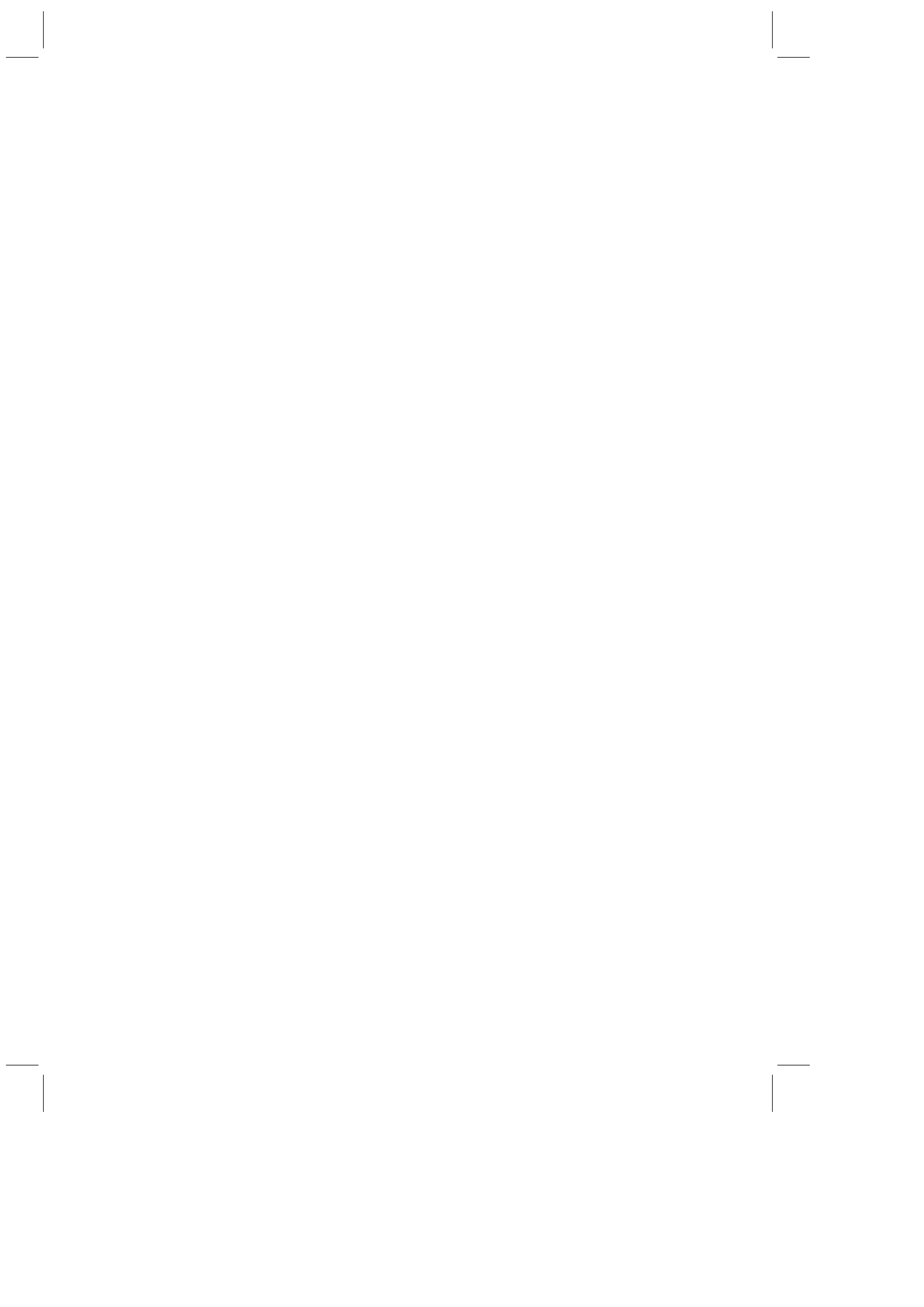
or

PROGRAMMING MULTI-CORE AND MANY-CORE COMPUTING SYSTEMS



PROGRAMMING MULTI-CORE AND MANY-CORE COMPUTING SYSTEMS

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright ©year by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created ore extended by sales representatives or written sales materials. The advice and strategies contained herin may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

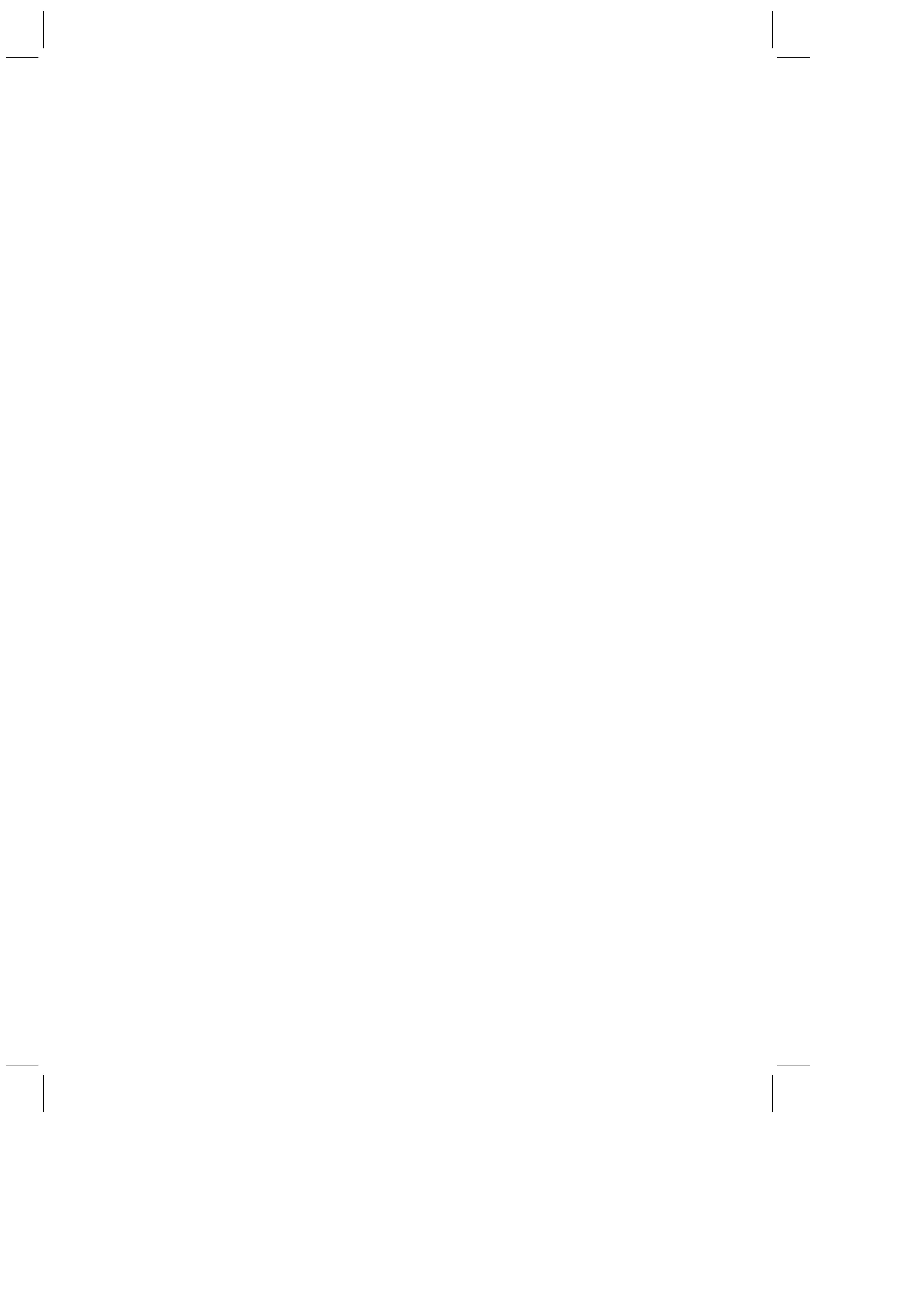
For general information on our other products and services please contact our Customer Care Department with the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

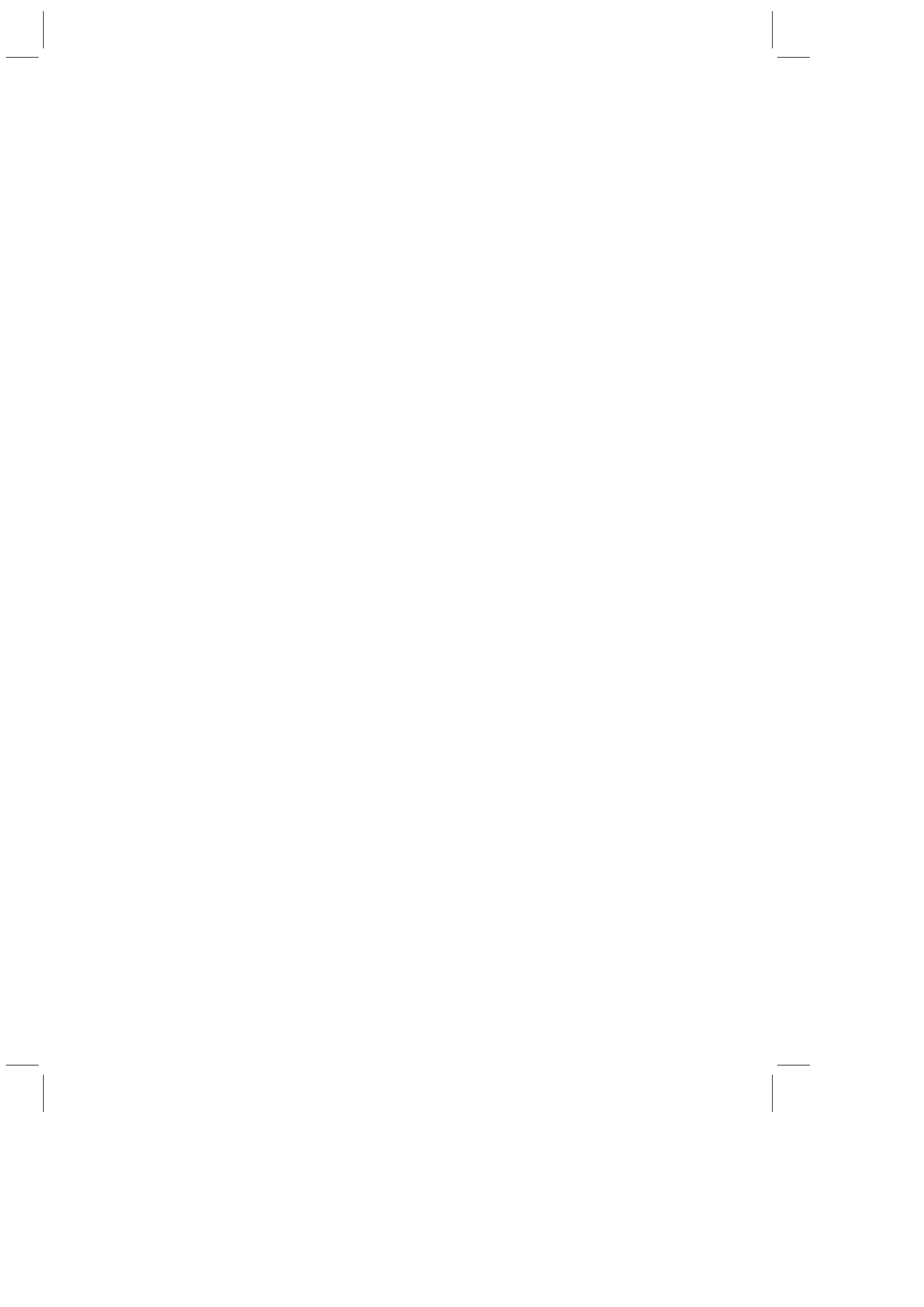
Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic format.

Library of Congress Cataloging-in-Publication Data:

Title, etc
Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

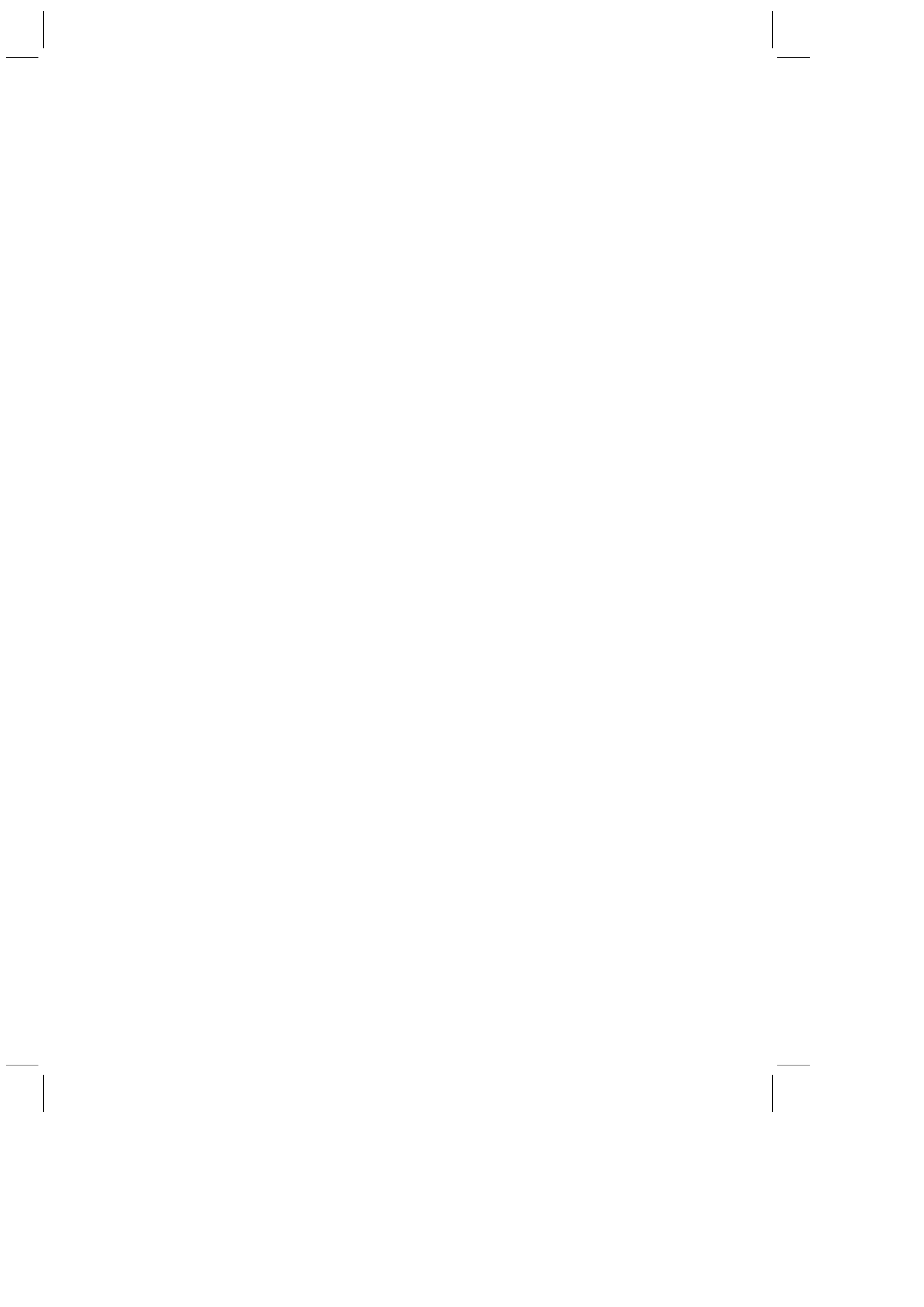




CONTENTS

1 Autonomic Distribution and Adaptation 1
Lutz Schubert, Stefan Wesner, Daniel Rubio Bonilla and Tommaso Cucinotta

1.1 Introduction 1
1.2 Parallel Programming Models 2
1.2.1 Explicit Communication and Synchronisation 2
1.2.2 Implicit Communication 4
1.2.3 Automated Parallelisation 5
1.3 Concurrent Code 6
1.3.1 Concurrency Analysis and Exploitation 7
1.3.2 Mapping and Adaptation 11
1.4 Conclusions 12



CHAPTER 1

AUTONOMIC DISTRIBUTION AND ADAPTATION

Lutz Schubert[1], Stefan Wesner[1], Daniel Rubio Bonilla [1] and Tommaso Cucinotta[2]

[1]HLRS - University of Stuttgart, Stuttgart, Germany
[2]Real-Time Systems Laboratory, Scuola Superiore Sant'Anna, Pisa, Italy

1.1 INTRODUCTION

It has been noted multiple times in this book how the future development trend of processor architecture goes towards heterogeneous mixed many- multi-core systems, such as already demonstrated by IBM's CELL processor[1] or the OMAP5 by Texas Instruments [2]. It is thereby also obvious that future software (and implicitly software developers) has to exploit parallelism in order to improve the efficiency of execution, or even just to enable additional features and functions. The main problem however consists in the complexity of the according programming models and the degree of knowledge required about program behaviour for its effective parallelisation. In order to execute tasks and functions in parallel, their dependencies have to be identified and

[1]`http://www.research.ibm.com/cell/`
[2]`http://www.ti.com/ww/en/omap/omap5/omap5-platform.html`

title, edition. By author

Copyright © 2011 John Wiley & Sons, Inc.

work or data segmented in a fashion that improves rather than reduces the overall execution performance.

Most programming models originate however from the purely sequential computing area and offer little support for writing parallelised applications. With the rise of High Performance Computing (HPC) and hence heavily parallel environments, these classical models were extended with features to offer some support for parallelism that mostly consist of explicit or implicit means for controlling communication. As the HPC domain has so far been a restricted usage area, compared to common day-to-day programming and application development, developers in this domain typically pursue(d) a very specific interest and therefore spend the time and effort on learning dedicated programming extensions, and in particular on parallelising their code and taking additional precautions to achieve maximum performance. With the entry of parallelism into the desktop domain, this additional effort is however no longer justifiable and more usable models are required in order to enable "mainstream parallelism".

This chapter describes an approach for increasing the scalability of applications by exploiting inherent concurrency in order to parallelize and distribute the code. We thereby focus in specifically on concurrency in the sense of reduce dependencies between logical parts of an application. Concurrency forms a crucial part in any parallelisation approach, as the degree of dependencies across potential threads defines the delay due to messaging and synchronisation overhead. For example, loop unrollments show best performance improvement if they are highly concurrent and thus vectorizable.

What is even more important, though, is the fact that concurrency can be exploited for parallel execution of *sequential* (i.e. unparallelizable) code logic. In other words, if multiple, independent sequential segments can be identified, they can be executed in parallel to each other. Thus concurrency exploitationg directly affects the limiting factor of Amdahl's [?]. We will show how graph analysis methods can be employed to assess the dependencies on code level, so as to identify concurrent segments and to relte them to the specific charateristics of the (heterogeneous, large-scale) environment.

1.2 PARALLEL PROGRAMMING MODELS

As the need for parallel applications increases, so does the demand for efficient and yet easy to use programming models and languages that enable scalable and - in the long run - portable behaviour over heterogeneous infrastructures. In the following, we will provide an overview over some of the existing parallel programming models and their strengths, respectively weaknesses with respect to these specific goals:

1.2.1 Explicit Communication and Synchronisation

The most classical approach to parallelisation consists in enabling the development (and parallel execution) of "threads", e.g. [?]. Like processes, threads are effectively

nothing else but independent applications that however can identify each other in order to exchange data through dedicated communications. As the description indicates, the classical thread- enhancements do not offer an explicit means for data synchronisation or sharing. In other words, the developer has to identify communication points in his program and explicitly specify the data to be transmitted - in order for one thread to share data with a second thread, it is therefore necessary for the first thread to explicitly select and send the respective data set to the second thread which in turn will have to wait for reception of this data. In the case of explicitly shared data, this means furthermore that the according dataset will have to be send back and forth in order to maintain consistent state. Accordingly it is easy to introduce synchronisation and hence performance issues.

It can therefore be generally noted that thread-based programming models are only efficient given the right expertise of the developer and a use case where data needs only be shared at clear, discreet points in time. If the according knowledge is missing, or should the synchronisation points not be obvious enough, this approach can not only lead to significant performance losses, but also to locking and unpredictable behaviour. Many parallelisation methods can raise this condition, such as write before read across iterations in a parallelised loop. What is more, the approach as such does not support addressing the heterogeneity issue of computing systems, which not only affect how these threads need to be compiled, but in particular leads to deviations in the synchronisation behaviour, if e.g. the execution speed between resources deviates from one another.

With the introduction of the **Message Passing Interface**[3] (MPI), an attempt was made to standardise the communication between threads (and processes) in order to principally allow message-based data synchronisation across infrastructures. Using MPI it is thus possible to execute different threads and / or processes in different environments and nonetheless communicate with each other, as long as all involved systems share information about the thread-IDs. Since MPI promotes a specification and therefore a general strategy, rather than an explicit execution model or framework, it could be easily integrated into existing programming models and compiler models.

MPI provides all the essential capabilities needed to deal with large scale and heterogeneous infrastructure. However, it's efficiency depends almost completely on the capabilities of the developer. Furthermore, MPI was developed for multi-processor systems with an explicit communication framework between these units - in other words, MPI does not cater for indirect communication or shared memory systems. Most modern multi-core processors however build on some form of hardware based cache coherency (ccNUMA) for which MPI is not suitable or at least generates unnecessary overhead. Even though future many-core systems will most likely not share memory across all processing units, we must nonetheless assume, that at least some cores will share memory across or at least grant remote access to this memory [**?**].

As opposed to real distributed systems, where each processing unit / node contains its own full environment, machines in which multiple units share a common memory

[3] see `http://www.mpi-forum.org/`

- either physically, or through a hardware based protocol - are comparatively easy to program for. This is simply due to the fact that the developer does not explicitly have to cater for sharing and distributing state related data, but instead can assume "global" state across all threads, so that all processes can handle the data as if local. The simplest example to parallelising work in a shared-memory environment therefore consists in loop unrollments where the individual iterations of a loop are executed in parallel. Notably, the loops may not have any dependencies across iterations, i.e. no access to previous values n[i] = f(n[i-1]) as this will cause conflicts if n[i], n[i+1], ... n[i+j] are calculated in parallel. The **Open Multi-Processing** [4] API (OpenMP) is the classical programming extension to develop such shared-memory applications.

However, OpenMP does not cater for heterogeneous architectures as they actually may arise in future multi-core systems, even if it is not necessarily to be expected that cores of different types will not directly share cache or memory. As different cores will have different execution performance with respect to the same tasks, even in a straight-foward work segmentation, such as in the aforementioned loop-unrollment, the individual processes may deviate, leading to similar locking and read-write issues as in the thread-based case.

As noted, future processors will not solely rely on shared or cache coherent memory architectures, as these approaches do not scale to the degree needed and cause performance loss due to the maintenance overhead. Instead, the microarchitecture will have to rely on combined models of shared memory tiles connected with other tiles over a network-on-chip communication infrastructure - in principle very similar to modern days' cluster architectures that effectively integrate a large amount of processors with multiple cores in a high-performance network environment. Accordingly, most modern day HPC developers already employ a mixture of OpenMP and MPI programming models to realise large scale applications that scale across the heterogeneous hierarchical infrastructure.

Obviously, this makes usage just more complicated for the average developer. What is more, the heterogeneity of future systems is expected to increase even beyond the point where it can still be handled with this approach.

1.2.2 Implicit Communication

Rather than having the user / developer deal directly with the specifics of the hardware, most modern programming models approach the problem by abstracting the system and having the middleware, respectively the compiler deal with the actual architectural details: for example, the **Partitioned Global Address Space**[5] (PGAS) model builds on the simplicity of programming shared memory machines and therefore exposes capabilities of (virtually) shared memory spaces that the API converts into message calls or actual shared memory usage, basing on the specifics of the archtiecture. However, this requires that the according infrastructure information is provided to the compiler - as we will discuss below, this is currently not generally pos-

[4] `http://openmp.org/wp/`
[5] `http://www.pgas-forum.org/`

sible and compilers will instead base on generic assumptions about the infrastructure characteristics.

Even though the PGAS approach has the big advantage of being comparatively easy to use, in particular for more skilled developers, most actual implementations of PGAS still suffer from performance issues. This is due to the fact that the compiler effectively still converts the shared memory access requests into a set of message based transactions, non-regarding the infrastructure. As noted, currently hardware descriptions are not used for the purposes of steering compilation though. Without this, even the PGAS model cannot avoid running into similar problems as the OpenMP and MPI combined approach, that is the inability to handle the large scope of heterogeneity we are about to face in day-to-day development.

It should not be disregarded thereby that even shared memory programming is still too complicated for many developers and in particular is not applicable to all development and use cases. In fact, most applications do not even execute complex algorithms that would benefit from the shared memory approach, respectively require complete rethinking on the developer's side. Accordingly, many manufactures pursue a more user-centric approach which essentially tries to take over parallelisation tasks for the developer. However, optimal parallelisation actually belongs to the class of NP complete problems [**?**], so that the "automagic" parallelisation can (and should) not be expected. On the other hand, sub-optimal parallelisation still can improve performance of common code and thus provide an acceptable solution for the average developer.

1.2.3 Automated Parallelisation

The main goal of modern programming models consists in simplifying usage of the increasingly complex modern infrastructures - in particular in order to overcome the problems of scale and heterogeneity. As it cannot be expected that developers deal with all types of future infrastructures by themselves, the programming model has to abstract from the hardware and still make best use of it in terms of performance - most current approaches thereby base on some form of virtualisation technique in order to hide the infrastructure complexity.

The general principle behind these approaches consists in identifiyng algorithmic patterns which indicate parallelisable functions, such as loops, queries etc. Similarly, some libraries offer functionalities which are implemented in a parallel fashion, thereby replacing the sequential implementation as provided by the standard extensions. The latter approach is particularly popular for mathematical libraries in HPC environments. The main problem with both approaches however consists in potential errors introduced through parallelising an otherwise sequential invocation, e.g. by neglecting time-dependent read-write operation on a specific memory space. To reduce this problem, almost all models require the developer to provide additional information or, more frequently, to explicitly invoke parallel versions of the according functionalities, such as "Parallel.For". The parallel .NET extensions for example provide a series of parallel database queries and functions as part of the LINQ instruction set.

This approach essentially leaves all performance related decisions to the developer, i.e. whether to use a parallel or sequential implementation of a specific functionality. Unexperienced developers will for example often select a parallel loop even for simple computational tasks, thus creating overhead for thread instantiation, distribution and communication that reduces performances below the pure sequential execution. In large scale systems, such a degradation of performance can easily arise due to the high communication overhead introduced by wide distributions of code. What is more, due to the nature of this approach, only specific parts of the code can be parallelised in the first instance, leaving many parts of the code sequential. The performance gain therefore strongly depends on the type of algorithms to be executed and the expertise of the developer.

1.3 CONCURRENT CODE

The keyword for further parallelisation, in particular of common work tasks that do not adhere to the typical parallel patterns above is therefore "concurrency". It also determines whether a loop or a pattern can be effectively executed in parallel in the first instance:

Concurrency in this specific context reflects the dependencies of a given code segment or function on other functions or parts of the code. The higher the degree of concurrency, i.e. the less dependencies exist, the more effective is its parallel execution and the less likely delays occur due to synchronisation overhead. In the ideal case, such as in embarrassingly parallel tasks, the concurrency reaches a maximum that implies that there are virtually no dependencies between the processes.

Obviously, a high degree of concurrency does not necessarily imply that the according segment can be executed in full parallel. A single shared variable can stall the full execution, if the seemingly concurrent code has to wait for the first thread to finish its calculation before the variable is free for access. At the same time, this obviously depends on the read-write order of the respective segments. Accordingly, and as discussed in more detail below, it is difficult to automatically identify concurrency in a given code efficiently. More realistic approaches, such as the **Star Superscalar** programming model [**?**] therefore require the developer to explicitly annotate data dependencies across their code and functions. This information can then be exploited by the compiler to generate a dependency graph which provides implicit information about the execution order and potential points for parallelisation and task distribution.

Star superscalar is thereby still very coarse granular and expects specific function calls to exhibit concurrency, rather than e.g. direct work load in a loop. It furthermore does not assess the execution speed of individual function blocks, so that resources may not be used to their full optimum - nontheless the model provides an easy method to increase the overall execution performance.

Essentially, even classical parallelisation measurements base on the principle of maximising concurrency between threasd, so as to minimise dependencies and thus communication and synchronisation overhead. Concurrency identification can therefore be regarded as the key factor in (semi-)automated parallelisation and in addressing

the requirements for future programming models. As indicated, however, concurrency cannot be reliably identified automatically:

1.3.1 Concurrency Analysis and Exploitation

There is an extensive literature on automated parallelisation which deals with multiple aspects of concurrency analysis in order to identify dependencies. In general, the stronger such a dependency, the less parallelisable the according function. However this furthermore depends on the sequence of read-write statements in the code and on frequency of such occurences etc. As a rule of thumb, the gain achieved through the concurrent execution must be higher than the loss introduced this way. Whilst this may sound trivial, it has multiple implications:

The major performance loss occurs by latency introduced through any delays - the most obvious are (1) waiting for data to become available and (2) passing it to the respective thread, respectively returning results. Similarly, additional delay arises through access to shared memory spaces. Less obvious however is the fact that many implicit operations will cause additional delays - this ranges from the overhead for creating the thread to executing system calls. In the first case, additional operations need to be executed in order to perform a seemingly simple task - this however involves a high degree of additional message passing. In the latter case, the major reason for delay is not so much the communication overhead, but the fact that in all setups limited resources exist. This includes not only exclusive devices (such as hard drive or keyboad), but also the operating system - most modern OS architectures are monolithic and hence centralistic in nature (see e.g. [**?**]). System calls will build up with the increasing number of threads and processes being executed concurrently, thus affecting the scalability of the operating system drastically.

A particularly relevant limited resource is the underlying network itself: not only does it introduce physical limitations in term of bandwidth and latency, but more importantly, it will be used by multiple processes at the same time, thus leading to further reduction of the bandwidth and implicitly to further delays. ccNUMA architectures particularly suffer from this reduction of bandwidth introduced by the consistency maintenance tasks of the cache coherency protocol. In other words, concurrency analysis must not only respect the dependencies within the code, but also across the infrastructure and in order to achieve efficient execution, this system information must be fed back to the mechanisms for thread distribution and instantiation:

The most general approach to identifying concurrency in a given code consists in analysing variable usage throughout the code logic and all its invocations. If two segments share a parameter, they become co-dependent according to the type of actions executed on the variable (i.e. read or write actions). The main problems consist obviously in reassigning the same variable name in different contexts, respectively in passing the content to other variables. Similarly, we need to distinguish between global and local usage scope, as well as between references and copied instances.

Most strategies focus less on individual variables, as their impact is comparatively low, rather than larger data or address spaces, i.e. memory ranges. In most programs, they are represented as arrays over which the algorithm acts. Arrays and in particular

indexes of arrays are thus the primary interest of most concurrency analysis mechanism. The principle itself is straight-forward: depending on the access pattern, and in particular the index relationships across iterations, specific parallelisation techniques can be employed. For example

- No dependencies:
  S: A(i) = A(i) + C(i)
  T: B(i) = B(i) - C(i)
- True dependency (same iteration):
  S: A(i) = A(i) + C(i)
  T: B(i) = B(i) - C(i) + A(i)
- True dependency (with previous iteration):
  S: A(i) = A(i) + C(i)
  T: B(i) = B(i) - C(i) + A(i-d)
- Antidependence (WAR):
  S: A(i) = A(i) + C(i) - B(i)
  T: B(i) = B(i) + C(i)
- Antidependence (WAR) (with increased index)
  S: A(i) = A(i) + C(i)
  T: B(i) = B(i) - C(i) + A(i+d)

Obviously this approach concentrates on concurrency in loops rather than general occurences of concurrent segments. The principle nonetheless may also be applied across different logical segments, given that the parameters, i.e. the array, in question can be uniquely identified.

Figure 1.1 A dependency graph derived from code behaviour analysis. Edges on the left denote data-flow and on the right work-flow (simplified).

This **source code level analysis** however neglects two crucial aspects: (1) most code behaviour depends on the data, i.e. the concurrency may alter given a specific

data set, and (2) the execution speed and actual memory usage of the code cannot be assessed correctly, so that potential synchronisation issues cannot be detected, unless the concurrent segments are essentially uniform, as is the case in loop unrollment.

What is more, the analysis is generally restricted to the source code at hand, leaving aside aspects of implicit dependencies that arise e.g. from system calls, resource access and similar.

In other words, the approach is comparatively restrictive in comparison to the techniques and means applied by expert parallel developers. Accordingly, there is no guarantee for improved execution performance following this approach, even if resources are generally exploited better.

An alternative to source code level analysis consists in monitoring the actual execution behaviour of a program on **machine code level**. The Service-oriented Operating Systems project[6] (S(o)OS) promotes this approach to gain more fine-granulated data specific information about not only the actual data-flow, but also the work-flow of the code. The (runtime) behaviour provides additional information about the actual connectivity between the individual segments and thus its requirements towards the communication model, i.e. the relationship of latency versus bandwidth.

Implicitly, runtime behaviour effectively provides more information about the potential code distribution than the programmer can currently encode in the source code. This is simply due to the fact that this is not in-line with our current way of writing programs and is implicitly not directly supported by programming models. The foundation is however laid out by integration of remote processes (web services) and dedicated synchronisation points in parallel processes âŁ“ this does not always reflect the best distribution though, as the according invocations are mainly functionality- rather than communication-driven.

By integrating a memory monitor into the kernel, the operating system can acquire information about the memory access behaviour of the full scope of the code, i.e. including jumps, data access and, interestingly, system calls. Like in the source code model, the system can use this information to generate a dependency graph, not unlike the one generated in Star Superscalar (see above). Accordingly, the information can be used in a similar fashion by analysing this dependency graph with respect to the concurrent segments and potential parallel execution.

Due to the nature of runtime code analysis, however, the dependency information is much more fine granulated, leaving little room for "obvious" concurrency. Instead, the graph has to incorporate additional information that for example the Star Superscalar model and the source code level analysis do not consider, in particular:

- access frequency (of invocations or read / write actions)
- type of action (jump, read, write)
- access order in time
- size of the code / data accessed

[6] `http:\\www.soos-project.eu`

With this information, we can derive a graph where the the strength of the relationship and the size of the underlying code / data is encoded as weights (or distances) of vertices and edges. The dependency information in this graph can be used to extract different segments in the form of subgraphs according to nearness (connection strength) and combined size. In other words, according to the number of memory accesses with fewer accesses implying a potentially good cutting point. Segmenting the graph is thereby similar to the problem of identifying the maximum flox in a flow network, and thus the max-flow-min-cut theorem which is often also applied for segmentation purposes in image analysis (see e.g. [**?**]). The minimum cut in our case therefore reflects the segments that share the least dependencies. This means that the created segments can principally be distributed over multiple cores, if the timing dependencies (i.e. synchronisation delays) between the individual functions is respected.

Figure 1.2 Potential segmentation of the reduced graph (simplified).

Dependent segments thereby can nonehtless still be executed in parallel if the according communication and synchronisation means are provided, as discussed above.The maxmimum execution speedup through this form of parallelisation is thereby directly related to the maximum degree of execution overlap that can be achieved without affecting consistency of the program. The overlap should thereby be ideally identical to the maximum delay created by communication.

What is more, by applying similar pattern analysis approaches as in the source code analysis, potential points for parallelisation rather than just concurrent execution can be identified. As such, it can be for example shown that the graph of an Antidependence loop iterates across memory in line with the index and that the cross-dependency between S and T is depicted by a read access prior to a write access on the same memory space, so that both S and T can execute in full parallel by overwriting memory (for B) from S with data from T after execution, or by first executing S in full parallel before unrolling T.

Whilst the benefits of this approach are obvious, the method nonetheless suffers from two strongly related issues: as the code behaviour may be impacted by the environmental conditions, ranging from dynamic sources (such as a keyboard) to data-specific behaviour (for example reacting to specifc occurences in a data stream), the actual dependencies may alter over time. Accordingly, the segmentation at a given point in execution may not be static itself, but subject to changes over execution time.

Figure 1.3 Code to infrastructure mapping principles.

1.3.2 Mapping and Adaptation

It was already mentioned in the beginning of this chapter, how the architecture of future processors is going to change and deviate drastically from today's more or less homogeneous and uniform setups. Already setups such as Intel's Many Integrated Core[7] (MIC) architecture clearly show the tendency towards non-uniform connectivity between processing units, i.e. network on chip connections between cores. Texas Instrument and IBM on the other hand show how future processors will integrate various processing types in a single chip.

Figure 1.4 Rearrangement of concurrent logical segments for maximum speedup.

[7] `http://www.intel.com/technology/architecture-silicon/mic/index.htm`

Accordingly, it will become more than ever important to respect the actual hardware specifics for parallel code distribution. In particular, this relates to the following main criteria:

- cache size
- connectivity (bandwidth and latency)
- ISA / capabilities

These criteria specify how individual code segments should be deployed relative to one another within a single processor, so as to reduce unnecessary overhead and exploit the given specifics to their most. For example, many applications contain logical parts that can be easily vectorized, or at least executed efficiently on a SIMD (Single Instruction, Multiple Data) unit, such as stream processing tasks. If such a unit is available in the processing infrastructure, it should therefore ideally be exploited for the according logic. Similarly, two threads communicating frequently should be placed next to each other in a Network on Chip structure, so as to exploit the communication linkage between the two, rather than far apart with multiple concurring threads inbetween which will lower the bandwidth and increase latency.

The according information for exploiting code specifics can be easily derived from the code analysis as described above. For example the best communication layout relates directly to the connectivity weight between code segments in the behaviour graph. Mapping this relationship information to the network layout is obviously an NP complete tasks, yet classical graph matching strategies can be applied to this problem (see e.g. [**?**]).

More problems however are posed by the specific capabilities of a given processing unit: in order to fully exploit (and to cope with) the heterogeneity of the infrastructure, compiler and ideally execution manager (such as the operating system) of the respective code should be capable of identifying, interpreting and using the hardware specific characteristics. In the example provided above, this would mean that the SIMD core is retrieved and the specific criteria, respectively capabilities towards the code are identified. It would furthermore mean that the infrastructure uses this information to prepare the code accordingly.

As long as the processing units in question adhere to the same Instruction Set Architecture (ISA), conversion (in the sense of potential rearrangement of code) can be easily adhered to. It must be expected however that future models will not even maintain compliant ISAs anymore - accordingly, a porting request of a specific code segment to a non-compliant unit will implicitly require conversion of the underlying ISA. Obviously, this easily achieved at source code level by providing the according compiler directives - on machine code level, this however is nearly impossible without major loss of efficiency. Implicitly, the information gained from machine code monitoring can only be indirectly be exploited, by feeding it back to the source code level and hence the compiler.

What is more, however, is that no current hardware description method allows to identify capabilities in the required fashion. The most widely used description

language for hardware - the Very High Speed Integrated Circuit Hardware Description Language[8] (VHDL) - is too detailed to allow such information extraction and is furthermore too complicated for easy adaptation to the arising scope of new infrastructures. Baaij et al. therefore promote a hardware description language basing on functional declarations that would allow more abstract queries over the structure so as to derive capability information, such as processor type (cf. [?]).

1.4 CONCLUSIONS

The advances in heterogeneous multicore systems has taken the software industry and the programming language community essentially unprepared. Already the advances made on hardware level progress faster than the ones on software and programmming level, so that by now processor architectures start to offer more capabilities than a developer can sensibly exploit. Within this chapter we have highlighted which specific obstacles hinder the developer from making use of such new systems and which specific obstacles are yet to overcome in order to enable the broad scope of developers that will have to make use of these systems in the near future.

A promising approach thereby consists in exploitation of concurrency, rather than "automagic" parallelisation which can only lead to very suboptimal solutions. Concurrency can be analysed on multiple code levels, thus providing information of different granularity - however, all approaches so far still base on the programmer providing the according dependency information. So far this information cannot even be properly validated against the code, thus making it error prone.

What makes the exploitation of concurrency so specifically interesting in this context is not only its ability to support the developer though. Even more important is the capability that proper exploitation of concurrency can further reduce the limitations posed by Amdahl's law: next to the classical means of parallelisation (segmentation of work or data), it can also affect the "unparallelisable" part of the code, i.e. which is denoted as "sequential part" in Amdahl's law. This is achieved by executing multiple sequential logical parts at the same time, rather than parallelising the respective code itself. Due to the nature of this type of parallelisation, however, the scalability is not only restricted by the number of available processing units, but also by the number of concurrent segments that can be identified. In other words, if only 10 concurrent segments can be identified, the maximum theoretically possible speedup is 10 - even if more processing units are available.

Not only the high degree of scalability will pose issues to future programming models, but in particular the large variance of processor architectures with increasing deviations even on the ISA level. So far, the developer must be well aware of these differences in order to give according compiler instructions. To enable compilers or even the execution infrastructure to automatically detect and exploit the hardware specifics, new description languages are needed that can expose the respective unit's characteristics and capabilities in a fashion that can be interpreted according to the

[8] `http://www.vhdl-online.de/`

infrastructure's needs. Functional languages thereby show high promise, as they are more intuitive and flexible than traditional models, and enable abstract queries from which additional information about the specifics can be derived.