Μια Δομημένη Αρχιτεκτονική για το Σύστημα Χρόνου-Εκτέλεσης του Παραλληλοποιητικού Μεταφραστή OMPi

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Γεώργιο Φίλο

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Ιούλιος 2008

Dedication

To my parents and my sister Christina

Acknowledgements

I would like to thank my supervisor Professor V. V. Dimakopoulos for his continuous guiding, motivating and especially for the patience he has shown until this thesis was completed. I also thank him for the opportunity he gave me to travel to Poland/Krakow and present a part of this thesis. Furthermore, i would like to thank Dr. Panagiotis E. Hadjidoukas for his valuable help and the time he spent sharing his knowledge during the elaboration of this thesis.

Table of Contents

List of Figures

List of Tables

Abstract

Giorgos Ch. Philos. MSc, Computer Science Department, University of Ioannina, Greece. July, 2008. A Modular Architecture for the Runtime System of the OMPi Compiler. Thesis Supervisor: Vassilios V. Dimakopoulos.

OpenMP has become a standard paradigm for shared memory programming, as it offers the advantage of simple and incremental program development, in a high abstraction level. In this thesis we purpose a modular architecture for the runtime support of OpenMP programs produced by the OMPi source-to-source compiler. We present the implementation of our runtime system, along with detailed performance evaluation results.

The purpose of this thesis is twofold: study nested parallelism support in OpenMP and extend OpenMP applicability to clustered environments. In the first part we deal with multilevel parallelism which is a major feature of OpenMP. Specifically, threads encountering nested parallel regions are allowed to spawn new threads dynamically. Although many contemporary OpenMP compilation systems provide some kind of nested parallelism support, there has been no evaluation of the overheads incurred by such a support. In this thesis, we present a new runtime threading module for OMPi, called PTHR, which provides basic support for nested parallelism. Using a novel microbenchmark suite, we evaluate how a multitude of freeware and commercial OpenMP compilers behave in the presence of nested parallelism.

In the second part, we concentrate on computational clusters. The most widely used and arguably most efficient tool for programming clusters is the Message Passing Interface (MPI). However, MPI is rather cumbersome as it burdens the application programmer with the explicit distribution of program's data and the orchestration of communications by hand. As OpenMP becomes more and more popular nowadays, researchers have studied ways of extending OpenMP to clusters mostly using Shared Virtual Memory (SVM) libraries which give the illusion of a shared address space on top of a distributed memory environment. We present a new module called OPRC, which is part of the runtime system of OMPi, enabling the execution of OpenMP programs on clusters. The unique features of our work include an abstraction layer which decouples the runtime core from the actual SVM library, making it possible to utilize any arbitrary SVM implementation. We have successfully integrated 5 different SVM libraries with different memory consistency protocols and memory allocation semantics. Our implementation follows a hybrid approach

whereby the SVM subsystem is only utilized for user program shared variables, while internal scheduling and synchronization operations rely on explicit MPI calls. We finally present an experimental evaluation of our platform over a cluster with Gigabit Ethernet interconnects, using a number of typical parallel applications.

Εκτενησ Περιληψη

Γιώργος Φίλος του Χρήστου και της Υπαπαντής. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων. Ιούλιος, 2008. Μια Δομημένη Αρχιτεκτονική για το Σύστημα Χρόνου-Εκτέλεσης του Παραλληλοποιητικού Μεταφραστή OMPi. Επιβλέπων: Βασίλειος Β. Δημακόπουλος.

Το OpenMP είναι ένα πρότυπο για την ανάπτυξη παράλληλων εφαρμογών σε μηχανές κοινής μνήμης. Υποστηρίζει τις γλώσσες προγραμματισμού C/C++ και Fortran. Αποτελείται από ένα σύνολο από οδηγίες (directives) και ρουτίνες βιβλιοθήκης. Σε αντίθεση με άλλα πρότυπα, όπως το πρότυπο POSIX για τα νήματα, το OpenMP είναι μία διεπαφή υψηλότερου επιπέδου που επιτρέπει την παραλληλοποίηση ενός σειριακού προγράμματος με έναν απλό και αυξητικό τρόπο. Οι οδηγίες προστίθενται σε ένα σειριακό πρόγραμμα C/C++ ή Fortran με τέτοιο τρόπο ώστε απλά να αγνοούνται στην περίπτωση που ο μεταφραστής δεν υποστηρίζει οδηγίες OpenMP. Συνεπώς, το πρότυπο επεκτείνει παρά αλλοιώνει την γλώσσα προγραμματισμού.

Ένα βασικό χαρακτηριστικό του OpenMP είναι ο πολυεπίπεδος παραλληλισμός. Συγκεκριμένα, τα νήματα που εισέρχονται σε εμφωλευμένες παράλληλες περιοχές επιτρέπεται να δημιουργούν δυναμικά νέες ομάδες νημάτων. Το χαρακτηριστικό αυτό είναι σημαντικό για ένα ευρύ σύνολο από παράλληλες εφαρμογές που απαιτούν πολυεπίπεδο παραλληλισμό ώστε να πετύχουν ικανοποιητική επιτάχυνση (speedup). Παρά την σημασία του, η υποστήριξη του άργησε να εμφανιστεί στους μεταφραστές. Στις μέρες μας, οι περισσότεροι μεταφραστές OpenMP παρέχουν κάποιου είδους υποστήριξη για πολυεπίπεδο παραλληλισμό. Όμως, μέχρι στιγμής δεν έχει παρουσιαστεί κάποια μελέτη σχετικά με το επιπλέον κόστος που εισάγεται στο σύστημα λόγω της διαχείρισης των πολλαπλών επιπέδων παραλληλισμού. Στην παρούσα διατριβή, παρουσιάζουμε μια βιβλιοθήκη χρόνου-εκτέλεσης (runtime library) για τον παραλληλοποιητικό μεταφραστή OMPi που επιτρέπει την εκτέλεση OpenMP προγραμμάτων με βασική υποστήριξη για πολυεπίπεδο παραλληλισμό. Επιπλέον, αναπτύξαμε μια πλατφόρμα αξιολόγησης (benchmark) και παρουσιάζουμε μια πειραματική μελέτη της απόδοσης ενός συνόλου από εμπορικούς και πειραματικούς μεταφραστές OpenMP υπό το καθεστώς πολυεπίπεδου παραλληλισμού.

Πρόσφατα, η έρευνα έχει στραφεί σε τρόπους επέκτασης της εκτέλεσης OpenMP προγραμμάτων σε μεγαλύτερα υπολογιστικά περιβάλλοντα, όπως οι συστάδες υπολογιστών (clusters), συνδυάζοντας έτσι την απλότητα του προγραμματιστικού μοντέλου του OpenMP με

την υπολογιστική ισχύ που προσφέρουν τα συστήματα αυτά. Το πιο γνωστό και ευρέως αποδεκτό μοντέλο προγραμματισμού σε συστάδες υπολογιστών είναι η μεταβίβαση μηνυμάτων και συγκεκριμένα το MPI (Message Passing Interface). Παρόλο που η προσεκτική χρήση του MPI επιφέρει καλή απόδοση στις εφαρμογές, ο προγραμματιστής καλείται να οργανώσει τις επικοινωνίες και να κατανέμει τα δεδομένα του προγράμματος ρητά στους κόμβους του συστήματος. Οι υπάρχουσες υλοποιήσεις OpenMP για συστάδες υπολογιστών χρησιμοποιούν εσωτερικά βιβλιοθήκες κοινής εικονικής μνήμης (Shared Virtual Memory - SVM) που παρέχουν έναν εικονικό κοινό χώρο διευθύνσεων. Συγκεκριμένα, η κοινή μνήμη είναι εικονική και αποτελείται από τμήματα των φυσικών μνημών των κόμβων, ενώ η συνοχή και η συνέπεια της κοινής μνήμης υλοποιείται εξολοκλήρου σε λογισμικό. Με αυτόν τον τρόπο, εξασφαλίζεται το μοντέλο κοινής μνήμης που προϋποθέτει το πρότυπο OpenMP, σε ένα κατανεμημένο περιβάλλον. Παρόλα αυτά, η απόδοση αυτών των βιβλιοθηκών δεν είναι ικανοποιητική. Η συχνή και χρονοβόρα επικοινωνία που απαιτείται για να εξασφαλιστεί η συνοχή και συνέπεια της κοινής μνήμης μειώνει αισθητά την απόδοση των εφαρμογών, ιδίως όταν η εφαρμογή απαιτεί συχνή τροποποίηση κοινών δεδομένων.

Στην παρούσα διατριβή, παρουσιάζεται μια βιβλιοθήκη χρόνου-εκτέλεσης για τον παραλληλοποιητικό μεταφραστή OMPi που υποστηρίζει την εκτέλεση OpenMP προγραμμάτων σε συστάδες υπολογιστών. Για λόγους απόδοσης, η υλοποίηση μας είναι υβριδική: ένας πυρήνας SVM χειρίζεται τα δεδομένα της εφαρμογής που ορίζονται ως κοινά μεταξύ των διεργασιών, ενώ οι ανάγκες για την δρομολόγηση και τον συντονισμό των διεργασιών εσωτερικά στην βιβλιοθήκη εξυπηρετούνται με MPI. Ως αποτέλεσμα, επιτυγχάνουμε αποδοτικότερες επικοινωνίες. Ενώ οι υπάρχοντες μεταφραστές OpenMP συνήθως στοχεύουν μία συγκεκριμένη βιβλιοθήκη SVM που είναι αναπόσπαστο τμήμα του μεταφραστή, η υλοποίηση μας μπορεί να εκμεταλλευτεί οποιαδήποτε τέτοια βιβλιοθήκη επιθυμεί ο προγραμματιστής καθώς το σύστημα χρόνου-εκτέλεσης του OMPi είναι ανεξάρτητο από τον πυρήνα SVM που χρησιμοποιείται. Ενσωματώσαμε επιτυχώς 5 βιβλιοθήκες που ακολουθούν διαφορετικά μοντέλα συνέπειας της κοινής μνήμης και παρουσιάζουμε πειραματικά αποτελέσματα από μια συστάδα υπολογιστών, μελετώντας συγκριτικά την επίδοση του μεταφραστή OMPi, σε ένα σύνολο από παράλληλες εφαρμογές.

Chapter 1

Introduction

1.1 OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessor programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives and a runtime system supporting calls. In contrast with other APIs such as the POSIX threads, OpenMP is a higher level API which allows the programmer to parallelize a serial program in a simple, controlled and incremental way. It provides directives for expressing parallelism, worksharing and synchronization. The OpenMP directives are added to an existing serial program written in C/C++ or Fortran in such a way that they can safely be discarded by compilers that do not support OpenMP (thus leaving the original program unchanged). As a consequence, OpenMP extends rather than changes the base language (C/C++ or Fortran).

Nowadays, OpenMP has become a standard paradigm for programming symmetric shared memory multiprocessors (SMP). Its usage is continuously increasing as small SMP machines have become the mainstream architecture even in the personal computer market, thanks to the domination of multicore CPUs. Its popularity has been proven from the fact that many research and commercial/proprietary OpenMP compilers are now available. Companies like Fujitsu, HP, Intel, Microsoft and Sun have developed OpenMP-compliant compilers. Also, a multitude of research/experimental OpenMP compilers exist nowadays. Namely, some of them are: the OMPi compiler [10], the Omni compiler [31], the OpenUH compiler [27], and the Nanos Mercurium compiler [2].

Without dispute, OpenMP is very popular nowadays. It's main advantage is the programming simplicity. The API hides all the cumbersome details from the user. Since the first version (v1.0) of the API specification, a number of new features have been added to OpenMP. Its current version is v3.0. Research on OpenMP includes the improvement of the API so as to be more useful to the end users, and the development of efficient compilation and runtime systems supporting OpenMP.

1.2 Thesis Objectives

The OMPi compiler is a light-weight, portable and modular source-to-source compiler for the v2.5 OpenMP specification. It currently supports only the C programming language. The OMPi compiler is the result of the work of the Parallel Processing Group (`PARAGROUP`) at the Computer Science Departure of University of Ioannina. Its first public release was in 2003. The current version of OMPi is v1.0.0 featuring a redesigned-from-scratch translator and an enhanced runtime system, which is based on the work described in this thesis. This work is mainly focused in the extension of OMPi's runtime system. Specifically, the contributions of this work are the following:

- Provision of runtime support for nested parallelism, along with a novel microbenchmark suite for assessing its performance.
- Development of a new portable and modular runtime library for the execution of OMPi programs on top of clusters.

1.2.1 Nested Parallelism in OpenMP

Nested parallelism has been a major feature of OpenMP since its very beginning. As a programming style, it provides an elegant solution for a wide class of parallel applications, with the potential to achieve substantial utilization of the available computational resources, in situations where outer-loop parallelism simply cannot. Notwithstanding its significance, nested parallelism support was slow to find its way into OpenMP implementations, commercial and research ones alike. Even nowadays, the level of support is varying greatly among compilers and runtime systems.

Our objective is to provide runtime support for nested parallelism in OMPi. To this end, we first develop a new threading library for OMPi. We also develop a microbenchmark suite based on the EPCC microbenchmarks [4], which allows us to measure OpenMP overheads when nested parallelism is in effect. Using our methodology, we perform an experimental study of the overheads introduced in nested parallelism, providing results for a number research/experimental and freeware/proprietary compilation systems.

1.2.2 OpenMP on Clusters

Computation clusters have emerged as a cost-effective approach to high performance computing (HPC). Individual machines unified by a LAN, either using a commodity or high performance interconnect, can be viewed as a virtual large-scale machine with a big number of processors and can be programmed as such. They offer an expandable and reliable computational environment which is quite more economic than large massively parallel machines. However, programming for a cluster is rather cumbersome. The most widely used and arguably most efficient tool for cluster programming is the Message Passing Interface (MPI). Nevertheless, MPI forces the programmer to explicitly distribute the program's data and orchestrate communications by hand, and as a result it has not found its way to mainstream computing.

An alternative to MPI is the use of *shared virtual memory* (SVM) libraries which give the illusion of shared memory. An equivalent term for shared virtual memory is *software distributed shared memory* (sDSM). Many SVM libraries have been developed in the past. They all provide an API for allocating shared memory on a distributed environment along with synchronization routines. Most of them employ relaxed memory consistency protocols meaning that memory updates are delayed until synchronization. Consequently, this forces the programmer to insert explicit synchronization calls in order to make sure that the program executes correctly. Although SVM systems do not seem to be able to achieve the speedups possible with carefully hand-coded MPI programs, they have nevertheless been proven successful for a number of data-intensive applications. A problem with SVM systems is the complete incompatibility between the various implementations and the esoteric API they usually provide. As a result, it is not always easy to experiment with and compare such systems.

The combination of OpenMP and SVM systems has been proposed by many researchers as a convenient means of leveraging a cluster, matching the programmer-friendliness of OpenMP with the SVM layer that abstracts away the underlying distributed architecture. Any peculiarities of the SVM layer are completely hidden from the programmer and are left to the compiler and runtime system to handle. A number of research/experimental compilers support OpenMP on clusters. The Omni compiler [31] uses the SCASH SVM system to implement the shared memory semantics. The ParADE OpenMP translator [21] is based on the Omni Compiler and utilizes its own underlying SVM system. The Nanos compiler [8] also supports a cluster execution environment. Many researchers began with the development of SVM systems and later integrated a compiler and runtime system for the support of OpenMP on clusters. Intel has recently released v9.0 of its OpenMP compiler, which extends it to clusters [13]. It internally targets a modified version of the TreadMarks commercial SVM system [23].

However, almost all OpenMP implementations are based on a tight coupling of the compiler and the runtime library. The whole system targets SMP machines or clusters but

usually not the both. Even in the few cases that supports both, there is a fixed, built-in threading library and an SVM core and the generated code targets them specifically, making it almost impossible to experiment with alternative configurations.

Our objective is to develop an efficient and modular runtime system for the execution of OMPi programs on top of clusters. We use a hybrid approach where communication at the runtime library is achieved by explicit message passing (MPI), while an SVM core provides the shared memory semantics at the application level. The key feature of our design is that the SVM core is not a fixed part of the runtime system, allowing the integration of any desirable SVM library. We managed to experiment with different configurations and provide comparative results.

1.3 Thesis Structure

This thesis is organized as follows:

- Chapter 2 presents briefly the OpenMP API and describes the compilation process and the basic transformations made by the OMPi compiler, in the presence of threads or SVM processes.
- Chapter 3 presents in detail the runtime architecture of OMPi.
- Chapter 4 describes our design for the support of nested parallelism. In addition, it presents our microbenchmark methodology, along with comparative experimental results for a multitude of OpenMP compilation systems.
- Chapter 5 is a self-contained introduction to shared virtual memory concepts. It also surveys the ongoing research regarding OpenMP program execution on clusters using SVM libraries.
- Chapter 6 presents our runtime architecture for the execution of OMPi programs on top of clusters along with implementation details. This chapter also provides experimental results.
- Chapter 7 concludes this thesis with a summary of our contributions and possible directions for future work.

Chapter 2

OpenMP and OMPi

2.1 OpenMP in Brief

The OpenMP API is comprised of three primary components: compiler directives, runtime library routines and environmental variables. The OpenMP directives in C have the general format of:

```
#pragma omp directive-name [clause,...] newline
```

Each directive applies to the succeeding statement, which must be a structured block. OpenMP specifies a set of syntax and binding rules for the directives. In this section we will not cover all the details; instead, we will present briefly the most important and commonly used features of the API. For more details, the reader is referred to the official OpenMP API specification [1].

2.1.1 The Parallel Construct

The programmer defines a structured block of code to be executed by multiple threads using the `parallel` directive. OpenMP adopts the fork/join model. The master thread, i.e. the thread that originally executes the user program, creates a team of worker threads whenever a `parallel` directive is encountered. All worker threads independently execute

the same block of code enclosed within the `parallel` directive. At the end of the parallel region, all threads are synchronized and only the master thread continues with the sequential execution of the succeeding code. For example, consider the following code in OpenMP/C:

```
main(){
  int id;
  /* Fork a team of threads */
  #pragma omp parallel private(id)
  {
     /* Each thread has its own id */
     id = omp_get_thread_num();
     printf("hello from thread %d\n", id);

     if(id == 0)  /* Only master do this */
     {
       printf("number of threads = %d\n", omp_get_num_threads());
     }
  }
  /* Only the master thread reaches this point */
}
```

Each thread has a unique id which is available through a call to `omp_get_thread_num()`. Threads are numbered sequentially starting from 0 (master thread). The number of threads executing a parallel region is queried by a call to `omp_get_num_threads()`.

The number of threads in a parallel region depends on the following factors:

- Use of the `omp_set_num_threads()` library routine.
- Value of the `OMP_NUM_THREADS` environmental variable.
- Implementation default.

The `omp_set_num_threads()` has precedence over the `OMP_NUM_THREADS` environmental variable. By default, a program with multiple parallel regions will use the same number of threads to execute each parallel region. This behavior can be changed to allow the runtime system to dynamically adjust the number of threads that are created for a given parallel section. The programmer can turn on the dynamic mode through the following methods:

- Use of the `omp_set_dynamic()` library routine.
- Setting the `OMP_DYNAMIC` environmental variable.

To assure that the requested number of threads will actually be created, the programmer must turn off the dynamic mode and explicitly set the number of threads via the `omp_set_num_threads()` routine.

OpenMP allows parallel regions to be nested each other. This feature is optional. When nested parallelism is supported by an implementation and is enabled, multiple teams of threads are created. Each thread in the first level creates a new team. If nested parallel is not supported or is disabled, each thread in the first level creates a new team consisting of only one thread, that is to say, the parallel region is serialized.

2.1.2 Workshare Constructs

The most important feature of OpenMP is the support of threads worksharing. The `for` directive is the most commonly used workshare directive in OpenMP programs. For loop iterations are divided into chunks and scheduled among the executing threads according to a schedule policy. Consider the following part of a simple matrix multiplication program using OpenMP/C:

```
1 #pragma omp parallel for private(i,j,k) schedule(static)
2    for(i = 0; i < rows; i++) {
3     for(j = 0; j < cols; j++) {
4       for(k = 0; k < rows; k++) {
5         c[i][j] += a[i][k]*b[k][j];
6       }
7     }
8    }
```

Note that, the `parallel` directive can be combined with the `for` directive in a single OpenMP statement. This means that a new team of worker threads will be created and the first `for` loop's iterations (line 2) will be scheduled among them. Each thread will execute the succeeding code (lines 3-6) as many times as its assigned iterations. Eventually, after the work is done, all threads will be synchronized.

OpenMP offers different schedule policies that can be applied to loop iterations. The default schedule policy, namely static, defines that loop iterations are divided into chunks of equal size and scheduled among the executing threads. However, the static schedule does not take into account the possible load/speed imbalance of the executing threads. For this reason, OpenMP provides two additional schedules, namely dynamic and guided. In dynamic and guided schedules, chunks are dynamically scheduled among the threads; when a thread finishes one chunk, it contents for another.

The `section` directive provides also workshare semantics. With the use of the `section` directive the work is divided into the user-defined sections. Each section is assigned to a different thread. If the number of sections are more than the number of threads, then some threads will eventually execute more than one sections.

2.1.3 Data Scoping

OpenMP provides a set of constructs to define how and which data variables in the serial section of the program are transferred to the parallel section of the program. The most commonly used data scope constructs are:

- `private`: This clause is used to declare a list of variables as private to each thread for a given region. Each thread reads/modifies it's own copies of these variables. Private variables can either be stack variables or even global variables.
- `shared`: This clause is used to declare a list of variables as shared among threads. Shared variables can either be global variables or stack variables.
- `threadprivate`: This directive is used to declare global file scope variables(C/C++) or common blocks (Fortran) as thread-private among the threads. The difference with `private` is that thread-private variables are able to persist among multiple parallel regions. Each thread gets its own copy of the variables, so data written by one thread is not visible to other threads.

By default, all variables are declared as shared. The `private` and `shared` clauses are used in conjunction with the `parallel` and `for` directives to control the scoping of enclosed variables. The `threadprivate` directive must appear after the declaration of the associated variables.

2.1.4 Synchronization Constructs

OpenMP provides a set of synchronization directives which are necessary when programming in a shared memory environment. These are the `barrier`, the `critical`, the `atomic` and the `flush` directives. The `barrier` directive provides a synchronization point among all threads in the thread team. When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached the same barrier. The `critical` directive specifies a critical region of code, i.e. a region of code that must be executed by only one thread at a time. The `atomic` directive is a mini critical section where only a specific memory location must be updated atomically. The `flush` directive is used to enforce a consistent view of memory.

2.1.5 The Reduction Clause

The `reduction` clause performs a scalar operation on the variables that appear in its list. A private copy for each variable is created for each thread. At the end of the reduction, the reduction operation is applied to all private copies of the shared variable, and the final result is written to the global shared variable. For example, consider the following OpenMP/C code which calculates the value of pi:

```
#define N 65536
#define W 1.0/N
main(){
  double pi = 0.0, lpi;
  int i;
  #pragma omp parallel private(i, lpi) reduction(+:pi)
  {
    lpi = 0.0;
    #pragma omp for schedule(static)
      for(i = 0; i < N; i++)
        lpi += (4*W)/(1+(i+0.5))*(i+0.5)*W*W);
    pi += lpi;
  }
 /* Master thread */
 printf("pi = %f\n", pi);
}
```

Iterations of the parallel loop will be equally distributed to threads (`static`). Each thread will calculate its own part of the final value of `pi`. At the end of the parallel loop construct, all threads will add their private values (`lpi`) to update the master thread's global copy. Instead of using the `reduction` clause, we could also use the `atomic` directive, so as each thread in the team atomically updates `pi`.

2.1.6 Library Calls and Environmental Variables

OpenMP defines a set of library calls to perform a variety of functions. We have already seen some of them, in the previous sections. Generally, these library routines are categorized as follows:

- Query the number of threads/processors, set number of threads to use.
- General propose locking routines.
- Set execution environment routines.

Also, the execution of the parallel code can be controlled through 4 special environmental variables:

- `OMP_NUM_THREADS`: Set the default number of threads to be created at the program's parallel regions.
- `OMP_SCHEDULE`: The schedule policy used at a `for` construct. Valid values are `static`, `dynamic` or `guided`.
- `OMP_DYNAMIC`: Enables or disables the dynamic adjustment of the number of threads available for executing parallel regions. Valid values are TRUE or FALSE.

- `OMP_NESTED`: Enables or disables nested parallelism. Valid values are TRUE or FALSE.

2.2 The OMPi Compiler

OMPi's source-to-source translator takes as input C source code with OpenMP directives and outputs transformed but *equivalent* C code augmented with calls to OMPi's runtime system. The compiler and the runtime system is entirely written in C. In its current version, it features a parser capable of understanding programs with C99 syntax and OpenMP v2.5 directives.

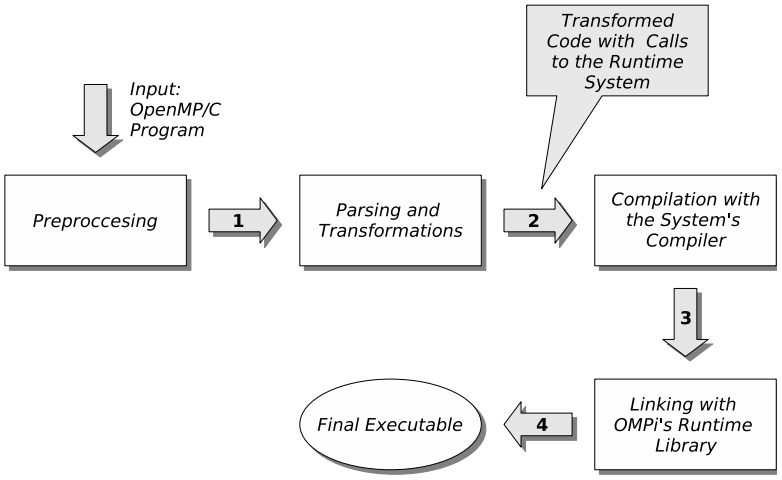

Figure 2.1: The compilation process.

During parsing, which is the first phase of the compilation process, an abstract syntax tree (AST) is built, which represents the original program. The AST is the input of the second (transformation) phase. The transformer visits the tree nodes and acts whenever a node containing an OpenMP statement is met; it then replaces the whole subtree rooted at that node by a new one which mostly maintains the original block of statements but has additional calls to the runtime system inserted at appropriate places. The third (final) phase of the compilation process simply traverses the transformed AST and prints out the corresponding C code. The resulting program is compiled by the system's native C compiler and linked with the runtime library producing the final executable. Figure 2.1 shows the compilation steps.

2.3 OMPi's Transformations for Threads

While some transformations are relatively intuitive, some others are quite involved. The most crucial transformation is the one made when an OpenMP parallel directive is encountered. For example, consider the following simple code in OpenMP/C:

```
void f() {
  #pragma omp parallel
  {
    printf("Hello world from thread %d\n", omp_get_thread_num());
  }
  g();
}
```

The equivalent but multithreaded code produced by OMPi is as follows. OMPi follows the outlining approach [7]. Specifically, the portion of the code enclosed within the parallel directive is moved to another function (`_thrFunc0_()`) which is eventually called by all created threads.

```
static void * _thrFunc0_(void *_arg) {
 /* #pragma omp parallel -- body moved bellow */
 {
   printf("hello world from thread %d\n", omp_get_thread_num());
 }
 return (void *)0;
}
```

In `f()`, a runtime call to `ort_execute_parallel()` is inserted in place of the migrated code. The master thread calls this routine to create a new team of threads. The first argument is the number of threads to be created. The -1 means that the runtime system will decide for the size of the thread team. The second argument is the name of the thread function (`_thrFunc0_`) and the third argument is a pointer to possible shared data among the threads:

```
void f() {
 {
 /* #pragma omp parallel */
 ort_execute_parallel(-1, _thrFunc0_, (void *)0);
 }
 g();
}
```

All new threads including the master thread call the thread function with the latter returning back in `f()` after thread-synchronization, so as to continue with the succeeding program code.

The most important problem arising from this design is that of variable visibility. As we already mentioned in Section 2.1.3, OpenMP provides ways of changing the default scope of variables used within a parallel region. For stack variables declared as `private`, the compiler just clones the variable declarations into the thread function. By default, these variables will be private among the threads. The same approach is used for global

variables declared as `private`. For global variables declared as `shared` there is nothing to do actually. The main problem arises for stack variables that need to be shared. The solution is the use of pointers. For example, consider the following code in OpenMP/C:

```
1 int a;
2 void f() {
3   int b, c, d;
4   #pragma omp parallel private(d)
5     a = b + c + d;
6  }
```

Variables `a`, `b`, and `c` must be shared by default. Variable `d` needs to be private. In this case the resulted transformed code is the following:

```
void f() {
  int b, c, d;
  struct { int (*b); int (*c); }_shvars = {&b, &c);
  ort_execute_parallel(-1, _thrFunc0_, &_shvars);
}
```

Global variable `a` needs no special treatment since global variables are by nature shared among threads. Variable `d` must be private to each thread; this is easily achieved by cloning `d`'s declaration in the thread function. However, `b` and `c` are to be shared but are stack variables. Sharing is achieved by creating pointers to them and passing these pointers explicitly to the thread function. Threads can access them through a runtime call to `ort_get_shared_vars()`. This also necessitates the transformation of the original code (line 5) since in the thread function `b` and `c` are now pointers.

```
static void *_thrFunc0_(void *_arg){
  struct {int (*b); int (*c);} *_shvars = ort_get_shared_vars();

  int *b = _shvars->b;   /* shared non-global */
  int *c = _shvars->c;   /* shared non-global */
  int  d;                /* private */

  a = (*b) + (*c) + d;   /* Transformation due to pointers */
  return (void *) 0;
}
```

2.4 Support for Processes

OpenMP is an API for programming parallel computers with physically shared memory. When the execution environment changes to a cluster, the programming model also changes. The two main changes are:

- We can no longer assume the thread-execution model. Execution entities (EEs) are now processes instead of threads
- The system's memory is no longer shared among the processors. System's memory is private and distributed among the computational nodes

Notwithstanding the programming model change, we must still provide the features of OpenMP, without changing the directives semantics. The programmer must be able to write shared memory based programs without caring whether the program runs on an cluster or a single multiprocessor system.

As far as the compiler is concerned, the majority of the original transformations made for the thread-model work fine in the process-model, too. However, global variables and OpenMP data clauses need a special treatment. In the thread-model, global variables are by nature shared among the threads. In the process model this is not the case; global variables are attached to each process's private address space. Also, consider the stack variables that need to be shared because of the presence of a `shared` OpenMP clause. Pointer passing no longer works because processes can not access the stack space of each other.

2.4.1 Global Variables

Global variables must somehow become shared among the executing processes. As we already mentioned in Chapter 1, SVM systems provide shared memory semantics on top of distributed memory systems. So, we have a way of allocating shared memory on top of a cluster. The question is, *how to reallocate the whole global address space into the SVM system's shared memory*? The answer is through the compiler. In particular,

- The compiler first identifies all the user's global variables in the program and transforms them into pointers of the same type as the original variables.
- The compiler creates a constructor function; a function that will be called before the `main()`, which makes a runtime call to `ort_sglvar_allocate()` for each global variable.
- Finally, the `ort_sglvar_allocate()` routine is responsible for passing the control to the runtime system. Upon initialization, the runtime library allocates a shared memory area and assigns the pointer of each variable to an appropriate offset of this area, writing in the initial variable's value, if any.

For example, consider the following:

```
int a = 1, b = 2, c;
void f() {
  #pragma omp parallel
      c = a + b;
}
```

Variables a, b, c are global variables and must be shared among processes. Moreover, a and b are initialized. The resulted transformed code is the following

```
int _sglini_a = 1, (*a), _sglini_b = 1, (*b), (*c);

static void *_thrFunc0_(void *arg) {
  /* #pragma omp parallel - body moved below */
  (*c) = (*a) + (*b);
  return (void *) 0;
}
```

Variables a, b and c are all transformed into pointers of the same data type. All references of these variables will be also transformed into pointer accesses. Moreover, 2 additional variables, namely _sglini_a and _sglini_b, contain the initial values of a and b.

```
void f() {
  ort_execute_parallel(-1, _thrFunc0_, (void *)0);
}

static void __attribute__((constructor))_init_shvars_0(void)
static void _init_shvars_0(void){
  ort_sglvar_allocate((void **)&c, sizeof(int), (void*)0);
  ort_sglvar_allocate((void **)&b, sizeof(int), (void*)&_sglini_b);
  ort_sglvar_allocate((void **)&a, sizeof(int), (void*)&_sglini_a);
}
```

The _init_shvars_0() is the constructor function called right before the program's main(). It contains 3 calls to ort_sglvar_allocate(); one for each global variable. The reader may wonder why the constructor function is necessary. Consider the case where many independent C modules contain global variable definitions and all are linked together into one executable file. In this case, its impossible to know all these variable definitions at the compile time. By defining a constructor function in each C module, we guarantee that all these constructors will be eventually called before main does. Also, we must ensure that the constructors names are different in each file. The parser takes care of this, by generating a unique id attached to the constructor's name.

Omni for clusters [31], follows the same strategy. However, some other implementations such as the NanosCompiler [8], are following a different approach. In Nanos, the whole process's address space is shared through the underlaying SVM system. In this approach, nothing has to be done for the global variables neither by the compiler nor the runtime system. As we will see later in this work, letting the SVM system to handle everything results in poor performance.

2.4.2 Non-Global Shared Variables

Stack variables can also be declared as shared through the OpenMP `shared` clause. In Section 2.3, we described the transformation made by our compiler in the case of threads. When the execution entities are processes, the pointers created by the compiler are no longer valid. A process can not access the private stack space of another process. However, if the stacks of all processes are explicitly allocated in shared memory, then the mechanism of the pointers will work without any further modifications. Considering a single level of parallelism support, we only have to make sure the master thread (process 0) runs on a shared stack. All other processes will access all shared stack variables through pointers to the shared stack of the master thread. These pointers are this way valid, since they point into a shared memory region. Note that, with this technique, the compiler needs absolutely no modifications. We will not further discuss the implementation details in this chapter, as the solution is implemented entirely at the runtime system.

In [14, 31] a different approach is followed. For every parallel region (a) a new shared memory area is created (b) stack variables are copied into this area and (c) at the end of the parallel region, variables are copied back into their original area and the shared memory area is released. This technique needs special treatment by the compiler and also hides a considerable amount of memory allocation/copy/deallocation overheads.

Chapter 3

OMPi Runtime Architecture

The runtime system of OMPi provides the execution entities that will carry out the work of OpenMP threads and controls their operation and synchronization. It consists of two modules, as shown in Figure 3.1. The first module (ORT) groups the EEs, coordinates them and schedules their execution within worksharing regions, but it *does not implement them*. The second module (EELIB) is the one that implements them. A multitude of EELIB libraries are currently available, adhering to a unified interface. ORT's operation is largely independent of the actual EELIB employed.

3.1 Initialization

Upon program startup, ORT is firstly invoked by a call to `ort_initialize()`. This routine is responsible for initializing the whole runtime system. The compiler inserts this call in the program's `main()` function. Its duties are:

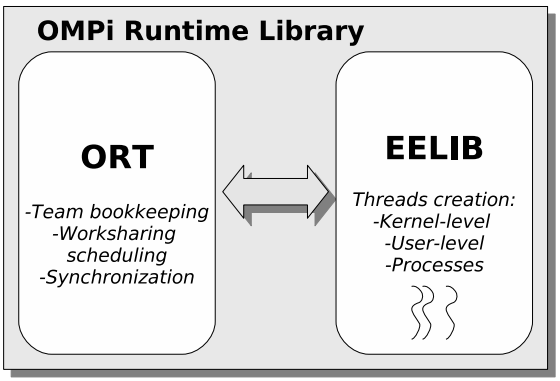

Figure 3.1: OMPi Runtime Organization.

1. Processing the OpenMP environmental variables.
2. The initialization of the EELIB.
3. The construction of the master's *control block* (eecb); an EE-specific block containing everything ORT needs in order to schedule the EE.
4. The management of the program's global data. This is only necessary when EEs are processes.

3.2 Entering a Parallel Region

When called to execute a parallel region (through the `ort_execute_parallel()`), ORT enters a negotiation phase with EELIB, asking for a particular number of EEs, depending on what the program requests and whether nested parallelism and the dynamic adjustment of the number of threads is enabled or not. After EELIB confirms the availability of EEs, it gets instructed by ORT to release them in a bunch, as a team. When an EE from the team commences execution, its very first obligation is to call `ort_get_ee_work()`, which supplies all the information for the work the EE is supposed to do.

Specifically, among other things, it provides a pointer to the function to be executed. At this point, each EE initializes its own eecb. The eecb includes information regarding the team size, the id of the EE within the team, its parallel level and a pointer to the eecb of the team's parent. Through the latter pointer, ORT maintains a dynamic tree of eecbs which grows whenever a new team of EEs is unleashed and shrinks whenever a team completes the execution of a parallel region. In Figure 3.2, such a tree is depicted. Upon startup, the sole EE running is the *initial* EE and operates in level 0. Whenever, an EE encounters a parallel region, it becomes the parent of the spawned team; if the parent is in level i, its children lie in level $i+1$. Also, note that a new eecb is created for the parent of the team, as a member of the spawned team. When the parallel region is over, the parent assumes again it's original eecb. The eecb holds additional information

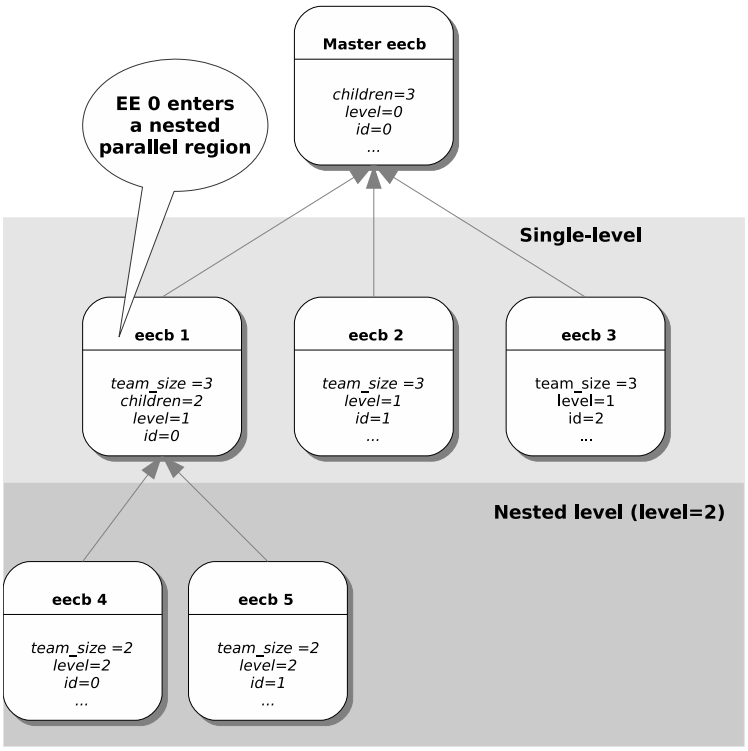

Figure 3.2: An example of the dynamic tree of eecbs.

for the EE that becomes a parent of a new team. This includes a barrier structure for synchronizing the team members, a `copyprivate` staging area for `single` constructs that require it and a structure with scheduling information for work-sharing regions.

There is no prerequisite regarding an EE's level, providing thus full and unlimited support for nested parallelism, as long as EELIB is willing to supply EEs.

3.3 Workshare Region Scheduling

OpenMP defines three workshare directives, namely, `for`, `sections` and `single` whereby the work is divided appropriately among the participating EEs. These code regions are normally blocking, in the sense that they conclude with an implied barrier that synchronizes the EEs before letting them continue their execution. However, when the `nowait` clause is present, there is no implied barrier and the region is non-blocking; such regions present bookkeeping complications. In all three directives, the runtime library needs some kind of counters to count the number of EEs that have passed through their regions. For example, in the `sections` case we need a counter x so as to assign the xth section to the xth arriving EE. For a `single` region; a region that must be executing by only one EE, all but the first EE that arrives should not execute the region. In order to ensure this, we need a counter or a flag. However, keeping a counter or a flag for servicing all workshare regions is impossible when regions are non-blocking. This is because some EEs of the

team may advance to subsequent workshare regions. In this case, multiple regions can be *active* at any given time. A workshare region is *active* when:

- at least one EE has entered and
- not all EEs have passed through it yet.

A solution to the problem could be to keep seperate counters or flags for each workshare region. The compiler statically numbers the directives, giving each one a unique id. This id is then used by the library to index the corresponding counter. However, this approach does not solve the problem completely. For example, consider the case where a `single` or a `sections` directive is called repeatedly within a loop. In this case, a counter or flag for this region is not enough. EEs proceed with different speeds and, due to the `nowait` clause, chances are that different EEs may have encountered the same region a different number of times at any given moment.

Solutions to this problem include bookkeeping using a dynamically allocated list of workshare region structures [3] or avoiding the problem altogether by disallowing more than one non-blocking regions to be simultaneously active, as in the runtime library of the Omni compiler [31]. The approach followed in OMPi is similar to [18]. In the control block of the parent of a team, ORT maintains a preallocated workshare queue of fixed size (MAXWS) with bookkeeping information about each active workshare region. Stored information includes construct-specific data (e.g. the number of remaining sections for a `section` construct; the next iteration to be scheduled and the increment step for a `for` construct; locks for protecting accesses to this data by the EEs of the team) plus queue-related data, such as the number of EEs that have exited (finished) this region. When the tail and the head of the queue are MAXWS regions apart, i.e. there are MAXWS simultaneously active regions, any EE that tries to activate a new region gets blocked until the tail of the queue advances. This way, we avoid the cost of dynamic adjustment of the capacity of the queue, without introducing the artificial barrier required in [18].

ORT optimizes the operation of the workshare queue by using lock-free accesses when possible and by employing atomic operations if available, resorting to plain locking whenever really necessary. A final optimization is the avoidance of the full initialization of the queue. Every time a new team of EEs is created, all regions of the queue must be properly initialized by the parent before being put to use. If MAXWS is not small this results in a major overhead. ORT avoids this by initializing only the first region of the queue; the first EE to enter a new non-blocking region is responsible for initializing the next region in the queue. This way, at any given time, the queue has one extra region ready for use.

From ORT's point of view, two routines are always involved when a workshare directive is encountered. Every EE begins its region with an `ort_enter_workshare_region()` call and finishes it with an `ort_leave_workshare_region()` call. These two calls do all the management of the workshare queue. If the EE is the first to enter a workshare region,

it is responsible for initializing the region's specific structure and also prepare the next region in the queue. All other EEs entering the region do absolutely nothing. When the EE finishes its assigned work, it just calls the `ort_leave_workshare_region()` routine. If the EE is the last to leave, it marks the region as `empty`. Otherwise, it just decrements the `notleft` counter. Marking the region as empty, enables us to do a kind of recycle; the region's structure can be used again by a subsequent workshare region in the program.

We close this section with an example:

```
void f() {
  int i;
  #pragma omp parallel
  #pragma omp for private(i) schedule(static)
    for(i = 0; i < 100; i++)
      do_some_calculations(i);
}
```

This is a simple program using the `for` directive and `static` schedule. The function called by all EEs is `_thrFunc0_()`:

```
1  static void * _thrFunc0_(void *arg)
2  {
3   {
4      int i;
5      int from_ = 0, to_ = 0, step_;
6      struct _ort_gdopt_ gdopt_;
7
8      step_ = 1;
9      ort_entering_for(1, 0, 0, step_, &gdopt_);
10     if(ort_get_static_default_chunk(0, 100, step_, &from_, &to_))
11     {
12       for(i = from; i < to_; i = i + 1)
13         do_some_calculations(i);
14     }
15     ort_leaving_for();
16   }
17   return(void*)0;
18 }
```

The first ORT routine called by each running EE is `ort_entering_for()`(line 9). Internally, this routine includes a call to `ort_enter_workshare_region()`. Its first argument informs ORT about the region type; blocking or non-blocking. 1 means that the region is non-blocking, i.e. it has a `nowait` clause. However, the compiler is clever enough to see that there is no need to have two barriers at the end of the parallel region; one for the `parallel` directive and one for the `for` directive. So, it removes the implied `for`

barrier and tells ORT that its a `nowait` region. The second argument tells ORT if the `for` directive is combined with the `ordered` clause or not. The third and forth arguments are the loop's lower bound and step, respectively. The last argument is used only for optimizations at guided and dynamic schedules. The `ort_get_static_default_chunk()` is responsible for scheduling the loop's iterations among the calling EEs (line 10). Finally, each EE finishes the region by calling the `ort_leaving_for()` (line 15) which internally just calls `ort_leave_workshare_region()`.

3.4 Synchronization

ORT provides support for the synchronization directives of OpenMP. ORT provides an efficient barrier implementation for the support of the `barrier` directive. The compiler replaces the directive by a call to `ort_barrier_me()`. When an EE calls this routine, it marks itself as blocked, using a shared array, and waits until the parent of the team wakes it up. Waiting is achieved by spinning on a flag. However, in order not to waste CPU cycles, EEs are spinning for a while and then yield. When the parent of the team reaches the barrier, it waits until all other EEs have reached the barrier. This is achieved by just checking the shared array. When this is done, it just sets the flag to true and releases all waiting threads. This is the default ORT's barrier implementation. However, ORT gives the programmer the ability to avoid it and use his own barrier implementation, if needed.

The `critical` and `atomic` directives are treated in the exact same way by ORT. EELIB's lock routines are used to provide the necessary mutual exclusion. The compiler places an ORT call at the beginning of the code to be protected and an ORT call at the end. For `atomic` directives, the same lock is used for all atomic operations. This lock is declared and initialized inside ORT. However, this is not the case for the `critical` directive. OpenMP allows critical directives to have distinct names. For this reason, the compiler declares a global lock for each distinct critical region and passes it to ORT. The first EE entering the critical region is also responsible for the initialization of the lock.

3.5 Handling Threadprivate Variables

ORT also provides the necessary mechanisms for handling the OpenMP threadprivate variables. The `threadprivate` directive specifies that named global-lifetime variables are replicated, with each thread having its own copy. The support of threadprivate variables is not a straightforward procedure under the original thread model. This is because, global variables are by nature shared among threads. The exact opposite occurs in the process-model. All global variables are by nature process-private and we need a mechanism to make them shared.

Both the compiler and the runtime library are involved in the implementation of the `threadprivate` directive. For example, consider the following piece of code, where variables `a` and `b` are threadprivate:

```
int a, b = 1;
#pragma omp threadprivate(a,b);
void f()
{
  #pragma omp parallel copyin(b)
   a = omp_get_thread_num() + b;
}
```

The function containing the parallel code follows. Variables `a` and `b` are transformed into `tp_a` and `tp_b`, respectively. The compiler also assigns a thread-specific data key to each threadprivate variable; the `tp_a_key` is dedicated to variable `a` and the `tp_b_key` is dedicated to `b`. As specified by the POSIX standard, all threads use the same key but they can have different values associated with it.

```
1  int tp_a, tp_b = 1;
2  static void *tp_a_key;
3  static void *tp_b_key;
4  static void * _thrFunc0_(void *arg)
5  {
6    int (* a) = ort_get_thrpriv(&tp_a_key_, sizeof(tp_a), &tp_a_);
7    int (* b) = ort_get_thrpriv(&tp_b_key, sizeof(tp_b_), &tp_b_);
8     /* Copyin initialization(s) */
9     *b = tp_b;
10    ort_barrier_me();
11    (*a) = omp_get_thread_num() + (*b);
12    return(void *)0;
13 }
```

Each thread entering the `_thrFunc0_()` function must initialize its own threadprivate copies. This is because, at line 11, threads must refer to their own threadprivate variables. This is achieved by using calls to the ORT's `ort_get_thrpriv()` routine (lines 6-7). Its arguments are the key, the size and a pointer to the variable. First, each thread allocates a memory area for the variable and copies in its initial value. The thread associates this area with the compiler's dedicated key. From now on, threads can "remember" their own threadprivate copies of each variable by using only the variable's key. These memory areas are not freed until the program terminates. Consequently, threads maintain their threadprivate variables among different parallel regions in the program.

The implementation of the `copyin` clause is relatively simple. The `copyin` clause simply specifies that all threadprivate variables appearing in its list must be initialized using

the master's corresponding values, before the actual parallel execution begins. Consider the previous example. The master thread is the only thread that uses the original variables `tp_a` and `tp_b` as its threadprivate copies. This is managed internally in `ort_get_thrpriv()`. These variables are accessible by all threads, since they are global scope. In this way, the master thread initializes `b` (line 1), and all other threads maintain its value by simply accessing it (line 9). A barrier is necessary (line 10) in order to ensure that all threads have completed the threadprivate initializations before the actual execution begins.

The `copyprivate` clause needs more effort by ORT. The `copyprivate` clause appears only in the `single` directives. It provides a mechanism to use a threadprivate variable to broadcast a value from one member of a team to the other members. The broadcast is done by calling ORT's `ort_broadcast_private()`. This routine takes as its input the pointers to the thread's private variables to be broadcast. In this routine, the thread (owner) dynamically constructs an array of pointers. This array is maintained at the parent's eecb so as all threads can access it. The other thread members just call `ort_copy_private()` to copy the new values into their threadprivate variables. Each of them, accesses the owner's variables (through the pointer array) and copies them into its own threadprivate space.

3.6 The Interface with EELIB

EELIB is responsible for providing all execution entities except the master EE, plus three types of locks: normal, nested and spin locks. The first two types are made available to the programmer through the OpenMP runtime library interface, while the third type is only used internally in ORT. When execution entities are threads, EELIB has no other obligation, as everything is handled entirely by ORT. However, when execution entities are processes, EELIB's interface in slightly extended to support the new execution environment. A shared memory allocation routine must now be provided by EELIB. Also, ORT's communication subsystem needs access to some special structures held by EELIB. All these issues, will be discussed in detail in the following Chapter.

Upon initialization, EELIB announces its capabilities to ORT, which include support of nested parallelism, support for dynamic adjustment of the number of EEs, the maximum number of EEs and the maximum number of nested parallelism levels supported. Regarding the EEs, EELIB implements three functions that are called by ORT (see Figure 3.3): `ee_request()`, `ee_create()` and `ee_waitall()`. The first two are used when creating a new team. The parent asks for a particular number of EEs through a `ee_request()` call. EELIB replies with the actual number it can provide. In EELIBs that do not support nested parallelism, the number returned is always 0 when called from a level ≥ 1. If the EELIB can not provide the requested number of EEs, and the dynamic adjustment of the number of EEs is disabled, the program is forced to an early termination. Otherwise, if

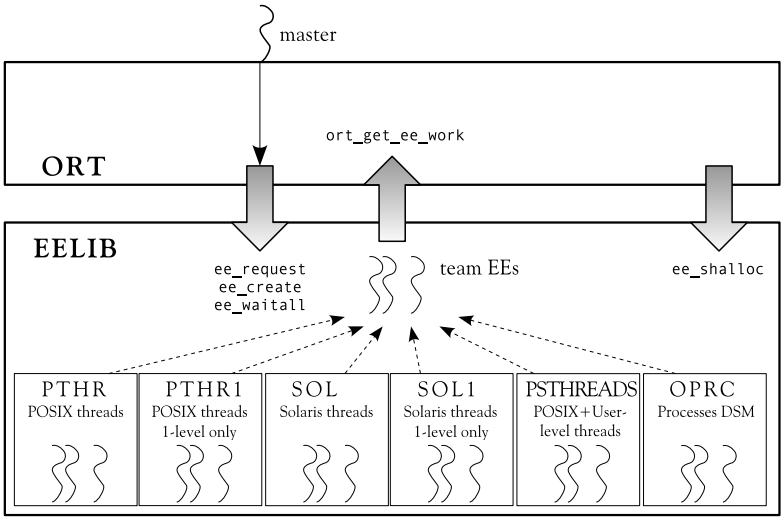

Figure 3.3: EELIBS and interface with ORT.

the dynamic adjustment is enabled, the program continues its execution with a warning. Thereafter, ORT calls `ee_create()` to instruct EELIB to actually create the requested EEs. ORT passes to EELIB all the necessary information it needs in order to create and direct the execution of the EEs, i.e. the number of EEs to be created and the function to be executed by all EEs. When an EE from the team commences execution, its very first obligation is to call `ort_get_ee_work()`, which fills in the EE's control block with the necessary information for the EE to proceed with the parallel region's execution. This routine is implemented in ORT. Upon completion of the parallel region, the master EE calls `ee_waitall()` and blocks until all other EEs in the team have finished their work.

Chapter 4

OMPi and Nested Parallelism

4.1 Nested Parallelism in OpenMP

Nested parallelism has been a major feature of OpenMP since its very beginning. As a programming style, it provides an elegant solution for a wide class of parallel applications, with the potential to achieve substantial processor utilization, in situations where outer-loop parallelism simply can not. However, even nowadays, the level of support is varying greatly among compilers and runtime systems. Even some of the proprietary OpenMP compilers do not fully support nested parallelism.

For applications that have enough and balanced outer-loop parallelism, a small number of coarse threads is usually enough to produce satisfactory speedups. In many other cases though, including situations with multiple nested loops, or recursive and irregular parallel applications, threads should be able to dynamically create new teams of threads because only a large number of threads has the potential to achieve good utilization of the computational resources. Figure 4.1 shows the classic example of Fibonacci numbers; the nth Fibonacci number is calculated recursively as the sum of the $(n-1)$th and the $(n-2)$th. In each recursive call, two threads are spawned with each one executing a `section`. As a result, the number of threads grows exponentially. If nested parallelism is not supported, speedup is limited to 2 because only two threads will be created at the first parallel region and will take the responsibility of executing all the required recursive calls.

The OpenMP specification [1] leaves support for nested parallelism as optional, allowing an implementation to serialize the nested parallel region, i.e. execute it by only one thread. In implementations that support nested parallelism, the user can choose to enable or disable it either during startup through the `OMP_SET_NESTED` environmental variable or dynamically at runtime through an `omp_set_nested()` call. The number of threads that will comprise a team can by controlled by the `omp_set_num_threads()` library call. This routine is only allowed to appear in sequential regions of code and consequently there is no way to specify a different number of threads for inner levels of parallelism. For this reason, OpenMP since version 2.0 provides the `num_threads(n)` clause. Such a clause can appear in a (nested) `parallel` directive and request that this particular region be executed by exactly `n` threads.

```
int fibonacci(int n)
{
  int f1, f2;

  if(n < 2) return 1;
  #pragma omp parallel sections num_threads(2)
  {
     #pragma omp section
       f1 = fibonacci(n-1); /* Recursive call */
     #pragma omp section
       f2 = fibonacci(n-2); /* Recursive call */
  }
  return (f1+f2);
}
```

Figure 4.1: Fibonacci numbers using nested parallelism.

However, the actual number of threads dispatched in a (nested) `parallel` region depends also on other things. OpenMP provides a mechanism for the dynamic adjustment of the number of threads which, if activated, allows the implementation to spawn fewer threads than what is specified by the user. In addition to dynamic adjustment, factors that may affect the actual number of threads include the nesting level of the region, the support/activation of nested parallelism and the peculiarities of the implementation.

According to the OpenMP specification, an implementation which serializes the nested `parallel` regions, even if nested parallelism is enabled by the user, is considered *compliant*. An implementation can claim *support* of nested parallelism if nested `parallel` regions may be executed by more than 1 thread. Because of the difficulty in handling efficiently a possibly large number of threads, many implementations provide support for nested parallelism but with certain limitations. For example, there exist systems that support a fixed number of nesting levels; some others allow an unlimited number of nesting levels but have a fixed number of simultaneously active threads. In the latter case,

a nested `parallel` region may be executed by a smaller number of threads than the one requested, if there are not enough free threads.

4.2 Enabling Nested Parallelism in OMPi

Two threading libraries are available for OMPi: a library based on POSIX threads (PTHR1) and a library based on Solaris threads (SOL1). The architecture of both libraries is identical. The only thing that changes is the type of the kernel threads used (POSIX or Solaris). Both libraries provide a single level of parallelism, i.e nested parallelism is not supported. In this section, we present a new threading library (PTHR) based on POSIX treads, which provides support for nested parallelism, while maintaining good performance levels even for the non-nested case.

We focused on the EELIB part of the runtime and managed to develop a new threading library specifically for supporting nested parallelism. The new library is called PTHR and utilizes POSIX threads. We also developed an equivalent library based on Solaris threads called `SOL` (see Figure 3.3).

4.2.1 The PTHR Threading Library

In order to provide full nested parallelism support, the PTHR library must be able to supply the requested number of threads, whenever ORT asks for it. This means that, if ORT requests for x threads at any parallel level $y \geq 1$, and the dynamic adjustment of the number of threads is disabled, the PTHR library is forced to release a bunch of x threads.

From the PTHR's point of view, this can be achieved by dynamically creating the requested threads using explicit `pthread_create()` calls. In this case, x threads will be created from scratch. Although this is a complete solution, it involves quite high bookkeeping overheads. These overheads are actually inevitable because we can not really guess the number of threads (so as to pre-create them) that ORT will eventually request upon a parallel region entrance. Even if this could be possible, the efficient management of a large number of threads has been proved to be not an easy task. For instance, time-sharing can significantly increase the implicit synchronization overheads associated with the thread management.

However, when the dynamic adjustment of the number of threads is enabled, the EELIB part is the one that decides on how many EEs it will supply to ORT. Based on this, we can still provide efficient but limited nested parallelism support. Our purpose is to limit the number of created threads. Specifically, the PTHR library pre-creates a fixed number of threads based on ORT's instructions. Whenever ORT asks for a particular number of threads, PTHR checks for available (idle) threads; these are the only threads it can

supply ORT with. Threads that finish their execution become idle. When all PTHR's threads are busy, the PTHR can not service ORT's request. In this case the (nested) parallel region is serialized.

Upon initialization, PTHR creates a pool of idle threads. The size of the pool is determined by ORT and depends on two factors: a) the `OMP_SET_NUM_THREADS` environmental variable and b) the number of the physical processors. If the user does not explicitly declare the `OMP_SET_NUM_THREADS` variable, then PTHR creates as many threads as the system's processors. The pool is actually a plain queue. Each thread is associated with a specific node in the pool, which contains thread-specific information including a flag representing the current state of thread (running or idle), the thread id within the team and the function to be executed.

Initially, the queue is occupied by threads waiting to be scheduled. Each thread waits by spinning on its own private flag. In order to avoid oversubscribing the processors, threads spin for a relatively small number of iterations and then yield the processor. Upon an `ee_request()` call, the PTHR library must inform ORT about thread availability. If it is called from level = 0, the caller is the master thread. Otherwise, it may be called by multiple threads which encounter a nested parallel region. In both cases, the requester checks the size of the pool. This is done by just reading a global counter (`plen`) which keeps the current size of the pool. If the requested number is smaller or equal to `plen` then PTHR is capable of serving the request. Otherwise, PTHR can partially serve the request with exactly `plen` threads. Before returning, `ee_request()` updates the size of the pool. Since many threads are simultaneously competing for the same global pool, the `plen` variable must be accessed and updated atomically. This is achieved by using a spin lock named `plock`. When level = 0, the use of `plock` is unnecessary, because the only running thread is the master.

The `ee_create()` call signals the start of the parallel execution. Its argument list includes the number of threads to be released (`numthr`), the function to be executed by all team members (`workfunc`) and a pointer to the team parent's eecb. The latter is used so as each thread remembers its own team parent. PTHR dispatches `numthr` threads from the pool and gives them work to do. Specifically, it traverses the first `numthr` pool elements, supplying each thread with an execution id, a pointer to the `workfunc` function and a pointer to the parent's eecb. It releases each initialized thread by simply setting its spin flag to false.

When a thread finishes the execution of the `workfunc` function, it simply rejoins the pool so as to be able to serve another request. Due to the implicit barrier at the end of every parallel region, threads rejoining the pool must somehow inform the parent of the team about their completion. This is achieved by keeping an extra field at the parent's eecb. This field is declared in ORT but is only accessible by PTHR. It is a pointer to a PTHR structure (`info`) containing two things: a) a `running` counter which represents

the number of running threads in the team and b) a spin lock for accessing this counter. A thread that becomes a parent of a team for the first time is responsible for initializing its `info` structure. Each thread after rejoining the pool, accesses the `info` structure of its parent and decrements `running` by 1. Finally, when the parent calls `ee_waitall()`, it blocks until the `running` counter becomes 0.

We can not claim that our implementation provides full nested parallelism support. This is because the PTHR library can not create new threads on the fly when the pool has ran out of threads. However, the user can ensure that enough threads will be available to serve nested parallel regions by simply setting the `OMP_SET_NUM_THREADS` environmental variable to the total desirable number. In this way, the pool will always maintain a sufficient number of threads in order to serve the program's requests.

Omni [31] handles nested parallelism in the same way; the special `OMPC_NUM_PROCS` environmental variable determines the size of the pool. In the Balder runtime library of OdinMP [20] the pool size is not fixed; it is expanded whenever it is necessary. All vendors that support nested parallelism also utilize on a pool of kernel threads. Specifically, in the Intel compiler [34], threads are not created until the first parallel region is executed, and only as many threads as needed by that parallel region are created. Further threads are created as needed by subsequent parallel regions. However, threads that are created by the runtime library are not destroyed but join a thread pool until they are called to participate in a subsequent team. In GOMP [29], the OpenMP implementation for GCC, the pool is exploited only for non-nested parallel regions, while threads are dynamically created for inner levels.

Our PTHR library, which has become the default EELIB of OMPi, although providing limited support of nested parallelism, is mostly optimized for single-level parallelism. For cases where deep nesting levels are expected, other libraries should be employed, e.g. the `PSTHREADS` [12] library. This library implements a two-level thread model, where user-level threads are executed on top of kernel threads that act as *virtual processors*. The number of the virtual processors never exceeds the number of the physical processors. Each virtual processor is a POSIX kernel thread which runs a dispatch loop, selecting the next-to-run user-level thread from a set of ready queues, where threads are submitted for execution. The primary user-level thread operations are provided by UthLib (Underlying Threads Library), a platform independent package. The `PSTHREADS` library is completely portable because its implementation is based entirely on the POSIX standard. The management of nested parallelism situations is efficiently handled by using adaptive work distribution schemes, such as thread migration.

4.3 Measuring the OpenMP Overheads

Despite the significance of nested parallelism in OpenMP, there is no research study made until now measuring the overheads associated with OpenMP constructs when nested

parallelism is in effect. Most works focus on application speedups, which give overall performance indications but do not reveal potential construct-specific problems.

The well known EPCC microbenchmark suite [4, 5] is the most commonly used tool for measuring runtime overheads of individual OpenMP constructs. However, it is only applicable to single-level parallelism. We managed to develop a set of benchmarks based on the EPCC microbenchmarks which measure the overheads of OpenMP constructs under nested parallelism. Using these benchmarks, we experimented with several freeware and commercial OpenMP compilers. The results of this section have been presented in [9].

4.3.1 The EPCC Microbenchmark Suite

The EPCC microbenchmarks are divided into two parts. The synchronization part measures the overheads of OpenMP constructs that require barrier synchronization (e.g. `parallel`, `parallel for`, `single`, etc) along with OpenMP constructs that require mutual exclusion (e.g. `critical`, `atomic`, etc). The other part is the scheduling part. This measures the overheads associated with the schedule policies of OpenMP, `static`, `dynamic` or `guided`, using a set of different configurations of the `chunksize` parameter.

The technique used to measure the overheads of the OpenMP directives, is to compare the time taken for a section of code to be executed sequentially with the time taken for the same code executed in parallel, enclosed within a given directive. Let T_p be the execution time of a program on p processors and T_1 be the execution time of its sequential version. The overhead of the parallel execution is defined as the total time spent collectively by the p processors over and above T_1, the time required to do the "real" work, i.e. $T_{ovh} = pT_p - T_1$. The per-processor overhead is then $T_o = T_p - T_1/p$. The EPCC microbenchmarks measure T_o for the case of single-level parallelism using the method described below.

A reference time, T_r, is first fixed, which represents the time needed for a call to a particular function named `delay()`. To avoid measuring times that are smaller than the clock resolution, T_r is actually calculated by calling the `delay()` function sufficiently many times:

```
for (j = 0; j < innerreps; j++)
  delay(delaylength);
```

and dividing the total time by `innerreps`. T_r is actually representing the time needed for the sequential execution. Then, the same function call (`delay()`) is surrounded by the OpenMP construct under measurement, which is in turn enclosed within a parallel directive. For example, consider the EPCC code that measures the `for` directive overheads, as shown in Figure 4.2.

```
1  testfor() {
2            ...
3   t1 = getclock();   /* start measurement */
4   #pragma omp parallel private(j)
5   {
6     for (j = 0; j < innerreps; j++)
7       #pragma omp for
8          for (i = 0; i < p; i++)
9            delay(delaylength);
10  }
11  t2 = getclock();   /* end measurement */
12 }
```

Figure 4.2: Portion of the `testfor()` EPCC microbenchmark routine.

At line 4, parallel execution begins. The created threads, which are as many as the processors, execute repeatedly the code of lines 7–9 for `innerreps` iterations. The parallel loop (line 8) has to schedule `p` iterations on exactly `p` threads using the default static schedule. That means that the loop's iterations will be equally distributed to the threads with each one of them getting exactly one iteration. Consequently, each thread will eventually execute the `delay` function for `innnerreps` times which means that each thread's work requires a total of T_r time.

The parallel execution time, T_p, is then defined as time needed to execute the whole measurement (lines 4–10), divided by `innerreps`. The overhead of the `for` directive is derived as $T_p - T_r$ since the total work done needs actually pT_r sequential time. Notice that, the measurement includes the time taken by the `parallel` directive. In order to avoid this, `innerreps` is large enough so the overhead of the `parallel` directive can be safely ignored. Each overhead measurement is repeated several times and the mean and standard deviation are computed over all measurements.

The thread/processor mapping plays a crucial role in the measurements. We must ensure that the number of threads running the parallel region of each measurement is equal to the number of present processors. This is because, we do not want to overestimate the overheads due to the time-sharing of the processors.

4.3.2 Our Methodology

To study how efficiently OpenMP implementations support nested parallelism, we have extended both the synchronization and scheduling microbenchmarks of the EPCC suite. According to our approach, the core benchmark routine for a given construct (e.g. the `testfor()` discussed above) is represented by a *task*. Each task has a unique identifier and utilizes its own memory space for storing its table of runtime measurements. We create a

team of threads, where each member of the team executes its own task. When all tasks finish, we measure their total execution time and compute the global mean of all measured runtime overheads. Our approach is outlined in Figure 4.3. The team of threads that execute the tasks expresses the outer level of parallelism, while each benchmark routine (task) contains the inner level of parallelism.

```
1  void nested_benchmark(char *name, func_t originalfunc) {
2    int    task_id;
3    double t0, t1;
4
5    t0 = getclock();
6    #ifdef NESTED_PARALLELISM
7     #pragma omp parallel for schedule(static,1)
8    #endif
9    for (task_id = 0; task_id < p; task_id++) {
10    (*originalfunc)(task_id);
11   }
12   t1 = getclock();
13
14   <compute global statistics>
15   <print construct name, elapsed time (t1-t0), statistics>
16 }
17
18 main() {
19   <compute reference time>
20   omp_set_num_threads(omp_get_num_procs());
21   omp_set_dynamic(0);
22   nested_benchmark("PARALLEL", testpr);
23   nested_benchmark("FOR",      testfor);
24        ...
25 }
```

Figure 4.3: Extended microbenchmarks for nested parallelism overhead measurements.

In Figure 4.3, if the outer loop (lines 9–11) is not parallelized, the tasks are executed in sequential order. This is equivalent to the original version of the microbenchmarks, having each core benchmark repeated more than once, due to the presence of the for loop (line 9). On the other hand, if nested parallelism is enabled, the loop is parallelized (lines 6–8) and the tasks are executed in parallel. Each thread of the first parallel level calls the corresponding measurement function (e.g. `testfor`) using its `taskid`. The number of simultaneously active tasks is bound by the number of OpenMP threads that constitute the team of the first level of parallelism. To ensure that each member of the team executes exactly one task, a static schedule with chunksize of 1 was chosen at line 7. In addition,

to guarantee that the OpenMP runtime library does not assign fewer threads to inner levels than in the outer one, dynamic adjustment of threads is disabled through a call to `omp_set_dynamic(0)`.

By measuring the aggregated execution time of the tasks, we use the microbenchmark as an individual application. This time does not only include the parallel portion of the tasks, i.e. the time the tasks spend on measuring the runtime overhead, but also their sequential portion. This means that even if the mean overhead increases when tasks are executed in parallel, as expected due to the higher number of running threads, the overall execution time may decrease.

In OpenMP implementations that provide full nested parallelism support, inner levels spawn more threads than the number of physical processors, which are mostly kernel-level threads. Thus, measurements exhibit higher variations than in the case of single-level parallelism. In addition, due to the presence of more than one team parents, the overhead of the `parallel` directive increases in most implementations, possibly causing overestimation of other measured overheads (see Fig. 4.2). To resolve these issues, we increase the number of internal repetitions (`innerreps`) for each microbenchmark, so as to be able to reach the same confidence levels (95%). A final subtle point is that when the machine is oversubscribed, each processor will be timeshared among multiple threads. This leads to an overestimation of the overheads because the microbenchmarks account for the sequential work (T_r) multiple times. We overcame this by decreasing `delaylength` so that T_r becomes negligible with respect to the measured overhead.

4.4 Assessing the Performance

Using our methodology, we experimented with a set of freeware and commercial OpenMP compilation systems. The freeware compilers are OMPi 0.9.0, Omni 1.6 and GCC 4.2.0. The commercial ones are the Intel C++ 10.0 compiler (ICC) and the Sun Studio 12 (SUNCC). For OMPi and Omni which are source-to-source compilers we chose to use GCC as the naitive back-end compiler. Also, OMPi was tested using two configurations, namely OMPi+PSTHR (PSTHREADS) and OMPI+POSIX. The latter configuration utilizes our implementation of the PTHR library.

All our measurements were taken on a Compaq Proliant ML570 server with 4 Intel Xeon III single-core CPUs running Debian Linux (2.6.6). Although this is a relatively small SMP machine, size is not a issue. Our purpose was to create a significant number of threads, which exceeds the number of available processors (4), in order to exploit the effects of nested parallelism. In the first level of parallelism, 4 threads are always created. Each one of them calls the original benchmark routine where it creates 2, 4 or 8 threads for testing a given directive. Consequently, the benchmark application creates a total of $4 \times 2 = 8$, $4 \times 4 = 16$ or $4 \times 8 = 32$ threads, respectively.

Most implementations start by creating an initial pool of threads, usually equal in size to the number of available processors, which is 4 in our case. In order to be sure that an implementation will actually create the requested number of threads in both parallel levels, we disabled the dynamic adjustment of the number of threads using a call to `omp_set_dynamic(0)`. For the case where $4 \times 4 = 16$ threads need to be created, we only had to make a call to `omp_set_num_threads(4)` upon the application startup. In all other cases, we explicitly set the number of inner threads using the `num_threads()` clause.

However, Omni and OMPi can not create more than 4 threads on the fly, even if is needed; they support nested parallelism as long the initial pool has idle threads, otherwise the nested parallel regions get serialized. To overcome this problem, in OMPi, we explicitly set the desired number of threads to be created using the `OMP_NUM_THREADS` environmental variable. In this way, the pool always maintains a sufficient number of threads to serve the parallel regions. The same thing was done also in Omni, using the `OMPC_NUM_PROCS` environmental variable. We have, however, been careful not to give those two implementations the advantage of zero thread creation overhead since with the above trick all threads are pre-created. For this reason, we include a dummy nested parallel region at the top of code, so as all implementations have the chance to create the requested number of threads before the actual measurements commence.

Our first set of results is depicted in Figure 4.4. We present the overheads of the `parallel`, `for`, `single` and `critical` directives, when 4×4 total threads are active. Each plot also includes the single-level overheads of each compilation system for reference. As we were expecting, overheads are increased when nested parallelism is in effect, mainly due to the presence of more active threads. We observe however that Intel, GCC, and Omni do not scale well in the `parallel` construct, although ICC remains quite fast. For all three of them, the runtime overhead is more than an order of magnitude higher in the case of nested parallelism. For ICC this could be attributed, in part, to the fact that threads join a unique central pool before getting grouped to teams [34]. On the other hand, both OMPi+POSIX and SUNCC clearly scale better and their overheads increase linearly, with SUNCC, however, exhibiting higher overheads than OMPi for both single level and nested parallelism.

Similar behavior is seen for the `for` and `single` constructs, except that GCC shows significant but not excessive increase. The Sun compiler seems to handle loop scheduling quite well showing a decrease in the actual overheads. This, combined with the decrease in the `single` overheads, reveals efficient team management since both constructs incur mostly inter-team contention. On the other side, Omni does not scale well in both situations. Among all, ICC and OMPi+POSIX have the smallest overheads for the single-level case, while OMPi+PSTHR has the smallest overheads, when nested parallelism is in effect. Especially in the `single` construct, OMPi+PSTHR shows the advantage of user-level threading: inner levels are executed by user-level threads, which mostly live in the processor where the parent thread is, eliminating most inter-team contention and

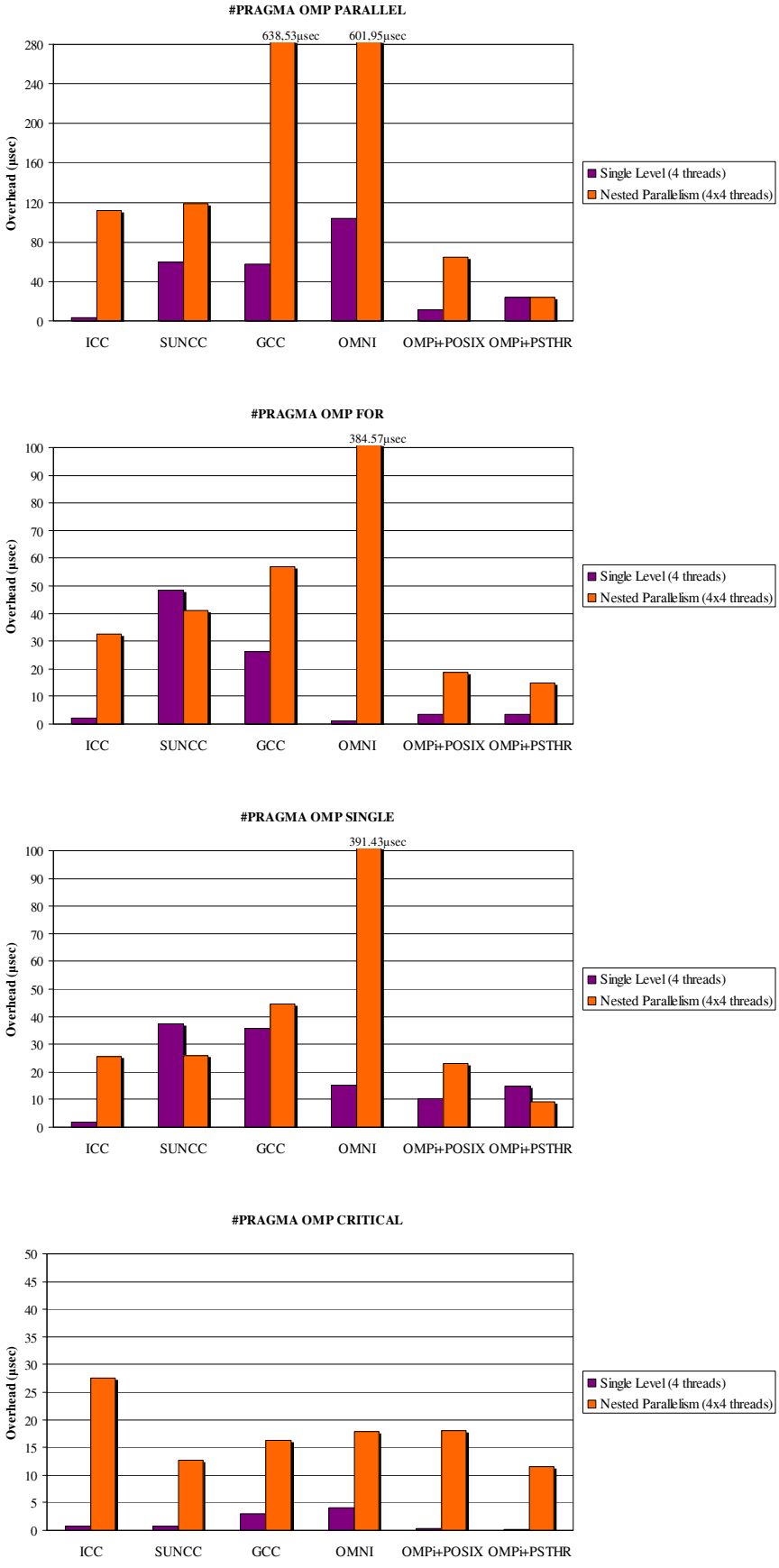

Figure 4.4: Overheads for the `parallel`, `for`, `single` and `critical`.

the associated overheads. In contrast, the (unnamed) `critical` construct incurs global contention since all threads from all teams must compete for a single lock protecting the critical code section. Overheads are increased significantly in all systems, suggesting that *unnamed* `critical` constructs should be avoided when nested parallelism is required.

Figure 4.5 includes results from the scheduling microbenchmarks. For presentation clarity, we avoided reporting curves for a wide range of chunksizes; instead we include results for `(static,1)`, `(dynamic,1)`, `(dynamic,8)` and `(guided,1)`. Schedules with a chunksize of 1 represent the worst cases, with the highest possible scheduling overhead. This is because, threads execute only one loop iteration before the compiler reschedules them for another one. Moreover, due to the nature of the dynamic and guided schedules, threads are continuously competing to gain a loop iteration. Scheduling overheads increase, as expected, for the `static` and `guided` schedules in the case of nested parallelism. The high overheads of OMPi+POSIX are mainly due the excessive locking that take place. It is expected that with the use of appropriate atomic operation primitives which are nowadays available, those overheads will disappear.

Overheads of the dynamic scheduling policy seem to increase at a slower rate and in some cases (SUNCC, GCC and OMPi+PSTHR) actually decrease, which seems rather surprising. This can be explained by the fact that for this particular scheduling strategy and with this particular chunk size, the overheads are dominated by the excessive contention among the participating threads. Recall that 16 threads need to be scheduled on 4 processors. With locality-biased team management, which groups all team threads onto the same CPU, and efficient locking mechanisms, which avoid busy waiting, the contention has the potential to drop sharply, yielding lower overheads than in the single-level case. This appears to be the case for the Sun Studio and GCC compilers. OMPi with user-level threading achieves the same goal because it is able to assign each independent loop to a team of non-preemptive user-level OpenMP threads that mainly run on the same processor. However, as the chunksize increases, jobs become coarser and any gains due to contention avoidance vanish. This case is depicted in the third plot of Figure 4.5. As the chunksize increases to 8, nested overheads increase for all implementations with respect to the single-level case.

In Figures 4.6 and 4.7 we present the results of our next experimentation: we delved into discovering how the behavior of our subjects changes for different populations of threads. We fixed the number of first-level threads to 4 but changed the second-level teams to consist of 2, 4 and 8 threads, yielding in total 8, 16 and 32 threads on the 4 processors. Because this was only possible using the `num_threads()` clause (an OpenMP V.2.0 addition), Omni was not included, as it is only V.1.0 compliant. Figures contain one plot per compiler, including curves for most synchronization microbenchmarks.

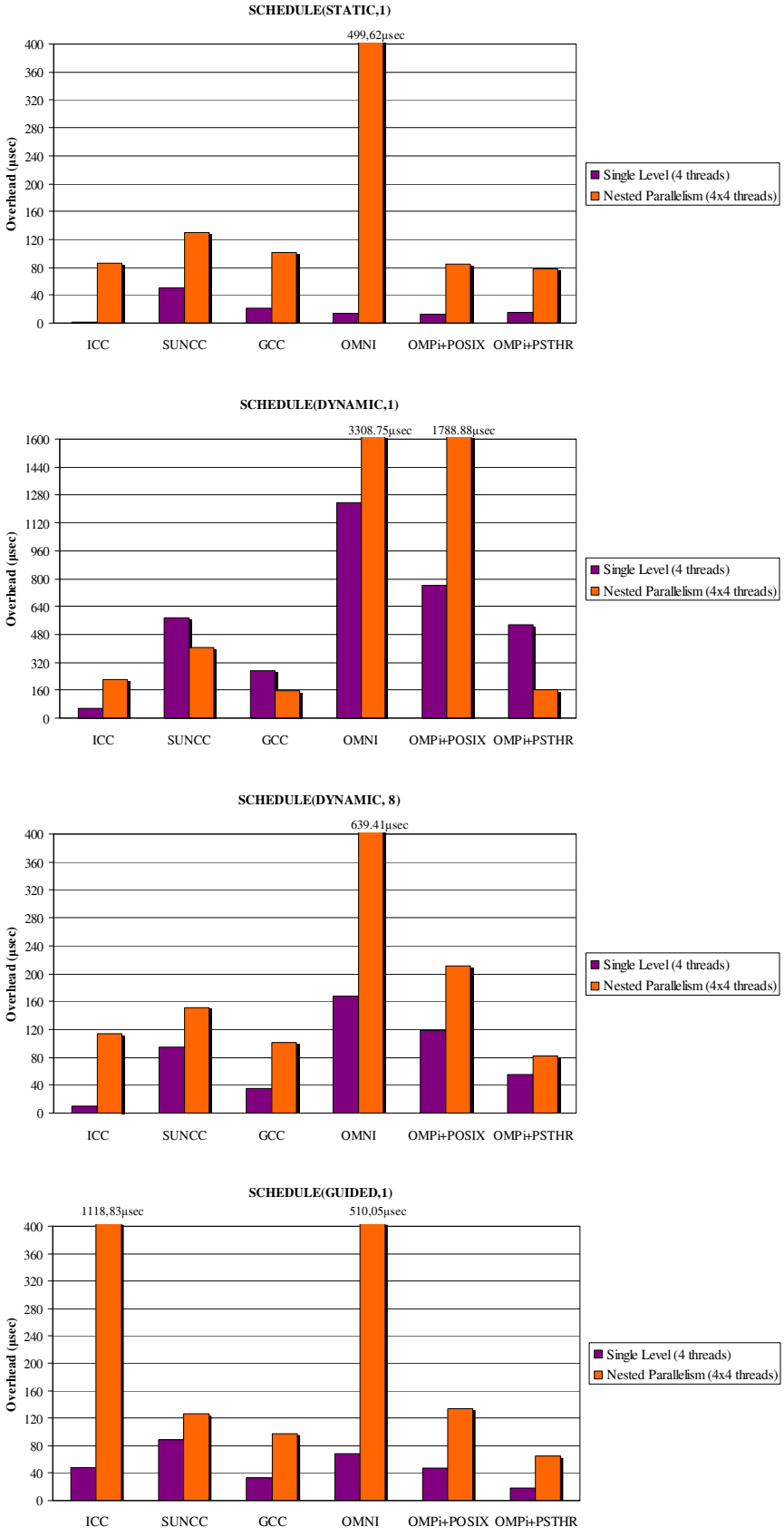

Figure 4.5: Scheduling overheads for `static`, `dynamic` and `guided`.

The results confirmed what we expected to see: increasing the number of threads in the second level leads to increased overheads. We observe that the `parallel` and the `reduction` directives exhibit exponential behavior in ICC and GCC. The latter seems that it can not handle the situation when 32 threads are present. By far, the most scalable behavior is exhibited by the OMPi+PSTHR setup, although in absolute numbers the Intel compiler is in many cases the fastest. Finally, the overheads of SUNCC on all cases are directly comparable with the ones of OMPI+POSIX, which seems to have a graceful reaction to increasing number of threads, while maintaining very low overheads for a single-level parallelism.

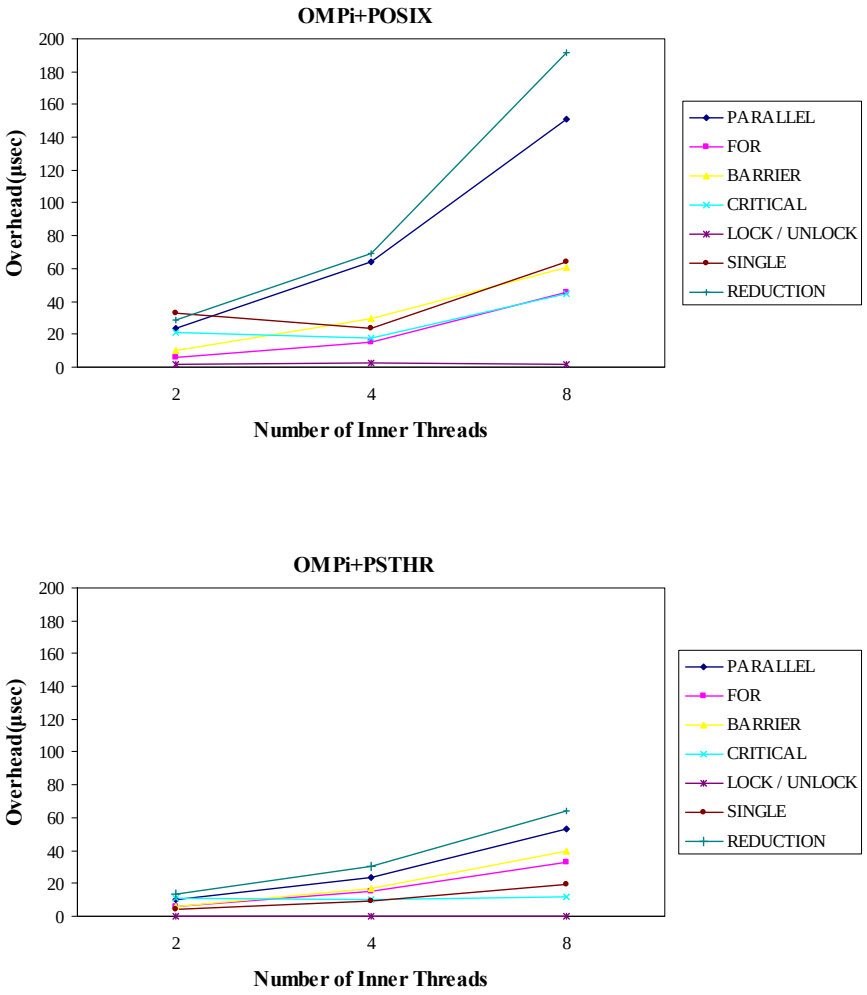

Figure 4.6: Synchronization overheads for OMPi on a different population of threads.

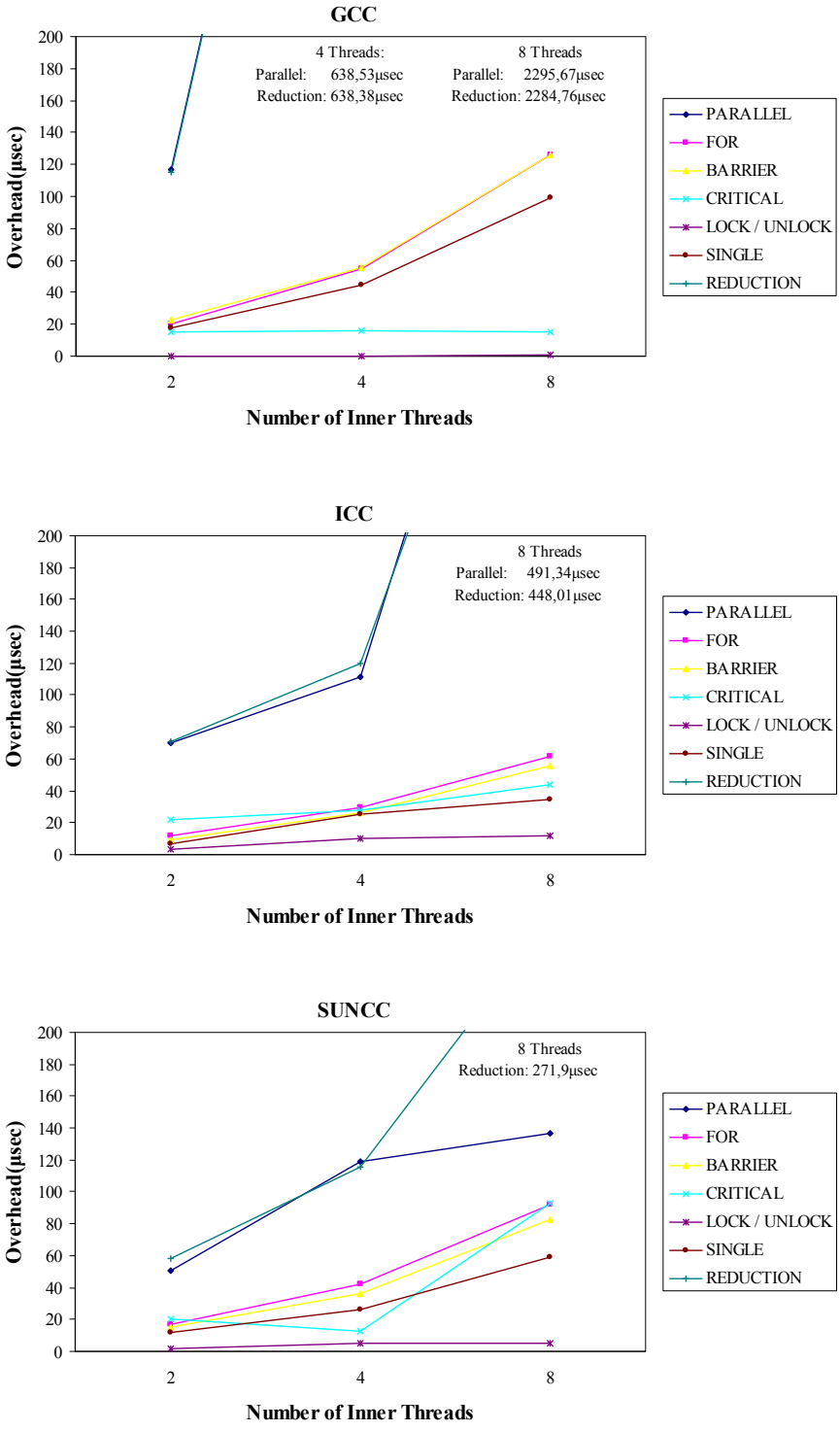

Figure 4.7: Synchronization overheads for GCC, ICC and SUNCC on a different population of threads.

Chapter 5

Shared Virtual Memory and OpenMP for Clusters

5.1 An Introduction to Shared Virtual Memory

Shared virtual memory (SVM) is a single address space shared by a number of processors in a distributed environment such as a cluster. Any participating processor has a *memory mapping manager* which implements the mapping between its local memory and the shared memory address space. Other than mapping, managers are also responsible for keeping the shared address space *consistent* at all times.

The difference between the hardware distributed shared memory systems and SVM, is that shared memory is implemented via software. Although the hardware approach has been shown to perform quite well, it incurs a high engineering cost and is usually not available in commodity systems. On the other hand, SVM is a cost-effective method for providing the shared abstraction model on networks of workstations since it requires no special hardware support and is relatively easy to implement. Application programs can use SVM just as they do on a traditional virtual memory system except that processes can run on different machines in parallel.

Traditionally, most SVM systems [15, 17, 21, 23, 24, 26] implement *page-baged* shared virtual memory. The virtual memory is partitioned into pages which can be replicated and migrated between processors on demand, just like a cache line in hardware DSM systems. In order to keep the copies of the pages synchronized, the system must supply a

mechanism to maintain coherence between them, called *coherency protocol*. The system must also provide a *memory consistency model*. While the cache coherency protocol determines *what* values should be visible to other processors, the memory consistency model determines *when* those values will be visible to other processors.

5.1.1 Page-Based SVM

A SVM system selects a portion of the virtual address space to implement the shared memory region. This space is divided into pages. The state of each shared page at any given time can be: *read-only*, *read-write* or *invalid*. Pages that are marked as *read-only* can have copies residing in the physical memories of many processors at the same time. A page marked *read-write* can reside in only one processor's memory if the coherence protocol is *single-writer* or it can reside on many processor physical memories if the SVM system implements a more advanced coherent mechanism, like a *multiple writers* protocol. A page marked as *invalid* is the result of a invalidate-type coherency protocol. The memory mapping manager views its local memory as a large cache of the shared memory address space for its associated processor, and manages it in fully associative mode at page granularity. The shared memory exists only virtually. A memory reference causes a page fault when the page is not in a processor's current virtual memory. When this happens, the memory manager retrieves the page either from the disk or the memory of another processor. If the page of the faulting memory reference has copies on other processors, then the corresponding memory mapping managers must cooperate to keep the memory coherent.

A very simple form of shared memory coherence is illustrated in Figure 5.1. In the beginning, processors P0 and P1 do not have a copy of the stippled shared page. Events occur in the order 1, 2, 3, 4. At first, P0 tries to read a page that its not present in its own local memory. This raises a page fault and control passes to the memory mapping manager (MM0). The memory mapping manager is actually a signal handler which is associated with a set of signals. The most common signal is the segmentation fault (`SIGSEV`) which is generated upon a page fault. P0 eventually obtains, through the handler, its copy of the shared page and the application process takes control again. Thereafter, P1 also requests the same page (2). A page fault occurs and its handler fetches a new copy of the same shared page from P0. The next event is a write request from P0 on the same page (3). However, the page is read-only protected causing a new page fault. The page handler of P0 knows that P1 has a copy of the page and forces it to be invalidated. P0 has now exclusive rights to the page, meaning that it can modify the page. Meanwhile, if P1 tries to access the page, a page fault will occur (4). P1's handler finds the processor which has the most up-to-date copy of that page, which is P0, and fetches a new copy.

Notice that, the physical address where the page is mapped may be completely different among the processors physical memories. Also, the handler must know or determine from where to obtain the up-to-date copy of a page or which pages it needs to invalidate before

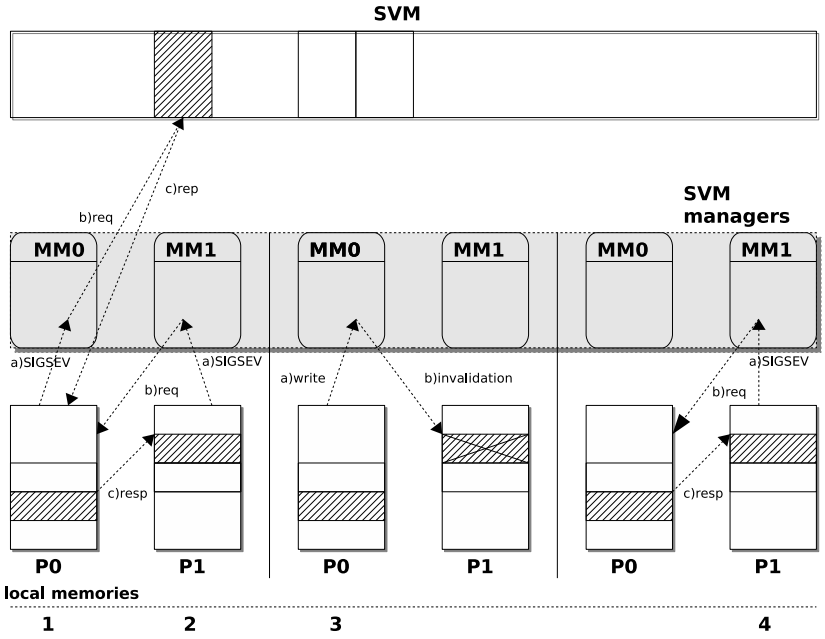

Figure 5.1: A simple form of memory coherence in SVM.

it returns the control to the application process. A page may have a home, determined by its virtual address. In this case, the handler maintains an entry for each page indicating its owner.

The main problem with paged-based shared virtual memory is the high overhead of the memory protocol invocation and processing. Page faults need time to generate the interrupt and switch to the execution of the handler. The latter is responsible for taking control and executing the memory coherency/consistency protocol, which is usually a heavyweight software implementation. In addition, the handler is also responsible for serving requests from other processors. Incoming requests slow down further the application's execution. In all cases, the invocation of the memory protocol also incurs considerable communication overheads. The large granularity of communication is a serious problem since even if a single word is needed, a whole page must be fetched.

It becomes clear that, in order to achieve good performance, SVM systems must avoid the frequent memory protocol invocations. In addition, the memory protocol itself must be efficiently designed to avoid large communication costs. For example, in a sequential consistency model, invalidations are propagated and performed as soon as the corresponding write is detected, so pages may be frequently ping-ponged back and forth among processors resulting in high delays in the application and communication level. For this reason, modern SVM systems employ more relaxed memory protocols, which delay the protocol invocation until it becomes absolute nessecary.

5.1.2 Memory Consistency Models in SVM

As we have already mentioned, the memory consistency model determines when the modified pages will be visible to other processors. A memory consistency model of a shared memory system formally specifies how the memory system will appear to the programmer. Essentially it defines constrains on the order in which memory accesses can be performed in shared memory systems. The stricter the memory consistency model, the easier for programmers to program, and the smaller the opportunity for optimization. Strict memory consistency models like sequential consistency result in a serious performance degradation in SVM. *False sharing* is a situation where multiple processors request for the same page but write different locations in it. In the sequential memory consistency model, a write operation on a shared page causes the coherence protocol to immediately invalidate all of its copies. If processors simultaneously write on the same page, even if they write on different locations, the page will be ping-ponged back and forth resulting in a high communication cost.

Although the memory consistency model specifies when coherence operations and data need to become visible, it can actually be implemented with various degrees of "laziness". Greater laziness implies greater complexity of the protocol, but fewer communication and protocol operations. In order to improve the performance of SVM systems, one of the most affective method is to relax the memory consistency model. Relaxed memory consistency models allow the propagation of the modified pages to be postponed until synchronization points, greatly reducing the impact of false sharing and the frequency of protocol operations.

A multitude of relaxed memory consistency models have been presented in the past. For example, TreadMarks [23] uses the *Lazy Release* memory consistency model while JIAJIA [15] uses the *Scope* memory consistency model (scC) [16]. In *Lazy Release* consistency, the propagation of the modified pages is delayed until a synchronization point is reached, i.e a barrier or a lock-acquire operation. When a process reaches the barrier it gets informed about which shared pages were modified since the last synchronization occurred. In the same way, when a process acquires a lock it gets informed for the modified pages, by the last process that released the lock. ScC is based on *consistency scopes* which are limited views of memory with respect to which memory references are performed. That is, modifications to data performed within a scope are only guaranteed to be visible within that scope. A consistency scope consists of all critical sections protected by the same lock. Additionally, barriers define a global consistency scope which includes the entire program. Any modifications made within a scope session become visible to processes that subsequently enter new sessions of that scope (acquire the lock or call a barrier). Modifications made outside the scope session are not guaranteed to be visible.

In general, all relaxed memory consistency models are variations of a general model rather than new models. From the user's point of view, the programming interface is closely

tied to the memory consistency model adopted by the SVM system. For example, in a SVM system with sequential consistency like Mome [17], the programmer can write the program just like he would do on a traditional shared memory system. However, in a relaxed model, the user must rely on the use of synchronization operations to enforce memory consistency. Moreover, the programmer must be aware of all the details of the underlying memory model. For example, programs that target JIAJIA, also run correctly with TreadMarks. However, the opposite is not true. JIAJIA uses the ScC model, which is slightly lazier than the Lazy Release model adopted by TreadMarks. Consequently, while relaxed models are more efficient than stricter ones, there is the trade-off of programming complexity. In any case, programming for shared virtual memory remains a simpler task than using explicit message passing techniques, like MPI.

5.1.3 Cache Coherency Protocols in SVM

The presence of multiple cached copies of a shared page requires a mechanism to notify other sharers of a modified memory location. There are two main categories of cache coherence protocols: *write-invalidate* and *write-update*. In the first category, a process writing a location in a shared page first invalidates all existent copies. When a remote process tries to access the invalidated page it generates a page-fault and its handler fetches the up-to-data copy from the writer. In the write-update category, the writer immediately supplies all processes with the modified pages, allowing them to create an up-to-date copy.

The cache coherency protocol is tightly related to the memory consistency model. Most SVM systems employ more complex coherence schemes. For example, TreadMarks and JIAJIA use the *multiple-writers* coherence protocol combined with a write-invalidate method. ParADE [21] uses the same protocol combined with a write-update method. By this protocol, multiple processes can write on the same page or on different pages simultaneously. This combined with the relaxed memory consistency model employed, greatly reduces false sharing and application delay. Each process modifying a page, first creates a *twin*. A twin is a replica of the page to be modified. After modifying the page, the process calculates a *diff* comparing its twin and its modified page. This diff its an encoding representing the changes that the process is responsible for. Upon a lock release or a barrier, processes send invalidation messages regarding the pages that they modified by the time after the last synchronization occurred. This causes the processes acquiring the lock or entering the barrier to invalidate their corresponding copies. Subsequently, when a process tries to access an invalid page, a page fault occurs. In TreadMarks, the process fetches the corresponding page and applies its own diff and all received diffs from the other processes that also modified this page. With the exception of the first time a processor accesses a page, each copy of that page is updated exclusively by applying diffs; a new complete copy of the page is never needed. In JIAJIA, the same approach is followed except that the home node of page receives and applies the diffs into the page. When processes request this page, the home node supplies them with the up-to-date copy.

In general, the coherence protocol must deal with three important questions: (a) *how to implement locks*, (b) *how to implement barriers*, and (3) *what to do when the access fault occurs?*. The answers depend on what memory consistency model and what coherence protocol are used. There is no standard regarding which memory coherency protocol should be always used. Some systems implement more than one coherency protocols and give users the choice of the most suitable protocol for their applications.

5.1.4 Memory Organization Methods

The memory consistency model and the coherency protocol determine the algorithm and the data structures to implement a SVM system. However, there is one more issue to be taken care of: *the management of the shared virtual address space*. In general, there are two methods for organizing shared virtual memory.

The first method organizes the shared virtual address space as a *cache-only memory architecture* (COMA), where all local memory of each node is treated as a large cache, and pages can be replicated or migrated on demand. TreadMarks uses this method. Shared pages are usually kept at the same virtual addresses on every processor's local memory. Each page has an owner, and a mechanism is used to find where the owner of the faulting page is when a page fault occurs. However, owners do not remain static; a page owner may migrate unexpectedly.

The second method organizes the shared virtual memory in a *non-uniform memory access* (NUMA) way. Each page has a fixed home and when a page fault occurs, the faulting processor can fetch the up-to-date page from the home directly. JIAJIA and Mocha [24], which is an improved version of JIAJIA, belong into this category. In JIAJIA and Mocha, each page has a home and homes are distributed across all nodes. References to remote shared pages cause these pages to be fetched from its home and cached locally. By the use of a cache mechanism, the size of the shared space can be as large as the sum of each machine's local memories, in contrast with TreadMarks where each local memory has to maintain a sufficient space for all shared pages. ParADE uses a hybrid approach, where the home of a page can migrate based on statistics. Specifically, for each shared page, it counts the number of page faults occurred. When this number is large enough for a particular process, it chooses that process as the page's home.

5.1.5 Application Programming Interface

All SVM systems allow the allocation of global memory and the transparent access to these globally shared memory segments. In addition, they provide a set of synchronization operations which can be used to coordinate the distributed tasks and to achieve a reliable program execution. However, the API is varying among different SVM systems. In the simplest case the routines are simply named differently, but in most cases they also have slightly different semantics. A typical example is whether the memory allocation is local,

i.e. allocated by a single node, or global operation, i.e. requires the participation of all nodes. For example, in TreadMarks a process allocates a shared memory area by a call to the `Tmk_malloc()` routine and distributes the memory information to all other processes by a call to the `Tmk_distribute()` routine. Upon synchronization, all remote processes will be informed of the new memory segment. In contrast, JIAJIA uses the global approach. All nodes must call the `jia_alloc` routine for the allocation to complete. In this case, an explicit distribution operation is not needed. In Mome [17], both local and global allocation routines are implemented. This difference can be lead to several code changes when porting from one API to another.

5.1.6 SVM for Clusters of SMPs

Early SVM systems assumed uniprocessor nodes, thus allowing only one thread per process on a node. Currently, commodity off-the-shelf microprocessors and network components are widely used as building blocks for parallel computers. This trend has made clusters of symmetric multiprocessors attractive platforms for high performance computing. However, the first generation SVM systems are too restricted to exploit multiprocessor nodes in the cluster. The next generation of SVM systems are aware of SMP nodes and exploit them by means of multiple processes or threads per-node. In general, the most common approach is the use of multiple threads, so nothing need to be done to provide memory consistency among the threads in a node. This also boosts performance because a page fetched by a thread as a result of a page fault is by nature visible to all the other threads within the process. The programming model is now hybrid with pure shared memory for intra-node communication and distributed shared memory for inter-node communication.

As far as the SVM system is concerned, the memory protocol needs to be carefully designed. The conventional page fault mechanisms will fail in multithreaded environments because multiple threads may try to access the same page while a thread is performing a page-update procedure. On the first access to an invalid page, the system will set the page writable in order to replace it with a valid one. Unfortunately, this change will also be visible to all application threads which will not rise a page fault when accessing the writable page and continue with garbage data. This situation is known as the *atomic page update* problem. The most obvious solution is to block all threads until the page-update is completed. However, this is not an efficient solution because threads will stop their execution even if pages are unrelated to them. In [22] the authors present 3 techniques for efficiently handling the atomic page update problem.

5.2 OpenMP and Shared Virtual Memory

Many researchers have proposed methods for extending OpenMP to clusters. A typical design of such a compiler includes a translator and a runtime system which utilizes a

particular SVM system [8, 13, 14, 21, 25, 31]. Through the latter, the compiler is capable of providing the shared memory model required by OpenMP, within a distributed environment.

5.2.1 Shared Variables

A major problem arising when moving to a cluster is that of variables visibility. Global variables are no longer shared among the system's processes. Also, stack variables that need to be shared inside a parallel region, due to the presence of a `shared` clause, need also special treatment. Some OpenMP systems overcome these difficulties by following an *everything shared* approach. By this approach, each process's entire address space is allocated in shared memory. In this way, global and stack variables are visible by every process. Nanos follows this approach [8]. Other compilers are based on translator instructions. The translator puts explicit calls to the runtime system regarding global variables that need to be allocated in the shared space. As already described in Section 2.4, Omni and OMPi follow this approach. OMPi handles the stack variables that need to be shared by letting the initial (master) process run on a shared stack. In order to support nested parallelism, all processes should run also on shared stacks. In Omni [31], stack variables that need to be shared inside a parallel region are copied into a shared memory area right before the parallel execution begins and are copied back into their original memory addresses after the parallel region ends. On the other hand, Intel [34] introduces a special directive named `intel_omp_sharable` for explicitly declaring global data that need to be shared. However, this approach requires applications to be modified in order to run correctly on a clustered environment.

5.2.2 Memory Consistency

In clusters, memory consistency is no longer handled exclusively by the underlying hardware. Instead, the SVM system is responsible for providing a consistent view of the shared data. Most SVM systems exploit relaxed memory consistency models which have major semantic differences with the models adopted in hardware shared memory architectures. These differences must be well hidden from the application programmer. Fortunately, OpenMP assumes a very relaxed memory consistency model. The `flush` directive is the only OpenMP directive which enforces a memory consistency operation to take place. In most cases, a flush operation is directly mapped to the corresponding synchronization operation required by the SVM system. For example, in TreadMarks a lock/unlock sequence is enough to provide the memory consistency needed. In JIAJIA, which employs a lazier memory model, a lock/unlock sequence is not enough to provide global memory consistency. In this case, a barrier operation must be performed. Consequently, the implementation of the `flush` directive is closely related to the memory consistency model used by the underlying SVM system.

5.2.3 Performance

In general, the performance of OpenMP systems utilizing shared virtual memory is not satisfactory. Frequent and costly page faults result in a significant performance degradation. Researches have shown that applications exploiting fine-grain parallelism do not perform well on these systems. For this reason, researchers have focused on finding ways of reducing the overheads associated with shared virtual memory. A way of improving performance is to avoid shared virtual memory by using explicit communication techniques, whenever possible [11]. For example, communications at the runtime library can be efficiently managed through MPI rather than through shared variables. In this way, shared virtual memory is only used for managing the program's shared data. Further improvements include optimizations for efficient distribution of the shared data among processes [8, 28]. Data locality is a major factor affecting an application's performance. If processes maintain locally most of the needed pages, page faults will occur rarely. The presence of multiple threads per-process can also result in overall performance increase in clusters of SMPs [11, 14, 21].

Chapter 6

OMPi and Clusters

6.1 A Modular Architecture

For the execution of OpenMP programs on top of clusters, we have developed a new EELIB module for OMPi, called OPRC. An SVM system is responsible for providing the shared memory abstraction needed by the OpenMP application. Our runtime system allows arbitrary SVM cores to be integrated into OMPi by decoupling the SVM core from the rest of the runtime system. OPRC makes arbitrary calls for shared memory allocation or synchronization without really knowing which SVM core is the actual target. We have managed to experiment with OMPi by using a number of different SVM systems: TreadMarks [23], JIAJIA [15], Mocha [24], ParADE [21] and Mome [17]. All but Mome use relaxed memory consistency models. Mome's memory consistency model is based on sequential consistency. The work of this chapter was presented in [30].

With OPRC's architecture, the incorporation of a new SVM system into the runtime library of OMPi is a straightforward procedure. For each candidate SVM system, we develop a C module containing all the OPRC routines that must be implemented with the help of the corresponding SVM core. Specifically, shared memory allocation and synchronization routines are implemented in this module and target the SVM core. We

have developed five different C modules, one for each SVM system (see Figure 6.1). For example, whenever the runtime library makes a generic call to `oprc_shmalloc()`, it is being translated into a `Tmk_malloc()` call if OMPi was configured with TreadMarks or into a `jia_alloc()` call if OMPi was configured with JIAJIA. Moreover, each of these modules implements the memory fence mechanism required by the OpenMP `flush` directive. The memory fence operation is tied to the SVM system's specific memory protocol. Usually, in relaxed memory consistency models, the fence operation is translated into a barrier operation or a lock/unlock sequence.

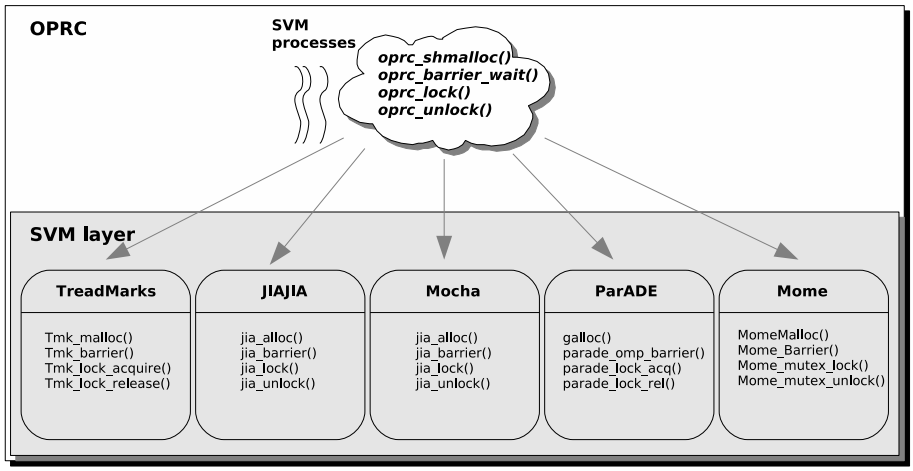

Figure 6.1: The OPRC library and its interaction with SVM systems.

As discussed in Section 5.1.5, some SVM systems require the memory allocation to be global, i.e. executed by all processes. Other systems require the allocation to be local, i.e. executed by exactly one process and distribute the result to the others. In our approach, all processes call a generic `oprc_shmalloc()` routine which is eventually mapped to the specific SVM system's allocation routine taking into consideration the allocation policy. For example, if the target SVM system is TreadMarks, only process 0 will eventually call the `Tmk_malloc()` routine and distribute the memory using the `Tmk_distribute()` routine.

6.2 A Hybrid Approach

Assuming the original thread model, execution entities are able to communicate with each other by exploiting the underlying physical shared memory. Communication at the runtime library is achieved by simply reading or modifying global variables. However, this is not the case when execution entities are processes. Memory is now distributed among the nodes in the cluster. One way of achieving inter-process communication is to exploit the SVM system's shared memory provision. All ORT and OPRC structures that need to be process-shared (e.g. ORT workshare specific structures) are explicitly allocated in the SVM system's shared memory. As a result, processes are treated in the

exact same way as threads. However, shared memory is no longer provided by hardware so additional synchronization operations must be incorporated into the code, in order to enforce memory consistency. All application's shared data (e.g. global variables) are handled also by the SVM system using the technique presented in Section 2.4.

Although this approach seems appealing, the performance is rather poor. The SVM system has to handle a possible large number of pages for the application's shared data along with pages related only to the runtime system's shared structures. Consider a barrier operation performed by OPRC in order to enforce consistency in its shared structures. All page modifications will be propagated to the processes including page modifications caused by the application even if the user has not explicitly requested a memory consistency operation. Moreover, frequent inter-process communication at the runtime level will result in frequent page faults. Whenever a page fault occurs, the application is suspended and the page handler is invoked. Consequently, the application is burdened with considerable overheads which are due to the runtime system.

A more efficient approach is to disassociate the SVM system from the runtime library's communications. Communications needed by ORT or OPRC can be efficiently handled by explicit message passing, using for example MPI. All communication patterns in both ORT and OPRC are well known at their design phase, in contrast to the application's data access patterns which are hard or even impossible to guess at compile-time. In our design, both ORT and OPRC communications are efficiently handled via MPI, while the application's shared data are handled via the underlying SVM system.

6.3 The OPRC Library

The control of the application's startup is moved to OPRC by renaming the application's `main()` function into `ompi_original_main()` and declaring a `main()` function inside OPRC. Note that, `main()` is called by all processes since all of them run the same executable. The first routine invoked is the SVM system's specific initialization routine. All processes are initialized and each one of them gets a distinct id. The master process (home) has id 0. As described in Section 2.4, the master process must somehow run on a shared execution stack. The `makecontext()`, `swapcontext()`, and `getcontext()` C library routines allow us to create a user-level thread and explicitly declare its stack memory area. We create a user-level thread (through `makecontext()`) which has its stack allocated by the SVM system's allocation routine. Process 0 is then switched to this user-level thread, and thus the desired effect is achieved. The process now runs on a shared stack and stack variables will be automatically allocated in shared memory. The new user-level thread begins its execution by calling the application's original main (`ompi_original_main()`). An alternative method would be to create a kernel-level thread (e.g. POSIX) and explicitly declare its stack to be shared. However, this would result in

two kernel-level threads with the one of them having no real work to do other than just spending computational resources while waiting for the other thread to finish.

However, keeping the master process stack in shared memory causes two problems. The first problem is that the process's signal handler also runs on this shared stack. This is quite dangerous, because the handler may modify pages that are invalid. This would cause page faults inside the handler. For this reason, right before switching to the user-level thread, we declare an alternative signal stack for the handler's execution, allocated in private memory this time. This was achieved by using the `sigaltstack` facility and forcing the handler to use this stack for the execution of the received signals (e.g. SIGSEV).

The second problem is closely related to the first. When the master process tries to access an invalid page, a page fault occurs. The signal handler receives the SIGSEV signal and invokes the memory protocol to fetch the up-to-date page. The handler writes some information to the process's stack in order to resume the application's execution right after the page request is served. What happens if the handler tries to write this information to the same invalid page which contains the data? In this case, a page-fault will also be raised inside the handler.

One way to avoid this problematic scenario is to ensure that shared data are far away in pages from the current execution pages. This can be achieved by using dummy "paddings" of size equal to the page size right after the declarations of the stack variables. In this way, we ensure that the current execution page does not contain shared data. Although this approach works, we choose to do something different: the master process runs always the work function on a private stack. Right before the parallel execution starts, process 0 switches back to the original private stack. All process's stack variables that may need to be shared inside the parallel region, are already residing in the shared stack and are accessible by all remote processes. When the process finishes its work, it assumes again the shared stack. The overhead of changing stacks is negligible with respect to the overall overheads due to the use of efficient user-level context switching.

6.3.1 OPRC Initialization

All processes start by calling the `oprc_initialize()` initialization routine. Like the others EELIBs of OMPi (e.g. PTHR), OPRC announces its capabilities to ORT, which include support of nested parallelism, the maximum number of processes and the support for dynamic adjustment of the number of processes. In its current version, OPRC does not support nested parallelism. The maximum number of processes available to ORT is limited by user parameter given at the command line upon execution request. That means that new processes can not be created on the fly. The dynamic adjustment of the number of processes is enabled by default.

Thereafter, each process initializes its own *pcb*; a control block containing process-specific information such as the process's execution id, the number of owned locks, the execution id of the team's parent process and a thread descriptor. A SVM system usually provides a number of locks. These are usually plain integer numbers to be used in the lock routines. These numbers are uniformly distributed among processes and each one of them keeps a counter of its active locks. The thread descriptor points to an extra kernel-level thread created by each process, which is called *server-thread*. All but process 0 then call an OPRC internal routine, named `wait_for_work()`, waiting for actual program execution.

6.3.2 The Server Thread Model

In our design, each process creates a *server thread* upon initialization. The server-thread is a POSIX kernel-level thread. Its main duty is to listen for incoming requests generated by remote processes or by its own host process. From now on, processes executing the application's code will be referred as application threads. Consequently, each node of the cluster maintains an application thread and a server thread. The communication between the application thread and its server is achieved by utilizing a local queue called *event-queue*. Specifically, the server thread inserts the received request into the event-queue in order for it to be served by the application thread. The most important requests each server thread can receive are the `PARALLEL`, `FINALIZE` and `ORT` requests. The first one signals a parallel execution event and targets a remote group of server threads. The `FINALIZE` event is generated by the home process upon program termination and targets all system's server threads. Finally, the `ORT` event is generated by an application thread requesting ORT shared data and will be described in Section 6.4.2.

6.3.3 Executing a Parallel Region

In single-level parallelism, the home application thread executing the sequential part of the application makes a call to `oprc_create()` whenever it encounters a `parallel` directive. Its arguments include the size of the team, say n, and the function to be executed by all team members. The home application thread generates a `PARALLEL` request which targets the first n remote server threads. (see Figure 6.2). An MPI message containing all the parallel region specific information (e.g. work function, parent's pid, etc) is constructed and is sent to the n server threads (1). Each server receiving the `PARALLEL` request immediately forwards it to the application thread by inserting it into the local event-queue (2). The application thread checks the event-queue on a regular basis looking for new events. By the time it receives the `PARALLEL` request (3), it immediately starts execution.

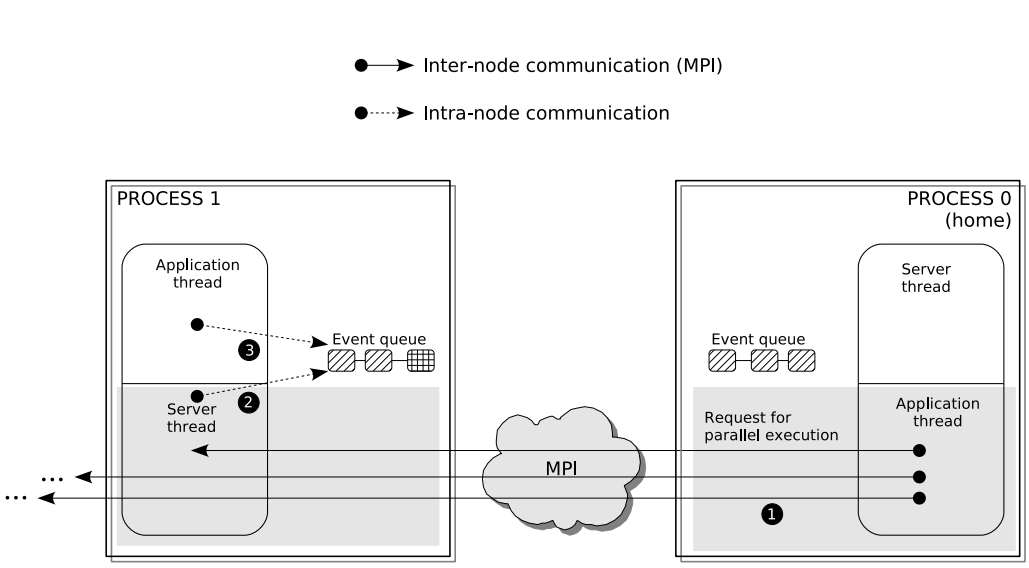

Figure 6.2: The series of events upon a parallel execution request.

6.3.4 Synchronization

Regarding barrier operations, we considered two design choices: a) use the MPI barrier routine (`MPI_Barrier()`) or b) use the barrier routine supplied by the SVM system. In most cases, OpenMP includes a memory fence (`flush`) operation at synchronization points. For example, the `barrier` directive which provides synchronization among application threads implies a memory fence operation. As already mentioned, SVM systems based on relaxed memory protocols provide memory consistency at synchronization points. Moreover, usually a barrier operation enforces global memory consistency. That is, all shared memory modifications made since the last synchronization occurred, are propagated to the application threads. Consequently, the barrier itself contains a memory fence operation. So, in our approach every call to a barrier operation is directly mapped into a call to the SVM system's barrier routine. However, we can not guarantee that all initially created processes will execute the barrier. The user can explicitly set the size of the parallel team through a `omp_set_num_threads()` call or through the use of the `num_threads()` clause. If the requested number of processes is smaller than the total number of processes in the system, the barrier will block waiting for all processes to arrive. To overcome this problem, whenever a barrier operation is performed, we force all possible idle processes to execute the barrier by sending a `SYNCHRONIZE` request to their server threads.

In order to provide consistency during lock operations, locks are also handled by the SVM system. These include locks utilized by ORT or application-level locks declared and used by the programmer. SVM locks are usually plain integer numbers. We only have to ensure that these integers are kept in shared memory so as to be readable by all processes.

6.3.5 Finalization

The last OPRC function called is `oprc_finalize()`, upon program finalization. The home application thread sends the `FINALIZE` event to all server threads including its own server thread. Each server receiving the event forwards it to the local application thread and terminates immediately. All processes are then terminated by calling the SVM system's finalization routine. The master process switches back to the original private stack, releasing the shared stack memory area right before termination.

6.4 Managing ORT

ORT maintains data that need to be accessible by all. For example, all scheduling information presented in Section 3.3 is stored in the team's parent control block (eecb), and all team members need to have access to it. Moreover, the OpenMP environmental variables declared by the application programmer need to be process-shared. Normally, allocating the parent's eecb and the structure holding the environmental variables in a shared memory area allocated by the underlying SVM system is enough for correct ORT execution. However, as we already discussed, this is not an efficient solution, due to performance issues. For this reason, we employed MPI for implementing the shared memory abstraction. ORT shared data reside in the home node. An application thread that needs to access the data, generates a request to the home's server thread. An access to ORT shared data deals with a small set of variables. In most cases, a simple increment or assignment operation is applied to a variable. In the original thread model, these accesses are protected by locks to ensure atomicity. In our case, a write operation is by nature atomic because a server thread services one request at a time.

We also simplified the management of the workshare regions by avoiding the utilization of the workshare queue described in Section 3.3. Specifically, when execution entities are processes, all workshare regions are only blocking. Despite the limitation introduced, in this way we avoid the communication overheads of managing the queue via MPI messages. The same approach is also followed by Omni.

6.4.1 ORT Initialization

The first routine called in ORT is `ort_initialize()`. The master process reads the OpenMP environmental variables and sends their values to all other remote processes using an MPI collective message. Server threads are not involved here. Thereafter, all processes call the `ort_share_globals()` routine. By this function, all application's global variables are reallocated in shared memory. As described in Section 2.4.1, for each global variable, a call to the ORT's `ort_sgvar_allocate()` is inserted by the parser at the generated file. By this routine, a list containing all the application's global variables is constructed. Each node of the list contains a pointer to the variable, the variable's size and

initial value. In `ort_share_globals()`, a shared memory area of size equal to the total size of the application's global variables is allocated using the SVM system's allocation routine. Variables are then mapped in this memory area.

6.4.2 ORT Communication Scheme

ORT shared data include the structure holding the environmental variables and the team parent's eecb. We only support single-level parallelism, so the parent of the team is always the home application thread. All processes read and write ORT shared data by moving data across the nodes using the underlying network. Environmental variables are rarely accessed by the processes and usually only for reading, in contrast with worksharing specific data which is frequently accessed and modified inside worksharing regions. These structures are stored in parent's eecb.

A simple example showing the communication steps upon a read request is illustrated in Figure 6.3. Application thread 1 makes a request to its local server thread (1). The request specifies the type of the shared data that it needs to read. The server thread forwards the request (through an MPI message) to the corresponding node maintaining the original data (2). This is the home node in our case. The home server thread is responsible for serving the request. A reply MPI message containing the corresponding data is sent directly to application thread 1 (3). Upon a write operation, the application thread generates a request which includes the modifications to be done. The home server thread is responsible for applying them to the original data.

Although we only support single-level parallelism, the above design can also work in nested parallel regions. Upon a new (nested) parallel region, all server threads are notified about the identity of the team parent. Consequently, the local server thread will know where to redirect a read/write request.

6.5 Experimental Results

In this section, we present representative experiments on a SVM cluster system. In all of our experiments, we tested two OpenMP platforms: OMPi+OPRC and an evaluation copy of the Intel 10.0 compiler with cluster OpenMP support [13]. Specifically, OMPi was tested using a number of different SVM cores (see Figure 6.1), while the Intel compiler (ICC) was configured using the default values. All experiments were performed on 8 nodes of a HP XC cluster system. Each node has 2 AMD Opteron 248 processors running Linux 2.6 and 4 GB main memory, while the nodes are interconnected with Gigabit Ethernet. The MPI library used in our experiments for communication and application launching is MPICH2 (1.0.6).

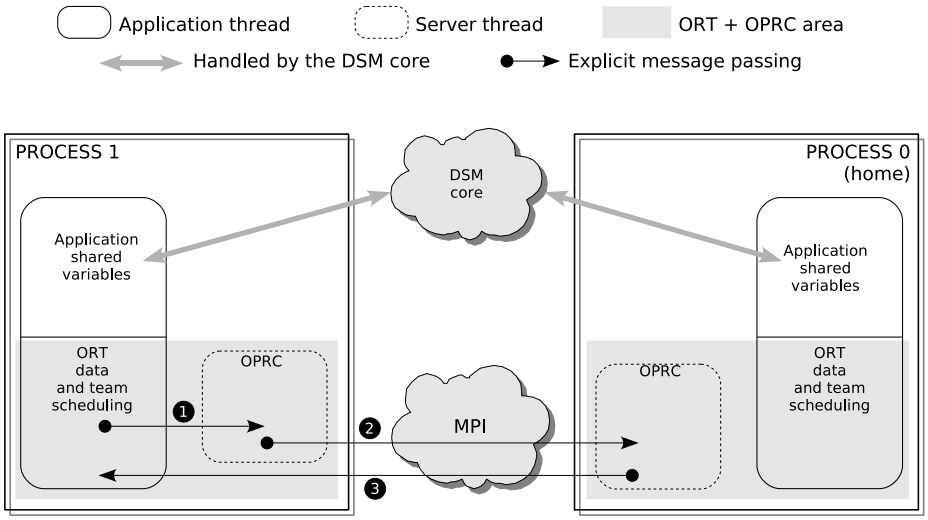

Figure 6.3: Communication steps followed upon a read request.

In Section 6.5.1, we present results for the EPCC microbenchmarks. In Section 6.5.2, we present the speedups gained when running a set of known parallel applications on an increasing number of nodes in the cluster.

6.5.1 EPCC Microbenchmarks

Our first experiment was to execute the EPCC microbenchmarks on a varying number of nodes in the cluster. For OMPi, the microbenchmark codes were executed without any modifications to the source code of the them. From the other hand, we had to explicitly insert a specific directive (`#pragma intel omp sharable`) for the management of global variables that need to be shared, in the case of the Intel compiler. For presentation clarity, we avoid reporting measurements of all EPCC microbenchmarks. Instead, we present results for the `parallel for`, `single` and `parallel reduction` directives. Also, we choose to present measurements for OMPi targeting Mocha and Mome. The former is a SVM system which is based on a relaxed memory consistency model (scope consistency) while the latter is based on sequential consistency. The behavior of OMPi targeting TreadMarks, JIAJIA or ParADE was similar to that of OMPi+Mocha, because all of them exploit similar relaxed memory consistency models.

Table 6.1:Overheads for `parallel for` (μs)

Compiler	**2 nodes**	**4 nodes**	**8 nodes**	**4 nodes $\times$ 2 threads**
ICC 10.0	905.86	1048.21	1205.84	1388.64
OMPi + Mocha	784.79	1051.65	1437.44	-
OMPi + Mome	491.09	834.39	1295.15	-

Tables 6.1, 6.2 and 6.3 summarize our results. Measurements regard overheads when running the EPCC codes on 2 nodes, 4 nodes, and 8 nodes of the cluster. In all cases,

Table 6.2:Overheads for `single` (μs)

Compiler	2 nodes	4 nodes	8 nodes	4 nodes × 2 threads
ICC 10.0	674.22	720.37	750.84	1242.81
OMPi + Mocha	315.78	578.93	773.82	-
OMPi + Mome	210.53	488.90	801.19	-

Table 6.3:Overheads for `parallel reduction`(μs)

Compiler	2 nodes	4 nodes	8 nodes	4 nodes × 2 threads
ICC 10.0	1527.18	3610.28	6362.96	5228.18
OMPi + Mocha	1151.47	2389.92	4729.03	-
OMPi + Mome	1065.97	12487.11	28639.31	-

a single application thread is executed in each node, although Intel can handle multiple application threads per-process. For this reason, we also present the case of 4 nodes with 2 application threads per-node in the case of the Intel compiler. The results show that OMPi is faster than Intel when the number of nodes is relatively small. However, Intel seems to scale better than OMPi. On 8 nodes, Intel and OMPi have similar overheads in all cases except of the `parallel reduction` overheads. Here, OMPi+Mome experiences very high overheads compering with OMPi+Mocha or ICC. A reason for this could be the strict memory protocol that Mome uses. All team members atomically write the shared reduction variable, while in every write operation, the new value of the reduction variable is immediately propagated to all other nodes. This causes the heavy-weight protocol of the sequential consistency to be invoked at every write operation. From the other hand, OMPi+Mocha or ICC, which targets a modified version of TreadMarks, experience lower overheads due to the fact that a light-weight relaxed memory consistency model is exploited.

Additionally, ICC experiences lower `parallel reduction` overheads in the 4×2 case. Although the number of execution entities remains the same (8), the `parallel reduction` overhead drops from 6362.96 μs to 5228.18 μs. This can be explained from the fact that intra-node threads share the modifications of the virtual memory. A page update performed by a thread is directly visible to all other intra-node threads through hardware shared memory. From the other hand, `parallel for` and `single` overheads increase in the 4×2 case. Considering the `single` overheads, this can be explained from the fact that multiple threads and processes are competing for the execution of the `single` region. In some implementations, the master thread is always responsible for executing the `single`, while other processes wait the master thread's completion. Other implementations use atomic regions to ensure that only a thread executes the `single` region. In both cases, an hierarchical barrier or lock is needed to be implemented. In the first level, intra-node threads are synchronized, while in the second level inter-node processes are synchronized. This clearly adds overheads to the all OpenMP directives that require synchronization

operations to be performed.

Table 6.4:Overheads for the OMPi compiler(μs)

	OMPi + POSIX (8 threads)	**OMPi + OPRC (8 nodes)**
`parallel for`	54.79	1437.44
`barrier`	32.97	229.39
`single`	55.71	773.82
`parallel reduction`	39.12	4729.03

In Table 6.4, we present results for OMPi when the benchmarks are executed on a single SMP machine or on the HP XC cluster system. The SMP machine is an Intel SR6850HW 4M model with 4 Intel Xeon dual-core 3.0 GHz processors running Linux 2.6 and 4GB main memory. We present results for the case of 8 threads on the SMP machine using OMPi+POSIX (PTHR) or 8 nodes of the cluster using OMPi+OPRC targeting the Mocha SVM system. Although we could run the benchmarks on a single node of the cluster and observe the performance in the case of a single SMP machine, this would limit us to a small number of threads (2) because each node of the HP XC cluster is a dual-core processor. For this reason, we chose to run the benchmarks on the Intel SR6850HW using 8 threads which is equal to the number of physical processors of the machine.

The results confirm our predictions. The OpenMP overheads are significantly increased in the case of OMPi+OPRC. In some cases, the overhead is more than two orders of magnitude bigger than in the SMP case. This is a presumable result considering the high network latencies involved in inter-process communication especially when compered with the latencies of threads communications in hardware shared memory systems. Moreover, whenever the SVM system is involved (e.g. `parallel reduction`), overheads increase even more. The authors of [33] performed a series of experiments regarding the Intel compiler for cluster OpenMP execution. A comparison of the OpenMP overheads using the EPCC microbenchmark suite is made when the target is an SMP machine or a cluster system. Their results show that in all cases, the overheads taken on the cluster are significantly bigger than the ones on the SMP machine independently of the underlying network fabric (Gigabit Ethernet or InfiniBand). However, a faster network fabric results in smaller overheads when the number of nodes increases.

6.5.2 Applications

In this section, we present experimental results for a class of known parallel applications: NAS EP, Matrix Multiplication (MM) and Molecular Dynamics (MD). The EP application is a part of the OpenMP implementation of the NAS Parallel Benchmarks [19]. MM is a simple parallel matrix multiplication application. MD is the C version of the sample application available at the official site of OpenMP (http://www.openmp.org).

The EP (embarrassingly parallel) benchmark generates pairs of Gaussian random deviates according to a specific scheme. This is the best case possible case for any kind of SVM system because there is no sharing of pages between different nodes of the cluster.

MD is a form of simulation in which atoms and molecules are allowed to interact for a period of time under known laws of physics, giving a view of the motion of atoms. MD exploits numerical methods to solve the problem. Given positions, masses and velocities of `np` particles, MD computes the energy of the system and the forces on each particle. A numerical iterative procedure is used to obtain an approximation whose precision depends on the number of simulation steps. The computation of forces and energies is fully parallel using a `for` directive by which particles are distributed among the execution entities. However, the initialization step is performed sequentially by the master thread (node).

In MM, which multiples 2 square matrices, the master thread performs the initialization step and then each OpenMP thread (node) computes its statically assigned chunk of iterations. After the parallel region, the master thread accesses the resulted matrix.

Figure 6.4 depicts our results. We executed the applications on 2, 4, and 8 nodes of the HP XC cluster system. We present results for OMPi targeting Mocha, TreadMarks, Mome and ParADE along with ICC results. In EP (class A), things go quite well. The speedups in all cases are close to the ideal. This is logical due to the fact that this benchmark does not modify shared data and consequently the underlying SVM system does not penalize the execution except for the first copy of the data. A perfect speedup its not achieved due to the reductions that need to be done at the end of the loop and because the static schedule is not perfectly balanced; some nodes have more work to do than others. In MM, two square matrices of size $N = 1024$ are multiplied. Although nodes modify shared data, the relaxed memory consistency models deployed by all SVM systems except Mome, limit false-sharing; nodes may concurrently write on the same shared page but page modifications are not immediately propagated to them.

Mome seems to suffer from its sequential consistency model. This becomes clear in the MD (4096 particles, dimension=3) case. Things seem to get out of control in the case of OMPi+Mome. The main reason for that is frequent false-sharing. In MD, shared data occupy only a small a set of pages. Moreover, pages need to be frequently accessed. Particles are distributed among the participating nodes, while the main computational step includes the calculation of the forces and potential energies of each particle with respect to all other particles. That means that the shared arrays keeping the forces and energies are frequently accessed from the applications threads and although threads are writing on different locations in the arrays, often the same page is involved. ICC and OMPI+ParADE achieve better speedups although they are not close to the ideal. A reason for that could be the reduction operations performed at the end of each computational step.

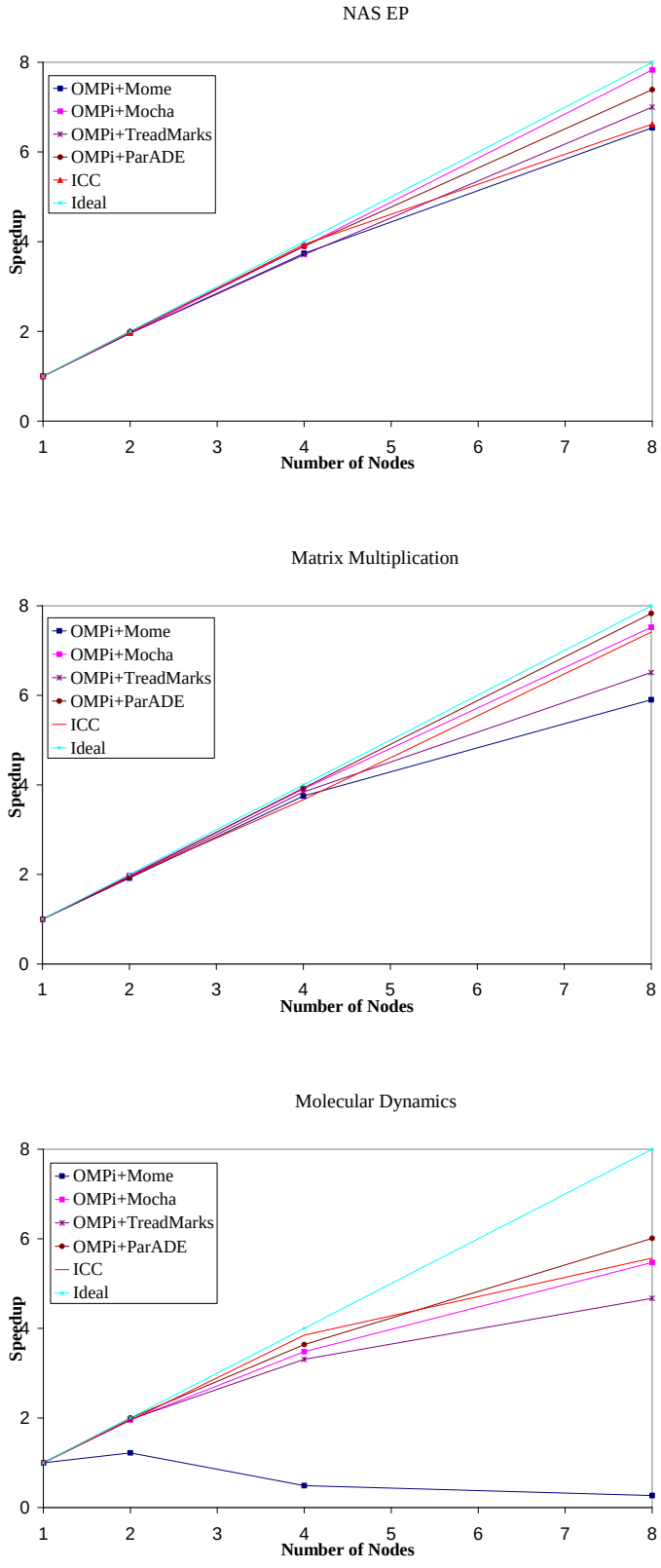

Figure 6.4: Speedups for NAS EP, MM and MD.

Chapter 7

Conclusions and Future Work

7.1 OpenMP and Nested Parallelism

In Chapter 4, we described our implementation of a threading library called `PTHR` for the support of nested parallelism. Also, we presented a novel methodology based on the EPCC microbenchmark suite which allows us to measure OpenMP overheads under nested parallelism. Using our methodology, we presented an extensive study of how commercial and research/expiremental compilers behave, in terms of overheads, when nested parallelism is in effect. To the best of our knowledge, this is the first study of its kind as all others have focused only on application speedups.

Our conclusion is that many implementations have scalability problems when nested parallelism is exploited and the number of threads increases well beyond the number of physical processors. This is most probably due to the kernel-level thread model the majority of the implementations use. When the number of threads that compete for hardware resources significantly exceeds the number of available processors, the system is overloaded and the parallelization overheads outweigh any performance benefits. Although our study was limited to two nesting levels, it became clear that studying deeper levels would only reveal worse behavior.

Possible future work on this subject includes the extension of our microbenchmarks to any arbitrary nesting level. Using the microbenchmarks as a tool, we can study ways of boosting performance. This is very important because nested parallelism is a very usable feature of OpenMP and is necessary on a wide range of parallel applications.

7.2 OpenMP and Clusters

In the second part, we presented the architecture of a new runtime library of OMPi, for the execution of OpenMP programs on top of clusters, called `OPRC`. It uses a hybrid approach where inter-process communication at the runtime library is achieved via MPI, while the shared memory abstraction at the application level is provided by an SVM system. As OpenMP becomes more and more popular, many studies have been proposed of combining an SVM system with an OpenMP compiler for the execution of OpenMP programs on top of clusters, matching the programmer-friendliness of OpenMP with the computational power of clusters. Recently, Intel presented the latest version of its OpenMP compiler which also includes support for cluster OpenMP. However, most implementations entirely use the SVM to offer shared memory semantics at both application and compiler level. We presented a more efficient solution by utilizing MPI for all the necessary communications in the runtime library of OMPi. Moreover, usually, most implementations target a specific SVM which is an inextricable part of the compiler. In our case, we managed to easily integrate a multitude of SVM systems due to the fact that the runtime library is actually independent of the target SVM system.

Regarding OMPi, many optimizations and extensions can be made as part of future work. Inter-process communication can be further optimized, while the translator can also take advantage of MPI whenever possible, limiting thus the utilization of the SVM system and subsequently boosting performance. The next step in the development of OMPi should be the support of multiple threads per-node, so as to exploit clusters of SMPs efficiently.

Although nested parallelism is a key feature of OpenMP, there has been no study of how nested parallel regions can be mapped on a cluster. All present OpenMP compilers for clusters do not support nested parallelism. Although, an obvious solution is to map the nested parallel regions locally on nodes using kernel-level or lightweight user-level threads, this would not exploit the computational resources of the cluster, in non-balanced situations. Consequently, more complex scheduling schemes must be considered. The development of efficient compilation systems for the execution of OpenMP on larger computational environments than a cluster, like grids, is in our opinion the next step in research. The compiler has to discover the multiple execution levels of the system in order to efficiently exploit the computational resources. For example, consider a system consisting of several clusters, while each node of the cluster is an SMP machine with each processor consisting of multiple hyper-threaded cores. The compiler's task is to discover the hierarchical execution levels and to map the execution vehicles into them in order to fully exploit the system.

Finally, our experience with OMPi shows that applications originally written taking into account the shared memory programming model may not perform well when executed on a cluster, especially when often communication is needed. In order to achieve better speedups, applications often need to be rewritten. However, optimizations like the ones

mentioned in Section 5.2.3, can significantly boost performance. Moreover, new special OpenMP directives for cluster application development could be introduced and exploited by advanced OpenMP programmers. For example, the programmer could use directives to explicitly distribute shared data in a way that every node of the cluster performs mainly computations with local data.

Bibliography

[1] OpenMP Architecture Review Board: OpenMP and C++ Application Pogram Interface, Version 2.5, May 2005.

[2] E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro and J. Oliver, NanosCompiler: A Research Platform for OpenMP Extensions, In *Proc. of the first Europian Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.

[3] C. Brunschen, OdinMP/CCp - A Portable Compiler for C with OpenMP to C with POSIX Threads, Master's thesis, Dept. of Information Technology, Lund University, Sweden, July 1999.

[4] J. M. Bull, Measuring Synchronization and Scheduling Overheads in OpenMP, In *Proc. of the 1st European Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, 1999.

[5] J. M. Bull and D. O'Neill, A Microbenchmark Suite for OpenMP 2.0, In *Proc. of the 3th European Workshop on OpenMP (EWOMP '01)*, Barcelona, Spain, 2001.

[6] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley, 1997.

[7] J.-H. Chow, L. E. Lyon and V. Sarkar, Automatic parallelization for symmetric shared-memory multiprocessors, In *Proc. of the 1996 conference of the Centre for Advanced Studies on Collaborative research (CASCONő96)*, Toronto, Canada, November 1996.

[8] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade and J. Labarta, Running OpenMP Applications Efficiently on an Everything-Shared SDSM, *Journal of Parallel and Distributed Computing* **66** (2006) 647–658.

[9] V. V. Dimakopoulos, P. E. Hadjidoukas and G. Ch. Philos, A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism, In *Proc. of the 4th International Workshop on OpenMP (IWOMP'08)*, West Lafayette, IN, USA, May 2008.

[10] V. V. Dimakopoulos, E. Leontiadis and G. Tzoumas, A Portable C Compiler for OpenMP V.2.0, In *Proc. of the 5th Europian Workshop on OpenMP (EWOMP '03)*, Aachen, Germany, October 2003.

[11] R. Eigenmann, J. Hoeflinger, R. H. Kuhn, D. Padua, A. Basumallik, S. Min and J. Zhu, Is OpenMP for Grids?, In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, 2002.

[12] P. E. Hadjidoukas and V. V. Dimakopoulos, Nested Parallelism in the OMPi OpenMP C Compiler, In *Proc. of the European Conference on Parallel Computing (EUROPAR '07)*, Rennes, France, August 2007.

[13] J. P. Hoeflinger, Extending OpenMP to Clusters, White Paper, Intel Coorporation, 2006.

[14] Y. C Hu, H. Lu, A. L. Cox and W. Zwaenepoel, OpenMP for Networks of SMPs, *Journal of Parallel and Distributed Computing* **60** (2000) 1512–1530.

[15] W. Hu, W. Shi and Z. Tang, JIAJIA: An SVM System Based on A New Cache Coherence Protocol, In *Proc. of the 7th International Conference on High Performance Computing and Networking (HPCN '99)*, Amsterdam, The Netherlands, April 1999.

[16] L. Iftode, J. P. Singh and K. Li, Scope Consistency: A bridge between release consistency and entry consistency, In *Proc. of the 8th ACM Annual Symposium on Parallel Algorithms and Architectures (SPAA '96)*, Padua, Italy, June 1996.

[17] Y. Jeegou, Implementation of Page Management in Mome, a User-Level DSM, In *Proc. of the 3th IEEE International Symposium on Cluster Computing and the Grid (CCGRID '03)*, Tokyo, Japan, May 2003.

[18] G. Zhang, R. Silvera and R. Archambault, Structure and algorithm for implementing OpenMP workshares, In *Proc. of the 5th Workshop on OpenMP Applications and Tools (WOMPAT '04)*, Houston, TX, USA, 2004.

[19] H. Jin, M. Frumkin, and J. Yan, The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance, Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[20] S. Karlsson, A Portable and Efficient Thread Library for OpenMP, In *Proc. of the 6th European Workshop on OpenMP (EWOMP '04)*, Stockholm, Sweden, October 2004.

[21] Y. S. Kee, J. S. Kim and S. Ha, ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Proc. of the 15th International Conference for High Performance Computing, Network, Storage, and Analysis (SC '03)*, Phoenix, AZ, USA, November 2003.

[22] Y. S. Kee, J. S. Kim and S. Ha, Memory management for multithreaded software DSM systems, *Parallel Computing* **30** (2004) 121–138.

[23] P. Keleher, A. L. Cox, S. Dwarkadas and W. Zwaenepoel, TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, In *Proc. of the Winter 94 USENIX Conference*, San Fransisco, CA, USA, January 1994.

[24] K. Kise, T. Katagiri, H. Honda and T. Yuba, Evaluation of the Acknowledgment Reduction in a Software-DSM System, In *Proc. of the 6th International Conference on Parallel Processing and Applied Mathematics (PPAM '05)*, Poznan, Poland, September 2005.

[25] Tyng-Yeu Liang, Shih-Hsien Wang, Jyh-Biau Chang and Ce-Kuen Shieh, Supporting the OpenMP Programming Interface on Teamster-G, *Advances in Grid and Pervasive Computing* **3947** (2006) 547–556.

[26] Tyng-Yeu Liang, Chun-Yi Wu, Jyh-Biau Chang, and Ce-Kuen Shieh, Teamster-G: A Grid-enabled Software DSM System, In *Proc. of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '05)*, Cardiff, UK, May 2005.

[27] C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, OpenUH: An Optimizing, Portable OpenMP Compiler, In *Proc. of the 12th Workshop on Compilers for Parallel Computers*, A Coruna, Spain, January 2006.

[28] S. J. Min, A. Basumallik, and R. Eigenmann, Supporting Realistic OpenMP Applications on a Commodity Cluster of Workstations, In *Proc. of the International Workshop on OpenMP Applications and Tools (WOMPAT '03)*, Toronto, Canada, June 2003.

[29] D. Novillo, OpenMP and automatic parallelization in GCC, In *Proc. of the 2006 GCC Summit*, Ottawa, Canada, June 2006.

[30] G. Ch. Philos, V. V. Dimakopoulos and P. E. Hadjidoukas, A runtime architecture for ubiquitous support of OpenMP, In *Proc. of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, Krakow, Poland, July 2008, to appear.

[31] M. Sato, S. Satoh, K. Kusano and Y. Tanaka, Design of OpenMP Compiler for an SMP Cluster, In *Proc. of the first Europian Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September 1999.

[32] M. Schulz, Overcoming the problems associated with the existence of too many DSM APIs, In *Proc. of the second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '02)*, 2002.

[33] C. Terboven, D. Mey, D. Schmidl, and M. Wagner, First Expiriences with Intel Cluster OpenMP, In *Proc. of the 4th International Workshop on OpenMP (IWOMP '08)*, West Lafayette, IN, USA, May 2008.

[34] X. Tian, J. P. Hoeflinger, G. Haab, Y-K Chen, M. Girkar and S. Shah, A compiler for exploiting nested parallelism in OpenMP programs, *Parallel Computing* **31** (2005) 960–983.

Author's Publications

- G. Ch. Philos, V. V. Dimakopoulos and P. E. Hadjidoukas, A runtime architecture for ubiquitous support of OpenMP, In *Proc. of the 7th International Symposium on Parallel and Distributed Computing (ISPDC'08)*, Krakow, Poland, July 2008, to appear.

- V. V. Dimakopoulos, P. E. Hadjidoukas and G. Ch. Philos, A Microbenchmark Study of OpenMP Overheads Under Nested Parallelism, In *Proc. of the 4th International Workshop on OpenMP (IWOMP'08)*, West Lafayette, IN, USA, May 2008, 1–12.

Short Vita

George Philos was born in Ioannina in 1982. He is a MSc student in the Department of Computer Science at the University of Ioannina (UoI) since September 2005. He received his BSc degree from the same department in 2005 and until now, he is a member of the Parallel Processing Group.

Contact e-mail: georgephilos@yahoo.gr