in this case all old slaves include page state information in the arrival message, as detailed in Section 8.2.3. With this data, the master can deduce the set of pages valid only on leaving processes, and it allocates an approximately equal number of such pages among some chosen processes (usually the continuing processes, exceptionally the joining processes (cf. Section 8.2.3, “Master thread”)) and includes this information in the departure messages. Any joining processes also get all non-default page state information for all pages from the master. Thanks to the garbage collection, new processes do not need to see any other memory consistency information, which has been discarded. Even so, the amount of information sent to joining processes often exceeds the MTU, requiring several messages.

After receiving the departure messages, in case of leaves, processes in parallel fetch pages from withdrawing processes, as allocated by the master.

This first barrier guarantees that garbage collection is completed before page state information is sent to any processes and in the case of leaves before the master begins to reallocate pages and those pages are fetched.

Next, a second barrier is performed, followed by all necessary reassignments, including the redistribution of lock managers and lock tokens, and all memory consistency information is cleared except for page state data.

This second barrier ensures that in the case of leaves any page transfers off leaving processes are herewith completed, and any duplicate departure messages of the first barrier are not sent after some process has already begun with any reassignments.

Thereafter a third barrier is performed without participation of any leaving processes. Any data needed from such processes has already been fetched, so they are now terminated.

This third barrier ensures that no process can proceed with its computation before all processes have performed reassignments and cleared their old consistency information.

Finally, if any abandoned or deferred join requests are pending, a new process is spawned accordingly, then processes resume or begin work.

For task parallel applications, Figures 8.4 and 8.5 only show what happens in the common case. Certain special cases described in Section 8.2.4 require extra or alternative actions, as shown in Figures 8.18 and 8.19.

**Slave threads**

Slave threads reach an adaptation point when they complete the application function sent by the previous OpenMP fork and wait for the next one, or, for task parallel
for (;;) {
    create interval
    send arrival message with slave’s consistency information
    receive departure message
    set adaptation state variables
    if (local process not joining)
        incorporate received consistency information
        perform garbage collection
    end
    barrier #1:
    if (any process is leaving and local process not joining)
        send lists with all pages requiring any leaving process for consistency
        and with all valid pages to master (Figure 8.7)
    else
        send default arrival message to master
    end
    receive departure message from master
    if (local process is joining)
        update all local page information with any non-default page state values and
        create list of any pages to be fetched from any leaving processes (Figure 8.11)
    elseif (any process is leaving but local process not leaving)
        reset writers and owners of all pages for which local process previously had writer or owner
        equal to any leaving process (Figure 8.12)
        fetch the share of pages only valid on leaving processes assigned to local process, if any (Figure 8.13)
    end
    barrier #2
    if (local process not leaving)
        close all connections to leaving processes
        perform process-id and associated reassignments
    end
    if (local process not joining)
        reset and discard consistency data
    end
    if (local process not leaving)
        barrier #3
    end
    if (application function pointer was received)
        call function
    else
        return 0
    end
}  

Figure 8.6: Tmk_wait when adapting.
applications, when they reach a task boundary and call the \texttt{Tmk\_leave} routine. For this, slaves use the \texttt{Tmk\_wait} routine (Figure 8.6), where an endless loop repeatedly receives the messages sent by the master's OpenMP forks and executes the specified function.

When arriving, slaves first send all their memory consistency information not seen yet by the master, as in a normal barrier. In the departure message, slaves receive all missing memory consistency data and incorporate it, whereupon they are ready to perform a garbage collection.

Thereafter, the system performs 3 barrier synchronizations. In the first barrier, if any process is leaving, slaves notify the master of all their valid pages and all pages for whose consistency they would have contacted a leaving process. Slaves now reset their page state data as indicated in the master's departure message, and if applicable they fetch pages from the leaving processes.

After the second barrier, non-leaving processes terminate all network connections to leaving processes and reassign process identifiers and other internal data, and non-joining processes discard most of the memory consistency information, as after a regular garbage collection.

Non-leaving processes now execute a third barrier and finally resume work by calling the specified function, if any, or returning to continue with the current function if the application is of the task parallel type and the process has interrupted an ongoing computation for the adaptation.

For task parallel applications, Figure 8.6 only shows what happens in the common case. Figure 8.20 illustrates how the system deals with certain special cases described in Section 8.2.4.

### 8.2.3 Maintaining consistency when adapting

**Slave threads at adaptation point arrival**

In an adaptation containing withdrawing processes, all slave processes use the routine shown in Figure 8.7 to construct a message comprising page-state information of the local process, with which the master is able to identify the set of pages that are only valid on leaving processes. Specifically, slaves send data about all pages with owner or writer fields pointing to any leaving process and all pages that are valid on the local process.

**Master thread**

In an adaptation containing withdrawing processes, the master process employs the routine shown in Figure 8.8 to merge its own page state data with all such data sent
assemble a message containing:

\[
\text{foreach page } p \\
\quad \text{if (writer or owner equals any leaving process)} \\
\quad \quad \text{add page-id} \\
\quad \quad \text{add code for potential fetch page and page state and local process-id} \\
\quad \text{elseif (page is valid)} \\
\quad \quad \text{add page-id} \\
\quad \quad \text{add code for page state and local process-id} \\
\text{end} \\
\text{end}
\]  

Figure 8.7: Slave processes: Send page state information to master process. Master can thereafter deduce pages only valid on leaving processes.

by slaves (Figure 8.7), letting the master determine all pages that are only valid on leaving processes.

A given page \( p \) may be valid on several processes at once, of which one \( P_{\text{leave}} \) may be leaving and another \( P_{\text{cont}} \) may be continuing. A third process having the writer of \( p \) set locally to \( P_{\text{leave}} \) can simply reset that writer to \( P_{\text{cont}} \). Such pages need not be moved off the leaving \( P_{\text{leave}} \).

The master constructs a valid list storing for each page \( p \) either (1) the highest process identifier of any non-leaving process with a valid copy of \( p \), if such one exists, or else (2) the lowest process identifier of any leaving process with a valid copy of \( p \). By setting the page’s valid list entry to the new owner, the master next allocates approximately a \( \frac{1}{n} \) share of the pages of category (2) to each one of \( n \) processes as follows: Generally, these \( n \) processes are the set of continuing processes, including the master. As an optimization, when the number of leaving and joining processes is equal in the current adaptation and the application is of the data parallel type, the joining processes are chosen as the set of \( n \) processes. Such applications often have regular data distributions, and when the joining processes act as ‘replacement’ for the same number of leaving processes, they typically access about the same range of pages. Letting these processes fetch some needed pages directly from the leaving processes therefore avoids the ‘detour’ of moving them first to some third process.

In addition, the master locally changes its page state data of all pages for which the owner or writer field pointed to a leaving process. It resets these fields using valid list, either invalidating and discarding a page if a slave process is designated as owner, or inserting the page in the local fetch list of pages to be potentially fetched by the master, if the master is the chosen new owner.

Thereafter, the master uses the routine shown in Figure 8.9 to inform any joining processes of all non-default page state data, so that these processes know from where to request valid page copies and what protocol a page is currently using. For any
foreach page p
  if (local writer or owner equals any leaving process)
    put p in local change_owners_list
  end
store valid on master or invalid local state in valid_list
end
foreach slave process q
  foreach (entry in message from q)
    determine page p with page-id
    if (p is valid on q)
      if (p’s entry in valid_list is invalid or q is not leaving)
        change p’s entry in valid_list to valid on q
      end
    end
  end
end
foreach page p
  if (p is only valid on a leaving process)
    insert p in leave_pages_list
  elseif (p in change_owners_list)
    set writer = owner = valid_list[p]
    set protocol = single.writer
    if (valid_list[p] is a slave process)
      set empty = TRUE
      set state = invalid
    end
  end
end
if (# joining and leaving processes is equal and program-type is data_parallel)
  designate n joining processes each as new owner of about 1/n of leave_pages_list pages
else
  designate n continuing processes each as new owner of about 1/n of leave_pages_list pages
  foreach page p of leave_pages_list with master as designated new owner
    insert p in local fetch_list
    set writer = owner = master
    set protocol = single.writer
  end
end
foreach page p of leave_pages_list with any slave q as designated new owner
  disable r/w access
  set writer = owner = q
  set protocol = single.writer
  set empty = TRUE
  set state = invalid
end

Figure 8.8: Master process: Determine pages valid only on leaving processes and select new owners and writers for pages otherwise fetched from leaving processes.
assemble a message containing:

\[
\text{foreach page } p \\
\quad \text{if (non-default writer or non-default owner or non-default protocol)} \\
\quad \quad \text{add page-id and writer} \\
\quad \quad \text{if (a new process is designated new owner of } p) \\
\quad \quad \quad \text{add process-id of process with valid copy} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{add owner} \\
\quad \quad \text{end} \\
\quad \text{add protocol} \\
\text{end} \\
\text{end}
\]

Figure 8.9: Master process: Send current page state data to joining processes.

assemble a message containing:

\[
\text{foreach (entry in slave } q\text{'s arrival message)} \\
\quad \text{if (writer or owner equals any leaving process)} \\
\quad \quad \text{add page-id} \\
\quad \quad \text{add designated new owner} \\
\text{end} \\
\text{end}
\]

Figure 8.10: Master process: Send new page state data to continuing processes, for pages otherwise fetched from leaving processes.
foreach (entry in message)
    determine page $p$ with page-id
    set writer, owner, protocol of $p$
    if (local is designated new owner of $p$)
        insert $p$ in local fetch_list
    end
end

Figure 8.11: Joining processes: Receive current page state data from master.

foreach (entry in message)
    if (local process $q$ is designated to have valid copy of page $p$)
        insert $p$ in local fetch_list
        set writer = owner = $q$
        set protocol = single.writer
    else
        if (page state of $p$ is invalid)
            disable r/w access
            set empty = TRUE
        end
        set writer = owner = designated new owner
        set protocol = single.writer
    end
end

Figure 8.12: Continuing slave processes: Receive new page state data from master, for pages otherwise fetched from leaving processes.

page for which a joining process is designated as new owner, the master also sends the pre-adaptation location of a valid copy, so that the joining processes can fetch the page from there and assume ownership.

If any processes are leaving, the master uses the routine depicted in Figure 8.10 to notify each continuing slave process $x$ of the process selected to have a valid copy of any page $p$ for which $x$ previously had the owner or writer field pointing to a leaving process. For some of these pages, $x$ may have been selected as the new owner. The algorithm ensures that $x$ is correctly informed of all the pages it must fetch from some leaving process, because for such pages $x$ must have had the writer field equal to some leaving process, because the writer field of any locally invalid page points to a process with a valid copy.

Slave threads at adaptation point departure

Any joining processes receive all non-default page state data for all pages from the master and apply it locally using the routine shown in Figure 8.11. If the local process
foreach (page p in local fetch_list)
    if (pre-adaptation value of writer is not local)
        if (page-state is invalid)
            enable r/w access to p
            set empty = FALSE
            put p in per-process final_fetch_list of pages to fetch
    end
end
fetch all pages in final_fetch_list with multiple pages per message

Figure 8.13: All processes: Fetch the allocated share, if any, of pages valid only on
leaving processes.

has been selected as new owner for any page, its number and the location of a valid
copy are put in the local fetch_list of pages to be retrieved from a leaving process.

In an adaptation with leaves, each continuing slave process x uses the routine
shown in Figure 8.12 to update local page state data of any page for which it previously
had the owner or writer field pointing to a leaving process. Additionally, if x has been
chosen as new owner for any such page, its number and the location of a valid copy
are put in the local fetch_list of pages to be potentially retrieved from a leaving
process.

Moving pages off leaving processes

Figure 8.13 shows pseudo-code of the actions performed by any process assigned
ownership of any page previously potentially only valid on some leaving process. The
process ensures that such pages are not needlessly fetched in case they are valid locally
(the writer field of a valid page may have been outdated, pointing to some leaving
process), while other pages are fetched efficiently, with multiple pages requested and
sent in each request-reply roundtrip.

8.2.4 Extra actions for task parallel applications

Task parallel applications in principle adapt exactly as data parallel applications do,
save for the need to call the Tmk_join, Tmk_fork and Tmk_wait routines explicitly by
the system, as the application code typically does not contain (a sufficient number of)
such calls to serve as adaptation points. For this purpose, the Tmk_leave routine
(Figure 8.14) is employed. Aside from this, the descriptions in the preceding sections
appertain to task parallel applications as well.

The application calls Tmk_leave at task boundaries, as we have seen in Sec-
tion 3.3.2. If an adaptation is due, Tmk_leave simply executes an OpenMP join-
if (leave request or join request of connected process)
  if (master process)
    Tmk_join();
    Tmk_fork(NULL, NULL);
  else
    Tmk_wait(0);
end
return (leave request by local process?)

Figure 8.14: Tmk_leave.

void appl_func(...)
{
  do {
    if ((dequeue a task) == FALSE) { — Process A: Arrival of 'prepare-to-adapt' request
      break;
    }
  }
  execute task; — Process B: Arrival of 'prepare-to-adapt' request
  while (!Tmk_leave());
}

Figure 8.15: If the task queue is empty, invocation of Tmk_leave depends on when a
'prepare-to-adapt' request arrives.

fork sequence using the above routines. As no function is supplied, Tmk_fork and
the slaves’ counterpart Tmk_wait return immediately after the adaptation, except for
joining processes. For new processes, the system overrides the NULL parameters and
sends a pointer of the currently-active application function, which they now begin
executing. Subsequently, Tmk_leave returns with either a FALSE value in the common
case, or with a TRUE value to notify the application of the local process’ withdrawal,
in which case the application process must terminate.

Tmk_leave first checks whether an adaptation is pending. In the absence thereof,
this routine returns immediately and therefore does not add any significant overhead.

Special cases
Task parallel applications cannot adapt efficiently in all cases if adaptations may only
occur wherever the Tmk_leave routine is invoked. A program may easily contain
many different compute phases where different task queues are used and/or where
data parallel computations are performed. In other words, the application itself may
contain many OpenMP join-fork sequential sections, in addition to any such sequences
inserted extra by the system when adapting. Each of the application’s OpenMP fork-
join parallel sections may or may not contain a task queue, and adaptations must be
Tmk_fork(appl_func, args)
{
  ...
  call appl_func
  {
    ...
    Tmk.leave()
    ....
    ← ‘Prepare-to-adapt’ request arrives between here ...
  }
}
Tmk.join()
{
  ...
  wait for all slave processes to arrive
  ...
  if (perform task parallel adaptation here...) {
    (*)
    Tmk_fork(NULL, NULL);
    Tmk.join();
  }
}

Figure 8.16: Master process: If a ‘prepare-to-adapt’ request arrives such that the master does not pass Tmk.leave, the master performs the adaptation while within Tmk.join.

Tmk.wait()
{
  for (;;) {
    ...
    send arrival message
    recv departure message
    ← ‘Prepare-to-adapt’ request arrives while waiting for message
    ...
    call appl_func
    {
      ...
      Tmk.leave()
      {
        if (perform task parallel adaptation here...) {
          FALSE
        Tmk.wait(0);
      }
    }
  } OR continue OR return 0
}

Figure 8.17: Slave processes: If a ‘prepare-to-adapt’ request arrives such that the process does not pass Tmk.leave, it performs the adaptation outside of Tmk.leave.
create interval

... wait for all slave processes to arrive, incorporate each slave's consistency information

... calculate new process ids
if (adaptation of task parallel program due and local process inside application join-fork sequence)
    Tmk_fork(NULL, NULL);
    Tmk_join();
end

Figure 8.18: Details of Tmk_join showing extra code added for adaptations of task parallel applications.

create interval

send departure messages to processes, containing:
    continuing processes:
        adaptation data + missing consistency data + function pointer and arguments + shared memory addresses
    leaving processes:
        adaptation data + missing consistency data + NULL pointer
    joining processes of a data parallel application:
        adaptation data + function pointer and arguments + shared memory addresses
    joining processes of a task parallel application when master inside an extra join-fork sequence:
        adaptation data + previous function pointer and arguments + shared memory addresses
    joining processes of a task parallel application when master inside an application join-fork sequence:
        adaptation data + function pointer and arguments + shared memory addresses
        + notification that master is inside an application join-fork sequence

perform garbage collection
perform adaptation with 3 barriers
if (application function pointer was passed)
    call function
end

Figure 8.19: Details of Tmk_fork showing extra code added for adaptations of task parallel applications.
for (;;) {
    create interval
    send arrival message
    receive departure message
    ...
    perform garbage collection
    perform adaptation with 3 barriers
    if (application function pointer was received)
        call function
    elseif (adaptation of task parallel program is due AND
        (local process is inside an application join-fork sequence OR
        (local process is joining AND master is inside an application join-fork sequence)))
        if (local process is leaving)
            return 0
        else
            continue
        end
    else
        return 0
    end
}
possible at any time.

Consider a program about to finish one task queue, the queue itself is empty but most processes are still busy solving the final tasks, and no process will produce any new tasks. If a ‘prepare-to-adapt’ request arrives at such a time, depending on where each process momentarily is in its execution, some processes may subsequently still pass `Tmk.leave` once more before learning that the queue is exhausted, while other processes may only attempt to dequeue a new task, and failing to do so, return from the current function without passing `Tmk.leave` anymore. Such scenarios are depicted in Figure 8.15.

Figures 8.16 and 8.17 show the effects in the larger context of `Tmk.join` and `Tmk.fork` on the master process, and `Tmk.wait` on slave processes, respectively. A master process not passing `Tmk.leave` after receiving a ‘prepare-to-adapt’ request continues on and returns from the current `Tmk.fork` and enters the next `Tmk.join`, i.e. the next OpenMP join, or it may already be waiting there, while a slave process may similarly finish the current application function and next wait for the master to send an OpenMP fork message. Recall that a ‘prepare-to-adapt’ request means the system adapts at the next OpenMP join-fork sequence. Processes finding themselves in an above-mentioned situation now attempt to adapt at such a sequential section found in the application code, while at the same time other processes may be inside `Tmk.leave` and try to adapt there at an extra sequential section initiated by the system, of which the application is not aware.

Performing the adaptation correctly in such cases is not trivial for the system. The processes having arrived at the application OpenMP join may not begin the next parallel section until all processes now in the `Tmk.leave` somehow continue and also arrive at the application OpenMP join, so that all processes begin the next parallel section simultaneously, so that the barrier-like semantics of the join-fork are preserved.

Note that the processes executing the `Tmk.leave` in the above scenario would perform one join-fork sequence more than the other processes arriving directly at the application join-fork sequence. One solution for this is to send to and receive from the former processes extra messages, letting them arrive at the application OpenMP join. These processes are not aware of the fact that some other processes might not have passed a `Tmk.leave`, so they require some notification thereof. An alternative idea is simply to let all processes in a task parallel program execute an extra join-fork sequence when adapting. We have implemented this approach and show the details for `Tmk.join`, `Tmk.fork` and `Tmk.wait` in Figures 8.18, 8.19 and 8.20, respectively.

If a ‘prepare-to-adapt’ request is pending when the master arrives in an application OpenMP join, the master executes an extra fork-join sequence with a NULL function pointer, just as if it were inside `Tmk.leave`. There are only two small differences in the
fork: (1) The master sends the current function pointer — i.e. \texttt{NULL} — not only to continuing, but also to any joining processes, and (2) the master notifies the joining processes to expect another departure message: The message sent to all processes in the following application OpenMP fork, containing the new function pointer for all processes. Correspondingly, the slave processes also check whether they are inside \texttt{Tmk.leave} or not. If not, they continue and wait for the master's next departure message, whereas the slaves inside \texttt{Tmk.leave} plus any leaving processes arrive at a return statement, ending this recursive \texttt{Tmk.wait} call, and they thus carry on with the compute function already begun.

If a 'prepare-to-adapt' request arrives at a later moment \( t \) than shown in Figures 8.16 and 8.17, i.e. once the master process has received all slave process OpenMP join arrivals, then the request is handled normally at the next application OpenMP join or the next \texttt{Tmk.leave}, whatever comes first for each process. In particular, even if the master is inside an application OpenMP join and at \( t \) it has not passed the test at (*) in Figure 8.16 yet, the master still does not perform the extra fork-join sequence there anymore.
Chapter 9

Related work

In this chapter we provide an overview of related work and discuss how our contribution positions itself with respect to other work.

The general idea of somehow adapting work to the varying availability of a set of networked compute resources has been pursued in various ways for some time. We observe first of all that many systems have some aspects in common with our system while differing in others, and that a strict classification is not always straightforward, as the boundaries between different types of systems are sometimes not clearly discernible. Nevertheless, we attempt to categorize related work according to what type of adaptivity and parallel processing the various systems offer.

In the first section, we survey systems that provide adaptivity in NOWs, but unlike Adaptive TreadMarks, they offer no parallel processing. In Section 9.2, we present systems for parallel processing in NOWs, but without adaptivity. In Section 9.3 finally we review systems supporting both parallel processing and adaptivity in NOWs, and we further distinguish between different forms of adaptivity. We make a brief excursion to multiprocessor systems and end with a comparative overview of adaptively parallel systems.

9.1 Adaptive computing

In this section we present systems that may dynamically adapt their use of compute resources in a network of workstations environment, but they do not support parallel computations. In many ways, they are a precursor of newer systems offering true adaptive parallelism.
9.1.1 Sequential computing

A number of systems permit sequential computations to use idle cycles on networked nodes, among others Butler [60], Condor [54] and Sprite [29]. Butler arranges for a command entered by a user to be executed remotely on an idle workstation belonging to the pool of currently-available machines, while attempting to mimic the effect of an equivalent command executed locally. Condor is more sophisticated, letting users submit jobs that are queued and executed as batch jobs on otherwise idle machines. The Condor library transparently forwards some of a user program’s Unix system calls from the executing to the submitting machine, but certain system calls are not permitted. Sprite is similar to Condor in that it also executes batch jobs remotely and forwards certain system calls on to some previous or home machine. Condor and Sprite can both adapt to the changing availability of a machine by migrating jobs off machines that become unavailable. For this, Condor checkpoints a running job to disk using the core dump mechanism and restarts the job on another machine, while Sprite transfers the various components of a process’ state to the new machine. However, Sprite suffers from residual dependencies, i.e. the on-going need for a machine to provide some functionality or maintain some data structures for a job that has been evicted from the machine.

9.1.2 Distributed computing

Another category of early systems supports distributed processing in a NOW with some form of adaptivity. However, for some systems one cannot easily resolve whether their nature is more distributed or more parallel. Generally, we regard systems with no or very infrequent interaction among slave processes of the same application as distributed. Two such systems are Marionette [77] and the Benevolent Bandit Library (BBL) [32]. Marionette invokes asynchronous RPCs executed by slave processes and it can recover from slave processor failures by restarting the operation on a new machine. Slaves only share data with the master, simplifying recovery. In addition, the pool of available machines can be adjusted at runtime. BBL is perhaps the first system that may be declared as supporting some form of adaptive parallelism. It executes distributed algorithms on networked IBM PCs running DOS: A machine is immediately returned when an owner begins to work again and the affected code is restarted on some other free machine. An example BBL application is a parallel merge sort, where slave processes exchange data infrequently, namely in-between sorting phases.
9.2 Parallel computing

In this section we review representative examples of systems that have been devised for parallel computing across networks of workstations, but without a capability to adapt the processor sets used during an application’s execution. However, many of these systems have served as a base for the development of adaptive systems and/or contain many mechanisms used by such systems.

Message passing is still widely used for parallel computations, specifically PVM [78] or MPI [33], either directly or as an underlying substrate for another programming paradigm. Many systems also offer a shared memory abstraction in a distributed memory environment such as a NOW. Memory consistency is either maintained at the granularity of pages or of objects. Typical examples of the page-based approach include Ivy [53], TreadMarks [48], CarlOS [50], Cashmere [76], Millipede [42], Quarks [79], and CVM [80], whereas representatives of the object-based approach include Linda [34], Munin [14], Midway [15], SAM [68], DOSMOS [17], Locust [24], CRL [44], Shasta [67], and DOSA [39].

Parallel computing on shared-memory multiprocessor machines is easier to implement due to the hardware support for the shared memory abstraction. As we focus primarily on distributed memory environments in our work, we do not examine such systems more closely except to note that one such system is perhaps one of the earliest where the term “adaptive parallelism” was employed, albeit with a different meaning: In the context of a parallel implementation of the PROTEAN program [37], adaptive parallelism was used to denote the variable grain size of individual tasks at runtime, whereas we use this term to express a potential variation of the processor set used by an application at runtime.

9.3 Adaptive parallel computing

In the rest of this chapter we survey parallel computing systems that may adapt their use of compute resources at runtime of an application. We differentiate between various forms of adaptivity. In this context, we first discuss how adaptive parallelism and load balancing relate, and thereafter we classify various adaptively parallel systems by three categories we establish.

Adaptive parallelism or load balancing?

Adaptive parallelism typically contains load balancing issues: Expanding a computation onto extra nodes and/or withdrawing a computation from some nodes in most cases requires a reduction resp. an increase of the load on the continuing nodes. The
key differences between the two concepts is that load balancing can only reduce part of a parallel application’s load on a given processor, while adaptive parallelism should be capable of removing all load from that processor, and it can include new, initially unknown processors. However, the distinction is a little more subtle: Consider a set \( N \) of processors of which a subset \( M \) are unavailable for the moment. Now, a parallel application might be initiated on the \( N \) processors, but immediately thereafter the load could be rebalanced in such a way that the \( M \) processors get a zero compute load, without yet having done useful work. Further adaptations could be performed by switching the allocated load on some processor from zero to some fractional value and vice versa. However, the framework of the computation is in some way still present on all \( N \) processors, it may simply be idle on \( M \) of these. Even if the system permits reducing the compute load to zero on those processors, other resources remain occupied, so the resource use is not zero. Should such a system qualify as adaptively parallel or only as being able to balance load? – The adaptive parallelism model we suggest permits all processes and resources previously in use by a parallel computation to be terminated resp. freed on any machine from which a parallel computation withdraws.

Some systems require the presence of some daemon process on any machine to be considered for an expansion of an on-going computation. Although such processes occupy resources, we consider such a system adaptively parallel if the daemon process can be terminated on a withdrawn machine \textit{without} affecting the computation continuing on some other machines. Adaptive parallelism should allow machines withdrawn from an on-going computation to be switched off or rebooted, and vice-versa to connect new machines to a network and then include them in a previously-started computation.

The goal of the above elucidation is to highlight important issues to be considered when evaluating such systems, and to show that systems declared as adaptively parallel may actually not be truly adaptively parallel if a more strict definition is used.

Three categories of systems

Based upon the above, we establish \textit{three categories of systems}: (1) Purely load-balancing systems on which a node’s load can only be reduced partially; (2) Load-balancing systems that permit a node’s compute load to be reduced to zero, but some of the node’s resources remain occupied throughout the computation; (3) Adaptively parallel systems where a withdrawing node may truly be disconnected from the other nodes harboring the on-going parallel computation, and vice versa new nodes may be included at any time.
The above three categories are closely related, and we cannot always uniquely classify a system. Nevertheless, in the next three subsections we provide an overview of systems belonging to category (1), (2) and (3), respectively. Adaptive TreadMarks distinguishes itself from the first two categories by providing true adaptive parallelism as characterized by the third category. A further unique trait is the support of an industry standard shared memory API, as evidenced by the overview at the end of the chapter.

9.3.1 Parallel computing with load-balancing

In this subsection we briefly survey a number of parallel processing systems that support only a load balancing-type of adaptivity where the compute load assigned to individual processes/threads may vary during runtime without being reduced to zero. The pool of processors actively working on a parallel program therefore remains static in such systems. Nevertheless, load balancing and truly adaptively parallel systems often employ many similar mechanisms, motivating us to discuss both types of systems.

One approach is to balance load by remapping and adjusting the data partitions that each process is working on. Examples for this are a version of the TreadMarks system with a load balancing extension [41] and the DOME system [12]. In TreadMarks, the iterations of each parallel loop of SPMD programs are automatically partitioned among the processes according to previous loops’ runtimes, thus rebalancing the computation in the advent of load. This scheme of iteration repartitioning is actually very similar to the mechanism we use for repartitioning among a changed number of processes. DOME [12] provides load balancing on a heterogeneous NOW. The data reallocation and redistribution is again handled automatically by the system, but programs must be of the SPMD type and use the DOME library. Rebalancing can then occur after each DOME operation.

Other systems use migration to achieve load balancing, among them UPVM [51], Amber [23], CHARM++ [45, 18], and a system supporting the Dataparallel C language [59]. UPVM supports dynamic load balancing for SPMD-style PVM applications via transparently migratable light-weight virtual processors (VPs). By allocating many more VPs than physical processors in use, the load can be adjusted by migrating VPs between machines, such that most machines run many VPs. VPs are thread-like and have less state than an operating system process, so migration is simplified. Amber supports load balancing with an object migration mechanism to control the location of data and processing. Consistency issues are simplified by using a function-shipping approach instead of the more common technique of moving data, although objects with data are also migrated when explicitly requested by the application.
Amber programs are written in an object-based subset of C++, supplemented with primitives for thread management and object mobility. The CHARM++ system also requires a certain programmer effort to be used. It migrates “object arrays” for load balancing, but for this the user must supply appropriate pack and unpack methods and use the CHARM++ library. Processes may communicate via a message passing paradigm. The Dataparallel C language supports only the restrictive SIMD programming model. The system implementing Dataparallel C adjusts load by automatically migrating “virtual processors” containing data and code, even among heterogeneous architectures.

Several systems offer load balancing on more tightly-coupled multiprocessor machines, e.g. CHAOS [72], COOL [21], and JOVE [75], but all of these systems again require the use of some proprietary library or language to achieve such functionality.

### 9.3.2 Parallel computing with semi-dynamic processor set

Some systems can vary at runtime the processor set computing a parallel problem, but they still require some resources on all potentially available machines, even while they are unavailable. These systems can therefore completely remove the compute load from an unavailable machine, as truly adaptively parallel systems can, but they still require such machines to remain connected, as load balancing systems do. Adaptive Multiblock PARTI (AMP) [31], Application Data Movement (ADM) [63], Abstractions for Adaptive Data Parallelism [64], and Millipede [42] belong to this category.

AMP runs SPMD programs adaptively by switching between active and “skeleton” process states according to the availability of processors. Adaptations can take place only outside data parallel loops, and although all processes of the application execute all parallel loops, a skeleton process simply has no more data and therefore does not perform meaningful computation. A programmer uses the AMP runtime library to recalculate loop bounds and to redistribute data at an adaptation.

Similarly, ADM also uses skeleton processes to later on achieve expansion onto new processors as they become available, and adaptations are again performed in-between data parallel iterations. ADM places much of the burden of how an application deals with an adaptation on the application programmer, but it can adapt by migrating data among heterogeneous workstations and includes load balancing by assigning different amounts of data to different machines.

Abstractions for Adaptive Data Parallelism is a system for data parallel programs that adapts by changing process states between awake and asleep and redistributing the data accordingly. The application programmer must supply an adapt routine which may only be called at one point in the code and causes global process synchronization, followed by a data redistribution.
Millipede offers load balancing and adaptivity with a transparent thread migration mechanism. An application is run with a predefined number of threads called workers, and at initialization time the system allocates a stack at identical virtual memory addresses for each thread on each machine, but any given thread is at most active on one of the machines, and all other copies are idle. Ownership of a machine is returned to some other user by setting all threads on that machine idle, although the stacks for all threads still remain allocated and the data remains available. Active threads execute jobs, so any incomplete jobs are migrated off a withdrawing machine, but the data is only transferred later by the DSM system, when access misses occur. Millipede programs are written in an extension of C, ParC or ParC++ [13, 11], such that the application is unaware of data and thread locations.

9.3.3 Parallel computing with dynamic processor set

In this subsection we review systems that genuinely have the ability to expand an on-going computation onto extra processors and to withdraw from processors in such a way that these can (probably) be disconnected from the computation. Piranha [35], Equus [49], Cilk-NOW [16], Calypso [10], Gardens [65], and several PVM-based systems, specifically MPVM [19], MpPVM [20] and DynamicPVM [28], belong to this category.

It appears that the Piranha authors were the first to use the term “adaptive parallelism” to denote such variable processor sets at runtime. Piranha provides adaptivity for programs using the Linda tuple space [34] by running low-priority daemons on any potentially participating machine, such that new application processes can be forked on joining machines and terminated again on withdrawing machines. However, the programmer needs to supply several appropriate routines which manage the adaptation, including the orderly transfer of any needed data from retreating machines.

Equus is also one of the earlier systems offering adaptively parallel computing. Equus is implemented at the operating system level and offers reconfigurable distributed computations (RDCs), machines may be added and withdrawn when all processes reach global synchronization points, and processes communicate with each other via special stream communication linkages offered by the Equus library. The Equus kernel running on all potentially participating machines supports the addition and withdrawal of compute processes, including process migration as an alternative form of withdrawal.

The Cilk-NOW system permits the adaptive and reliable parallel execution of programs expressed with a functional subset of the Cilk language. Machines may join and leave the computation anytime, under control of a node manager running as a
background daemon and monitoring a machine’s load. New nodes may “steal” work under the form of a closure from other nodes, and nodes may withdraw by migrating unfinished closures to some remaining node.

Calypso provides parallel computing with load balancing, adaptivity and fault tolerance. Programs are written in the CALYPSO Source Language (CSL). CSL is C++ with added constructs to express parallelism. Programs execute in a sequence of parallel steps, after each of which machines may be added and withdrawn and/or load is rebalanced. For each parallel step, the compiler creates a specified number of thread segments, then the runtime system dynamically allocates these segments to the machines participating in that step, so load balancing and adaptation is completely transparent to the application. Data is moved automatically using the shared memory paradigm, however, data is only shared at the beginning and end of a parallel step, so each segment operates like an independent task. Such a model simplifies fault tolerance, as any for any reason incomplete segment can simply be reexecuted by another processor.

Gardens is a programming language and system for adaptive parallel computing across NOWs, building on the concept of Active Messages (AM) [83], which is a form of lightweight asynchronous remote procedure call coupled with a synchronous (poll) accept mechanism. The ideas of AM are extended with mechanisms for task migration. A Gardens screensaver runs on every potentially participating machine, such that any idle machine can run a Gardens task, or a machine can be returned to another user by migrating tasks by copying their heap and stack to the same virtual memory location on another machine. One process may contain several such tasks, and it may end once all its tasks have been migrated off the local machine. Migration is permitted only when all tasks reach a “poll” point, which a programmer must explicitly insert in the application code.

The PVM-based systems MPVM, Migration-point based PVM (MpPVM) and DynamicPVM use various forms of migration to support adaptivity. They leave the number of processes unchanged at runtime, so adaptations are transparent to the user and application programmer and need not be dealt with in an application. These systems require a daemon process on machines considered as migration targets. MPVM is a user-level implementation running on various Unix flavors. To achieve this, not all Unix system calls are permitted in the application code. MpPVM supports process migration even in heterogeneous computing environments. MpPVM takes a high-level approach: Migration may occur only at migration points, inserted automatically by a precompiler, where a data analysis is performed and the minimum set of variables to be transferred is determined, reducing the amount of data moved. DynamicPVM contains process checkpointing, migration and restarting mechanisms based on Con-
9.3. ADAPTIVE PARALLEL COMPUTING

dor to move processes from one machine to another: Processes may be core-dumped to a shared file system and restarted elsewhere, but only when they are not inside “critical sections” such as when communication is taking place.

9.3.4 Adaptive parallel computing on multiprocessors

Various systems supporting adaptively parallel computations have also been devised for more tightly-coupled multiprocessor environments. Adaptive parallelism is easiest to achieve in hardware shared memory systems, as no data redistribution is required and the hardware automatically ensures the cache consistency of all processors. In addition, adaptations only cause minimal overhead due to the absence of data movement. In distributed memory systems however, the same issues apply as in NOW environments, but the dedicated networks of multiprocessor systems offer far superior latency and bandwidth compared to NOW networks, so adaptation costs are still orders of magnitude smaller.

In this thesis we focus primarily on NOW environments, where the performance obstacles for parallel computing and for adaptive parallelism are much higher than in multiprocessor environments. Therefore, as we can demonstrate that adaptive parallelism is attractive in NOW environments, it is even more so in multiprocessor environments, as far as performance is concerned. One difference however lies in the typical usage model, as multiprocessor systems normally do not have individual users for individual nodes. There, the predominant use of adaptive parallelism is as a scheduling tool to change the parallel jobs mix at runtime.

One example system implementing adaptive parallelism in a distributed memory multiprocessor environment is the Distributed Resource Management System (DRMS) [58] system. DRMS permits the reconfiguration of SPMD Fortran applications to a varying processor set at runtime, with the goal of increasing system utilization by possibly scheduling several parallel applications to run simultaneously. DRMS provides language extensions to Fortran, allowing an adaptation after each parallel step, with the application using special library calls to move and redistribute data. DRMS runs on an IBM SP2 machine, so the dedicated network allows for a fast data redistribution.

9.3.5 Comparative evaluation of adaptively parallel systems

Many issues must be considered when evaluating systems supporting adaptive parallel computing in NOW environments. Tables 9.1 and 9.2 provide a comparative overview of most of the adaptive systems outlined in the previous subsections, showing how each system fares for each of several characteristics. Table 9.1 contains systems that
do not let a parallel computation completely clear off a withdrawn machine, while Table 9.2 shows systems that do support true adaptive parallelism. For comparison purposes, our Adaptive TreadMarks system is also shown in both tables. The usability characteristics describe how a system can be practically deployed and therefore probably decide about its success or failure concerning widespread use, whereas the system characteristics are important mainly insofar as they have implications for the former. Below we list the usability characteristics that we deem to be the most relevant and state how they should ideally be fulfilled. These criteria are similar to those already mentioned in Section 1.4, although here we focus more on systems as a whole, not just on adaptive parallelism.

- What type of applications can be run?
  Ideally, a wide range of commonly-used parallel applications should be supported, instead of only select types such as programs with a very low communication-to-computation ratio.

- What programming paradigm and API is used?
  A widely-accepted industry standard is probably a prerequisite for the widespread acceptance and use of a system, and applications should not require reworking specifically to support adaptivity, or at most very straightforward modifications that can easily be made, such that existing applications can be run.

- Can the system truly employ a variable processor set at runtime?
  An individual user may wish to withdraw a machine from the parallel computation and reboot it with another operating system or run some other application locally requiring resources that are occupied by the parallel computation. Therefore, a withdrawal should remove all processes associated with the parallel computation.

- How often and how quick can adaptations be performed and is the system user-friendly?
  An adaptive system should always respond quickly (within a few seconds at most) to a withdrawal request, and users should easily be able to initiate adaptations.

- What is the system’s non-adaptive performance and what is the overhead for adaptations?
### Table 9.1: Systems for NOW-environment adaptive parallel computing that do not truly support a variable processor set at runtime, compared to Adaptive TreadMarks.

<table>
<thead>
<tr>
<th>System characteristics</th>
<th>Adaptive TreadMarks</th>
<th>UPVM</th>
<th>Millipede</th>
<th>Adaptive Multiblock PARTI (AMP)</th>
<th>Application Data Movement (ADM)</th>
<th>Abstractions for Adaptive Data Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable process or thread set at runtime</td>
<td>Yes</td>
<td>No</td>
<td>No, but switch threads' states between active and idle</td>
<td>No, but switch process' states between busy and idle</td>
<td>No, but switch process' states between awake and asleep</td>
<td></td>
</tr>
<tr>
<td>Adaptivity mechanism</td>
<td>Create and terminate processes</td>
<td>Migrate lightweight virtual processors</td>
<td>Migrate thread states to remote thread copies &amp; switch thread states</td>
<td>Move data and switch process states</td>
<td>Move data and switch process states</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usability characteristics</th>
<th>Parallel</th>
<th>Parallel</th>
<th>Parallel</th>
<th>Parallel</th>
<th>Parallel</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed or parallel processing</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td>SPMD data parallel</td>
<td>Data parallel</td>
<td>SPMD data parallel</td>
</tr>
<tr>
<td>Application types</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td>SPMD data parallel</td>
<td>Data parallel</td>
<td>SPMD data parallel</td>
</tr>
<tr>
<td>Programming paradigm</td>
<td>Shared memory</td>
<td>Message Passing</td>
<td>Shared memory</td>
<td>Message passing</td>
<td>Message passing</td>
<td>Message passing</td>
</tr>
<tr>
<td>API</td>
<td>OpenMP industry standard plus 1 routine f. task parallel progs.</td>
<td>PVM industry standard?</td>
<td>PVM &amp; PVM++</td>
<td>HPF and PVM plus AMP library</td>
<td>Very proprietary</td>
<td>MPI plus proprietary code</td>
</tr>
<tr>
<td>Non-adaptive performance</td>
<td>High</td>
<td>Low? (many virt. processors per processor)</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Variable processor set at runtime</td>
<td>Yes</td>
<td>No (cannot remove all load on a machine)</td>
<td>No (stack for each thread remains on each machine)</td>
<td>No (skeleton process needed on withdrawn machine)</td>
<td>No (skeleton process needed on withdrawn machine)</td>
<td>No (skeleton process needed on withdrawn machine)</td>
</tr>
<tr>
<td>Transparent adaptation</td>
<td>Yes</td>
<td>Yes?</td>
<td>Yes</td>
<td>Application uses explicit AMP calls</td>
<td>Application uses proprietary code for adaptivity</td>
<td>Application uses proprietary code for adaptivity</td>
</tr>
<tr>
<td>Anytime, quick withdrawal?</td>
<td>At any OpenMP join-fork / at task boundaries</td>
<td>Only rapid load reduction possible</td>
<td>Yes, with a few exceptions</td>
<td>Outside data parallel loops</td>
<td>Whenever adapt routine is called</td>
<td>Whenever adapt routine is called</td>
</tr>
<tr>
<td>Overhead of adaptivity</td>
<td>Low</td>
<td>Medium (cost of VP migration)</td>
<td>Low?</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Portability, implementation level</td>
<td>User-level</td>
<td>User-level</td>
<td>User-level</td>
<td>User-level</td>
<td>User-level</td>
<td>User-level</td>
</tr>
<tr>
<td>Supports heterogeneous NOW</td>
<td>No</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load balancing</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Attractive speedups and a modest runtime increase when adapting are required, otherwise computations will rather be performed sequentially and not with parallel programs.

- Can the system easily be deployed and utilized in common NOW environments?
  A system should not require modifications of existing environments and setups, so that it can easily be installed and used.

Tables 9.1 and 9.2 summarize our observation that many other systems have already explored various subsets of all the listed aspects. The main novel aspect of our work is therefore the combination and fulfillment of all the criteria of the above list.
<table>
<thead>
<tr>
<th>System characteristics</th>
<th>Adaptive TreadMarks</th>
<th>Pirhana</th>
<th>Equus</th>
<th>Cilk-NOW</th>
<th>Calypso</th>
<th>Gardens</th>
<th>MPVM</th>
<th>MpPVM</th>
<th>DynamicPVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable process or thread set at runtime</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Adaptivity mechanism</td>
<td>Create and terminate processes</td>
<td>Create and terminate processes</td>
<td>Create and terminate or migrate processes</td>
<td>Create and terminate processes</td>
<td>Migrate lightweight tasks and create and end processes</td>
<td>Migrate processes</td>
<td>High-level process migration</td>
<td>Migrate processes via core dumping</td>
<td></td>
</tr>
<tr>
<td>Usability characteristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distributed or parallel processing</td>
<td>Parallel</td>
<td>Distributed and coarse-grained parallel</td>
<td>Distributed or very coarse-grained parallel</td>
<td>Distributed</td>
<td>Coarse-grained parallel</td>
<td>Distributed and coarse-grained parallel</td>
<td>Parallel</td>
<td>Parallel</td>
<td>Parallel</td>
</tr>
<tr>
<td>Application types</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td>Distributed computations</td>
<td>Functional programs</td>
<td>Data parallel</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td>Data parallel, Task parallel</td>
<td></td>
</tr>
<tr>
<td>Programming paradigm</td>
<td>Shared memory</td>
<td>Linda tuple space</td>
<td>Message passing</td>
<td>Closures</td>
<td>Shared memory</td>
<td>Active Messages</td>
<td>Message passing</td>
<td>Message passing</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>OpenMP industry standard plus 1 routine f task parallel progs.</td>
<td>Linda plus application-specific code to support adaptivity</td>
<td>Proprietary Equus library</td>
<td>Functional subset of Cilk language</td>
<td>Calypso Source Language</td>
<td>Gardens proprietary API</td>
<td>PVM industry standard</td>
<td>PVM industry standard</td>
<td></td>
</tr>
<tr>
<td>Non-adaptive performance</td>
<td>High</td>
<td>Inefficient</td>
<td>High?</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>Variable processor set at runtime</td>
<td>Yes</td>
<td>Yes?</td>
<td>Yes? (Equus kernel remains on all machines)</td>
<td>Yes? (daemon remains on machine)</td>
<td>Yes? (daemon remains on machine)</td>
<td>Yes? (daemon remains on machine)</td>
<td>Yes? (daemon remains on machine)</td>
<td>Yes? (daemon remains on machine)</td>
<td></td>
</tr>
<tr>
<td>Transparent adaptation</td>
<td>Yes</td>
<td>Appl. uses proprietary code f. adaptivity</td>
<td>Application must use Equus calls</td>
<td>Yes? (daemon remains on machine)</td>
<td>Yes</td>
<td>Application uses Gardens calls</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Anytime, quick withdrawal?</td>
<td>At any OpenMP join-fork / at task boundaries</td>
<td>Yes, after executing retreat actions</td>
<td>At global synchronization points only</td>
<td>Yes? (with lag time)</td>
<td>After each parallel step</td>
<td>When all active tasks arrive at poll points</td>
<td>Yes?</td>
<td>At migration points</td>
<td>Yes, except inside critical sections</td>
</tr>
<tr>
<td>Overhead of adaptivity</td>
<td>Low</td>
<td>Low?</td>
<td>Medium? (process image transfers)</td>
<td>Low? (only needed data is transferred)</td>
<td>Medium? (process image transfers)</td>
<td>Low? (only needed data is transferred)</td>
<td>Low? (only needed data is transferred)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Supports heterogeneous NOW</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Load balancing</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.2: Systems for NOW-environment adaptive parallel computing that support a variable processor set at runtime, compared to Adaptive TreadMarks.
Chapter 10

Discussion and conclusions

The goal of this dissertation was to assess how adaptive parallelism can be achieved in distributed memory environments in a practical and attractive-to-use way. A crucial element on the path to success was the adoption of the shared memory paradigm, which alone permits the fully automatic data distribution and redistribution after every adaptation, thus avoiding the need to resort to proprietary libraries common to other adaptive systems. No need for such routines means no need or at best very little need to change application source code for the support of adaptivity: We show how many data parallel applications using the industry standard OpenMP API can be run adaptively without requiring any code changes for the purpose of adaptivity and how task parallel applications require only the insertion of one extra routine call at task boundaries for the same purpose, a code modification that can often be performed automatically by a preprocessor. Together, these two classes of applications demonstrate that a broad range of applications can benefit from adaptive parallelism. The thesis uses a software distributed shared memory system to show that their performance is attractive both when adapting and without adaptation: Running the system on a standard NOW with up to 16 dual-processor machines, we conclude that the extra overhead caused by adaptivity is very low, increasing an application’s runtime by at most a few percent even for very frequent rates of adaptation. The thesis further shows how we have implemented the support for adaptive parallelism.

We believe this thesis successfully shows that adaptive parallelism is a very attractive proposition in distributed memory environments, especially but not exclusively in multi-user settings, where adaptively-run parallel computations need hardly affect other users.

Our system has demonstrated that many programs can still achieve good speedups despite frequent adaptations. The value of adaptive parallelism becomes evident when considering the following points: (1) Adaptive parallelism is a prerequisite for parallel computations in (1a) a multi-user environment where individual users may desire
unrestricted access to machines without any prior reservations, or in (1b) environments where on-going computations need to vacate some resources required by some other program of higher priority. (2) Adaptive parallelism often significantly increases performance and throughput, as it lets parallel computations run on a much higher average number of machines. Without adaptivity, the same problem might be run in parallel only on a small set of machines, or it might be solved sequentially or even not at all, depending on how many machines are exclusively reservable or available throughout the computation. (3) Adaptive parallelism comes at hardly any extra cost, as already-available hardware and operating systems are used and other uses of the machines in question need not be diminished, they can still be handled with priority.

When evaluating the performance, one may realize that software DSM systems cannot quite match dedicated hardware shared memory systems, but this comparison is of lesser interest. Rather, the extra speedup achieved thanks to adaptive parallelism is the main issue. Even e.g. a speedup of 4 on an average of 8 machines, which most of the programs we tested easily beat, is attractive when the alternative is e.g. a sequential execution.

Parallel processing on NOWs may permit much larger problems to be solved than sequential processing, as the computation’s physical memory requirements are effectively distributed among the participating machines, as each process only accesses a part of all the data. Inasmuch as adaptive parallelism increases the average and the minimum number of available machines used by a computation, larger problem set sizes can still be computed without suffering from excessive paging effects. Further, the larger the data set, the longer a computation usually runs for a given application, so adaptive parallelism becomes especially important: The time an individual workstation is present in the pool of compute servers does not bound such lengthy program executions.

With the design and implementation of modifications of a real system in use, this thesis demonstrates that adaptive parallelism is attractive and feasible even without requiring a new memory consistency protocol. A judicious use of fork-join semantics and garbage collection mechanisms circumvents the need for the memory consistency protocol to support adaptivity.

10.1 Future work

Many avenues exist for future research work and enhancements based upon our work. Below we list a selection we estimate as promising.
**Load balancing.** The current adaptively parallel system could be integrated with the load balancing capabilities for data parallel applications offered by a different augmentation of TreadMarks [41], as we have seen in Section 9.3.1. In our system each parallel loop's iterations are partitioned among processes according to process' ids and the current total number of processes, such that all processes simply receive an equal share of iterations. This scheme could easily be modified to accommodate the current relative processing power of each process as a third factor. The relative processing power would reflect the current load on the corresponding processor, so the loop iterations would be allocated according to each process’ share of the total processing power available.

Actually, the current system can already perform coarse-grain load balancing when the number of processes exceeds the number of processors in use. Performing adaptations such that the number of processes on a given processor increases or decreases without completely vacating a processor has the same effect as load balancing.

**Fault tolerance.** For longer-running computations fault tolerance becomes increasingly important, and many possible solutions are conceivable. We have described one conceptually simple approach in Section 4.2.2. The lazy release consistency protocol itself also provides new opportunities for a fault tolerant DSM implementation, as mentioned in Section 6.2.2 of the LRC thesis [46].

**Prefetching and other optimizations.** Many opportunities for optimizations exist. Let us point out improvements that have been implemented for TreadMarks but are not currently part of our system [30, 55]: The compiler is used to compute data access patterns and then automatically performs source-to-source code transformations with new run-time system calls that aggregate data and communication, among others. Further, these code modifications permit the prefetching of data, significantly reducing access faults. Such techniques are employed for regular and irregular applications.

Other issues that we have not analyzed in more detail might include improved data placement strategies that take into account the communication patterns, and other process id allocation strategies at an adaptation.

**Compiler-controlled frequency of adaptation points.** Currently, adaptation points are predefined by parallel sections or task boundaries. However, in data parallel programs the compiler may be used to automatically insert extra adaptation points by performing strip mining- or loop tiling-type transformations. This technique can ensure a desired degree of responsiveness of the system to adaptation requests, espe-
cially leave requests, for cases where regular adaptation points can sometimes not be reached within a given time frame after a request is detected.

The transformations can be done in such a way that during normal execution the system “ignores” the global synchronization associated with the extra adaptation points, i.e. the synchronization operations are not performed unless an adaptation is pending, so the run-time overhead is minimal. The essential point is that the transformed application code contains the provision to adapt frequently enough, even if the extra code is only executed when an adaptation is actually requested.

Application programmers normally intuitively expect a sequential consistency-like execution model. For this, the correctness of programs is not affected by such (potentially) excessive global synchronization.

 Adaptive use of selected resources. We have so far only investigated adaptivity by creating and terminating standard Unix and Linux processes. Current operating systems often offer relatively little control over the resource usage of individual processes.

 Adaptive parallelism may however be employed beneficially in modified run-time environments where resources such as CPU time (for SMP machines on a per-CPU basis), physical memory, disk I/O, network usage, etc. can be explicitly controlled. This would permit new usage models and a better user acceptance: Strict limits on how much interference a local user will ever experience by processes of the adaptively-parallel computations could be imposed, such that a user need not fear that his workstation might be “hijacked” by some compute-intensive foreign processes, so he would more likely grant usage of his workstation. Different programs could easily be scheduled on the same workstation, as each program’s (set of) process(es) could be guaranteed a specifiable degree of access to resources. Such a technique could also alleviate the need for an urgent leave of a compute process, as a user would tolerate a defined disturbance level for some time, and the process could leave at a later opportune time.

 Policies can be implemented for one or a specifiable pool of processes to adhere to such limits while exploiting extra resources on the local machine as long as they are idle. Exactly such techniques are explored in the Active Harmony research effort [66] and in the “Performance Isolation” scheme for SMP systems [82], and even standard IRIX 6.5 offers a feature in this direction, Miser [73]. Active Harmony implements extensions to the Linux kernel, while the “Performance Isolation” model is realized by changing an IRIX 5.3 kernel. Miser in IRIX 6.5 lets batch jobs reserve a set of resources, including logical CPUs and physical memory.

 As the above examples show, support for such mechanisms and policies typically
requires operating system changes, and they may or may not require changes at the user level, e.g. inside a system such as our Adaptive TreadMarks, and/or in the applications, or only a separate controlling system may be required.

10.2 Trends and visions

We expect adaptive parallelism capabilities to become an integral part of many parallel computing systems used in the future. Even if it is not exactly our system that gains widespread use in future, we envision similar concepts to be incorporated into other future systems, e.g., other DSM systems that are perhaps object-based rather than page-based, or new implementations that provide even more efficient network communication.

NOWs and high-bandwidth networks are becoming ever more ubiquitous. The processing power of workstations is advancing so rapidly and so much that individual users very often do not require the full potential of their machines anymore for most of the time, so workstations are becoming more and more capable of running some compute-intensive process(es) of some parallel computation without a user noticing a reduction in availability – especially when resource use limitation techniques as discussed in the previous section are employed – and this trend likely will continue in the future. Adaptive parallelism can therefore become much more popular in future.

The emergence of new distributed computing efforts even across the internet also shows that the untapped potential of many mostly-idle machines is being realized more and more.

Static, data parallel scientific code has so far been the most common type of parallel application, and presumably this type will remain important in the future. This prediction also underscores the relevance of our work, as such applications are ideally-suited for adaptive parallelism thanks to their inherent use of global synchronization.

One of the main limiting factors for parallel computing performance on NOWs will for the foreseeable future probably remain network latency. This limit is usually more severe for non-adaptive parallel computing than for adaptation costs, as the data redistribution at adaptations usually affords far more opportunities of data aggregation than during normal computation. Therefore, any parallel application achieving good speedups in the non-adaptive case most likely will not be limited by any high adaptation costs.

The rapidly increasing amount of physical memory installed in common workstations is permitting ever larger data sets to be used efficiently in parallel computations on NOWs. As network bandwidths are similarly growing, adaptation costs will remain low, even without much improvement in network latencies: Exploiting such
higher network capacities – by aggregating more data in one message, by increasing page sizes, and with other measures – will permit very attractive adaptation costs even for data sets many times larger than the ones we have tested.

As NOWs become larger and DSM systems may scale up in future, parallel computations may run with more and more processes, so adaptivity correspondingly will become more of a necessity. The more machines are involved, the shorter the average interval between a change in the pool of available machines, for a given frequency of adapt events per machine. As the overall adaptation event frequency of a computation thus increases with an increase in the number of processors involved, the adaptation overhead still remains at an approximately unchanged level: We have shown how the cost per adaptation decreases roughly proportionally with an increase in the number of processes.

We envision NOW settings where individual users utilize a tool that enables them to easily specify when their machine(s) is/are to be available for parallel computing and perhaps also to specify how the availability shall be, i.e. which resources are considered and to what extent. The default setting should be to include each machine’s resources to a certain degree that still guarantees little interference with typical local user tasks, but the tools should enable each user to override these settings to his desire. Adaptively parallel computations should thus automatically make use of all machines designated as available.

We further envision adaptive parallelism to be used more on Beowulf-type clusters [1] to better schedule various parallel applications in parallel.

In summary, we believe that adaptive parallelism in DSM environments has by far not reached its full potential yet and will therefore experience much growth in future.
Bibliography


Curriculum Vitae

February 3, 1968  Born in Muscat, Oman
1973          English school, Melbourne, Australia
1973 – 1977  2 English schools, near Lisbon, Portugal
1977 – 1979  English school, Miri, Sarawak, Malaysia
1979 – 1986  German school, The Hague, The Netherlands
1986 – 1987  German school, Caracas, Venezuela
1987          Abitur exam
1987          Auxiliary exam to obtain Eidg. Maturität
1987 – 1988  Computer science studies at EPF Lausanne
1988 – 1990  Economics studies at HSG
1990 – 1994  Computer science studies at ETH Zurich
1991          Computer science internship at ZKB, Zurich
1992          Computer science internship at ABB Informatik AG, Baden
1994          Diploma in Computer Science, ETH Zurich
1994 – 1995  Research and teaching assistant
in the research group of Prof. B. Sanders,
Institute for Computer Systems, ETH Zurich
1995 – 2001  Research and teaching assistant
in the research group of Prof. T. Gross,
Institute for Computer Systems, ETH Zurich