Program Development Tools for Clusters of Shared Memory Multiprocessors

B. Chapman, J. Merlin, D. Pritchard
Dept. of Electronics & Computer Science, University of Southampton, Southampton, UK

F. Bodin, Y. Mevel
IRISA, University of Rennes, Rennes, France

T. Sørevik
Institute for Informatics, University of Bergen, Bergen, Norway

L. Hill
Simulog SA, Sophia Antipolis, France

Abstract.
Applications are increasingly being executed on computational systems that have hierarchical parallelism. There are several programming paradigms which may be used to adapt a program for execution in such an environment. In this paper, we outline some of the challenges in porting codes to such systems, and describe a programming environment that we are creating to support the migration of sequential and MPI code to a cluster of shared memory parallel systems, where the target program may include MPI, OpenMP or both. As part of this effort, we are evaluating several experimental approaches to aiding in this complex application development task.

Keywords: parallel programming, distributed shared memory, parallelization, program transformations, program development environment

1. Introduction

Most major hardware vendors market cost-effective parallel systems in which a modest number of processors share memory. Such shared memory parallel workstations (SMPs) are being increasingly deployed, not only as stand-alone computers, but also in workstation clusters. They are also used to populate the nodes of distributed memory machines such as IBM's SP-2 and the Compaq/Quadrics QM-1. The SGI Origin 2000 and HP's V-class systems provide cache coherency across their SMP nodes and have very low latency. Users are encouraged to program the entire system as if it were a large SMP. Within the US ASCI project, several large-scale platforms are being constructed with a similar mix of shared and distributed parallelism at the hardware level. At the other end of the scale, we expect to see a proliferation of moderately sized multiprocessor workstations and PCs with shared memory, as well as clusters of such machines being constructed and used to provide high performance computing at a low cost.

There are vast differences in the computational power, interconnection technology and system software of the range of architectures mentioned above. Moreover, current programming models have not been developed with hierarchies of both shared and distributed memory parallelism in mind. It is thus not surprising that none of the available programming approaches are well matched to the full range of architectures we consider. The relative cost of remote data accesses, for example, may not only significantly affect the overall program development strategy but even the choice of programming paradigm. Some of the issues involved in application development at the highest end are discussed by Keyes et al. in (Keyes et al, 1997).

We are creating a programming environment to facilitate application development for the range of systems described above. As part of this effort, we are studying alternative programming models for SMP clusters in order to better understand the issues involved. When completed, the **POST environment** will support the creation of programs using MPI and OpenMP, either alone or in combination, to target such systems. The environment will be highly interactive and will cooperate with the user to develop code rather than provide a high degree of automation. POST will help the user analyze an existing program before adapting it; yet it will also apply novel techniques such as Case-Based Reasoning to derive a successful adaptation strategy from a database of known strategies.

This paper is organised as follows. In the next section we discuss potential programming models for clusters of SMPs. We then describe experiments and early work using such models, before introducing the features of our environment in Section 4. Related work is outlined and our efforts are then summarized.

2. Programming SMP Clusters

For the purposes of this paper we consider SMP workstation clusters, software DSM systems, and tightly-coupled distributed memory systems with SMP nodes, all to be clusters of SMPs, where each SMP is a *node* in the cluster. Program performance on such systems will be affected by all of the system features mentioned in the previous section; in addition, the organization of local memory within each node will have a significant impact.

Available parallel programming models include MPI, for explicit message-passing programming, HPF for higher-level distributed memory parallel programming, and OpenMP (OpenMP Consortium, 1997) for multi-threaded, shared memory programming. The first two are appropriate for clusters of single-processor workstations, whereas the last of these was primarily developed with an SMP node in mind.

MPI is the most flexible and general of these paradigms, and it *can* be used to program all of the systems we consider. However, it does not directly support the exploitation of shared memory, and processes executing within a common shared memory environment must communicate explicitly. Simple experiments on a Quadrics QM-1 with SMP nodes showed that current MPI library implementations have not yet been adapted to deal with SMP nodes, and that they therefore transmit all messages via the communication interface. In consequence, they actually exchange data faster between processors on different nodes than on the same node (Lancaster, 1998).

HPF can also be used to program the range of systems indicated. It has benefits in terms of software maintenance, but is less general than MPI, and also does not provide features for shared memory programming. Nevertheless, it has been demonstrated that an HPF compiler can be used to generate multi-threaded programs that exploit SMPs, and suitable extensions to the HPF language may permit a flexible tasking model within HPF (Brandes, 1999).

OpenMP has been designed explicitly for shared memory parallel programming and realises this at an appropriately high level. Two features make it particularly interesting for SMP clusters: firstly, it permits nested parallelism, and secondly, it facilitates the specification of coarse grain parallel regions. However, OpenMP can only be deployed on a cluster of SMPs if there is software support for cache coherency across the cluster. Although this is currently only the case for the CC-NUMA machines on the market, on-going research efforts aim to provide such support for a range of clustered systems (Lu et al, 1999; Sato et al, 1999).

Where a software layer providing cache coherency is not present, MPI may be used either alone or in conjunction with OpenMP. The latter option permits direct exploitation of the architectural features of the system, and provides an easy upgrade path for existing MPI code. It is also possible to use HPF alone, or to interface HPF with OpenMP, and we are investigating the viability of a tighter coupling of this high-level alternative (Chapman et al, 1998).

3. Some Experiments

To date, most of our own experiments have attempted to use OpenMP, and combinations of OpenMP and MPI, on tightly coupled distributed memory systems with SMP nodes, and on the Origin 2000. We began with a broad range of experiences in MPI application development, and tool support for this paradigm.

The latter architecture poses a number of problems that differ substantially from other SMP clusters, since the operating system does not provide the same transparency of runtime behaviour. Working with it has permitted us to gain initial insight into some of the potential problems that this entails. This system has also enabled us to experiment with MPI, OpenMP, and their combination, as well as to compare this with the use of SGI native constructs for scheduling parallel loops, and allocating data and threads.

In addition to gaining material for a feature of our tool (cf. Section 4.2.3), this work was an essential part of our effort to better understand how the OpenMP and MPI paradigms were likely to be used in practice on clusters of SMPs, especially in combination with one another. It also helped us evaluate which of the many analysis and adaptation tasks that application migration requires would be the most time-consuming and difficult to realise in our context. Despite their obvious simplicity, these experiments have helped us design parts of the programming environment.

In one such experiment, the 3400-line NAS-BT benchmark was ported to OpenMP and evaluated on a 128-node Origin 2000 (Blikberg et al, 1998). At each time step it solves a large system of algebraic equations iteratively using the SSOR method[1]. The righthand side is updated in six separate, independent routines: it is easy to parallelize with a `PARALLEL SECTIONS` directive, needing a minor code modification to enable the use of an OpenMP reduction operation. But this strategy does not provide work for more than six processors. Since each section contains a large amount of fine grained parallelism, OpenMP's nested parallelism could remedy this: unfortunately, it was not implemented in our compiler. Worse still, data accesses differed in the individual sections, and the load balance was thus not good, even on exactly six threads. This approach was abandoned in favor of an alternative based upon the specification of parallel loops. Timings are in (Blikberg et al, 1998).

The loops performing grid point updates could be fully parallelized. Since iterations performed identical work, it was expected that an

[1] A new version of this benchmark suite which is better suited to OpenMP parallelization is under development at NASA Ames.

even distribution of iterations to threads would lead to a well-balanced computation. This was not the case: two of the threads consistently outperformed the others. The speedier threads were located on the same node as the data! Several experiments, by ourselves and others, confirm the importance of coordinating the placement of work and data on the Origin 2000, in order to ensure that data is available where needed, and that the workload remains balanced across the machine. SGI provides directives that enable to user to assume control of such mappings, in particular permitting a binding of the execution of code to the node storing a specified data object. However, these remedies are available on this system only; they are *not* features of OpenMP. On SMP clusters with longer latencies, it will be even more important to give the user some direct control over work and data placement.

MPI and OpenMP have been successfully used together in several notable experiments, including one award-winning computation (Bova et al, 1998), and are becoming an accepted combination, especially for running large programs on SMP clusters, e.g. (Faulkner, 1999). We discuss two examples.

Colleagues at Bergen have parallelized a 2D seismic model for the Origin 2000 using both MPI and OpenMP (Frøyland et al, 1998). In this case, the problem required parallelization of a number of shots, which were independent except for a global reduction terminating the computation. It was thus particularly easy to port this code under MPI; however, the maximum number of shots was 30. Since each shot contained much internal parallelism, OpenMP was used to parallelize the process code. As a result, the work for each shot scaled up to about 20 processors. With applications of this nature, it is relatively easy to create an outer layer of MPI code and then use OpenMP to exploit fine grain parallelism.

MICOM is an Atlantic ocean model code with roughly 20000 lines of code and about 600 loop nests. A message passing version of this program has been further adapted to run on clusters of SMPs (Sawdey, 1998). SC-MICOM is an MPI program which starts up a number of OpenMP threads on each executing node. It realizes a hierarchical data distribution strategy matching the cluster architecture, by first partitioning the grid into regions that are assigned to SMP nodes. On each node, the local region is further subdivided. Within each node, computations are distributed dynamically to the processors, thus achieving a good overall load balance. The entire application is written in an SPMD style: the MPI node programs start up a parallel region to execute the local program at the beginning of the computation. The region is terminated at the end of the run, and thus the threads remain in existence throughout. MPI communication is required to exchange boundary values at the global level. These, and all other MPI calls required within the program, must be executed by a single thread only; OpenMP provides several constructs to ensure that this is correctly handled, and it is therefore fairly straightforward to modify the program accordingly.

3.1. The Adaptation Process

If MPI is employed to coordinate the node processes of an application executing on a cluster of SMPs, then it will explicitly map data and work across the nodes of the target machine. The adaptation of a program to perform well under this paradigm, and the insertion of MPI constructs, requires that the user perform a global program analysis, in order to select the parallelisation strategy and understand where the approach may require additional code modification. The application developer will need to consider data access patterns, the call graph, local data flow, data dependences, and more, in order to correctly parallelise the code.

In comparison with the effort required to restructure a program for parallelisation under MPI or HPF, the insertion of OpenMP constructs is relatively straightforward. If OpenMP is used in SPMD style in conjunction with pre-existing MPI constructs, these must be identified and a single-threaded execution enforced, as indicated above. In addition, operations such as reductions may have to be decomposed into local (within-SMP-node) and global (cluster-wide) components.

Major tasks for the user during the OpenMP adaptation process will include selecting appropriate loops for parallelisation, together with the corresponding static or dynamic loop execution strategies. Obtaining good cache locality, and eliminating sources of false sharing, may require considerable restructuring of individual loop nests and, possibly, redefinition of data structures.

If OpenMP is used in conjunction with software support for cache coherency, it does not need to be combined with MPI. On large systems of this kind, however, we believe that the current OpenMP language may make it difficult to achieve consistent good performance without recourse to non-standard constructs. Nevertheless, for such systems the user must also pay careful attention to loop restructuring, with the goal of optimizing cache usage on each SMP node, and take good care to distribute work appropriately.

4. The POST Development Environment

The POST program development environment is an interactive system which provides Fortran program analysis and transformations to support application parallelisation with MPI library calls and/or OpenMP directives. It offers analysis required to help the user restructure a program for MPI parallelism; and it also provides functionality to generate and check OpenMP program constructs. It therefore supports porting efforts which adopt either one of these programming models, or which combine them in order to fully exploit the capabilities of SMP clusters. It is particularly suitable for any effort that extends an existing MPI program by OpenMP constructs.

At the heart of the environment is FORESYS (Simulog, 1996), a commercial Fortran source code analysis and restructuring system. It is an interactive system which is used for reverse engineering and upgrading of existing software, and for validation and quality assurance of new code. It performs an in-depth interprocedural program analysis and can display a variety of information, such as the call graph, data flow and data dependence information. It links graphical representations with source text displays wherever appropriate – for example, enabling the user to view source code of individual program regions via a mouse click on an interactive display of the call graph – and enables rapid navigation and inspection of the source code. Some of its advanced display functionality was originally realised in the ANALYST prototype (Chapman et al, 1997) and tightly coupled with FORESYS in the ESPRIT project FITS (Chapman et al, 1999).

Our implementation work is further based upon TSF (Bodin et al, 1998), a FORESYS extension that provides access to its abstract syntax tree, including the results of restructuring analysis, via a custom scripting language. It can extract and insert information into the FORESYS program database, and cause it to derive additional data.

The POST environment provides several levels of support for code adaptation, which we now describe. All features described here are realized via FORESYS and TSF.

4.1. Basic Level of Support

At the lower level, the environment helps a user gain an overview of the source program, which may already have parallel constructs. Support is provided for inserting MPI or OpenMP parallel constructs, in the form of templates. Syntactic checking ensures that environment variables have been inserted (cf. the TSF script in Figure 1), and that matching

```
SCRIPT insert_CALL()
 . . . . . . .
IF (NOTUSED("mpi_ierr")) THEN // exists?
 DECLARE ("include 'mpif.h'") // no: add
 DECLARE ("integer mpi_ierr") // declare
ENDIF
IF (NOTUSED("mpi_rank")) THEN
 DECLARE ("integer mpi_rank")
ENDIF
 . . . . . . .
ENDSCRIPT
```

Figure 1. TSF script example.

constructs, such as the `END` of a parallel region, have been properly defined. Other functionality includes the following:

Analysis: One can **browse** and **edit** the source code using graphical and text displays. Various kinds of global and local information on data structures, data flow and aliasing are available, and may be presented within a graph or by highlighting and colour-coding in the source text display. Data dependence graphs may be used to select loops for OpenMP parallelization.

Data **locality analysis** will evaluate the usage of array elements, indicating the cache re-use potential of arrays referenced, and including indirect accesses in the evaluation wherever possible. This analysis will also be used to help select a loop for parallelization from one of the candidate loop levels in a loop nest, as well as provide information on suitable strategies for assigning loop iterations to processors, in accordance with the loop parallelization.

Constructs that require special handling, such as a program's I/O, will be quickly identified.

Program structure display: Displays are being developed to show MPI processes and their interactions, as well as to indicate the scope of OpenMP parallel regions. The latter is needed because work sharing constructs may be placed inside procedures called from within parallel regions.

Transformations: A set of standard program transformations is provided, including loop distribution, loop fusion, loop interchange and array expansion, each of which may be applied interactively by the user.

An interprocedural algorithm is under construction to automatically parallelize selected loops under OpenMP. Since, with the exception of *orphan* directives, OpenMP has been based firmly upon previous sets of directives for shared memory parallel programming, there is already a body of experience in the generation of OpenMP loops.

4.2. Higher Levels of Support

Within the POST environment, we are also working on a range of features that go beyond the largely traditional approach outlined above. These include additional support for the task of identifying appropriate loop nests for parallelisation, and improving the performance of OpenMP parallel loop nests, permitting the user to create custom program transformations, so that specific tasks may be repeated, as well as experimenting with case-based reasoning, which we are evaluating as an approach to selecting and realising a parallelisation strategy for the code under consideration.

4.2.1. *Guidance in Applying Transformations*

We provide improved support for the task of creating OpenMP code by developing a feature to guide the user in this task. This includes a profile option to help detect the most time-consuming regions of a program, and also to determine whether there are performance problems associated with memory usage. Profiling results are able to give coarse information that indicates what fraction of the data accessed by a loop was in cache.

It also interacts with the user in the selection of a loop level for parallelization. Here, it is possible to highlight the loops that may be immediately parallelized, as well as display the dependences that prevent parallelization of other levels. Features from the basic support outlined above may then help the application developer check the references to the variables involved in dependences, in an effort to understand whether or not the corresponding dependences can be discounted.

The guidance system may also propose a loop optimization strategy. It suggests code improvements to the user in order to enable better automatic parallelization, or will propose pre-defined sequences of transformations to improve upon initial parallelization. In order to do this, we are studying the effect of a variety of transformations, and their combinations on selected target architectures. There is already a body of knowledge on this topic (cf. (Singhai et al, 1997)).

```
   DO 50 K = 1, KK
    KNM = K + KK*(NM-1)
    DO 50 J = 1, JJ
     DO 60 L = 1, ISU(J) !iterate over sea
      DO 60 I = IFU(J,L),ILU(J,L) !sea coords
       PU(I,J,K+1) = PU(I,J,K)+DPU(I,J,KNM)
60   CONTINUE
     DO 70 L = 1, ISV(J)
      DO 70 I = IFV(J,L), ILV(J,L)
       PV(I,J,K+1) = PV(I,J,K)+DPV(I,J,KNM)
70   CONTINUE
50 CONTINUE
```

Figure 2. MICOM: example loop

4.2.2. *User-Defined Transformations*

The TSF scripting language is provided as part of the POST environment. Since it enables users to create new transformations, the system is extensible. An application developer may perform a variety of program modifications, including highly repetitive tasks, via TSF scripts.

We briefly illustrate their use by showing how they enabled a fast MPI parallelization of the sequential MICOM code (see Section 3). Figure 2 shows a typical loop structure. Loops `J`, `L`, and `I` scan only those coordinates that correspond to the Atlantic ocean, and skip land areas. Loop bounds are read from pre-defined data structures. We distribute arrays by block in the first two dimensions, and may parallelize the loop nest in both `J` and `I` dimensions. Transforming the code requires adapting array declarations, ensuring that the local size includes overlap regions, modifying assignments and modifying loop bounds, where some redundant computations are performed to reduce communication. MPI communication calls are inserted.

A small set of TSF scripts was written to perform these changes. This effort took just a few days and saved weeks of manual adaptation. Results for our example loop are shown in Figure 3.

Among the additional benefits of using scripts are the reduction in the likelihood of programmer error, and the fact that they enable the same strategy to be applied to a revised version of the original (sequential) program with minimal effort.

4.2.3. *Case-Based Reasoning*

The final component of the environment may be seen as an alternative approach to program restructuring, which we apply to our set of

```
   do 50 K = 1,KK
    KNM = K + KK*(NM-1)
    do 50 J = 1-margin,jj+margin
     do 60 L = 1,ISU(J)
      do 60 I = max(1-margin,IFU(J,L)),
  &                min(ii+margin,ILU(J,L))
       PU(I,J,K+1) = PU(I,J,K)+DPU(I,J,KNM)
60   continue
     do 70 L = 1,ISV(J)
      do 70 I = max(1-margin,IFV(J,L)),
  &                min(ii+margin,ILV(J,L))
       PV(I,J,K+1) = PV(I,J,K)+DPV(I,J,KNM)
70   continue
50 continue
```

Figure 3. MICOM loop; result of TSF scripts

target systems. It aims to capitalize on, and facilitate the reuse of, the experience of program developers.

The expert knowledge associated with parallelization is not well structured and does not lend itself to conventional rule-based systems. In contrast, the Case-Based Reasoning approach (Kolodner, 1993) aims to solve a given problem by adapting the solution of a similar one already encountered. Its application in our context requires us to *identify* code fragments for investigation: these can be obtained by profiling. In order to *retrieve* similar stored cases, a notion of similarity must be defined and used to compare a code fragment with stored examples. Its definition may be context-dependent. Finding a suitable definition is hard: in a pioneering study, Mevel proposes a set of indices on which this might be founded (Mevel, 1999) and reports good results with abstractions based upon syntax, data dependence, control flow, etc. The solution adopted in the retrieved code may be automatically *reused*, if it consists of predefined automatic transformations, or it may need to be applied manually to the new case. Finally, it should be possible to *retain* the new solution.

In our environment, a case is a pair $C = (Problem,\ Solution)$ where *Problem* is a transformation problem and *Solution* is a sequence of transformations which represent its solution. In practice, C is defined as a set of fields. The *initial properties* define the context of a case, including the target architecture, the name of the transformation, and the application domain. The *final properties* specify properties of the code after applying the solution. The *original code fragment* is the state before the proposed solution is implemented. The *code pattern* describes code fragments on which the solution can be applied. It is used to identify cases that are similar to the original code fragment. A *list of program transformations* is given, which is to be applied to the original code fragment. Finally, we require a *documentation* of the solution, which describes the usage of, and any restrictions to, the proposed solution. This is especially important in our environment, which largely gives control to the user.

Many applications of this approach are possible. It may be suitable, for example, if a given loop performs a parallel array transpose inefficiently. The facility might be able to adapt a good transpose algorithm for immediate application in the given case. It may also enable the parallelization of loops that are too hard for the automatic system, e.g. if they require handling arithmetic series.

Within the POST programming environment, we are using this to construct a tutorial mode of operation, in which examples of parallelization principles and practice are illustrated by code fragments, with the *Problem* representing the code prior to parallelization, and the *Solution* representing a suitable adaptation. In addition, we are creating a demonstrator to show how the approach might be used to suggest transformation strategies in codes that are hard to handle automatically.

5. Related Work

Several research efforts aim to extend the range of applicability of the OpenMP programming paradigm. These include work to exploit multiple levels of parallelism within the compiler (Ayguade et al, 1999), as well as portable research implementations (Brunschen et al, 1999) and software systems providing cache coherency for a variety of (multiprocessor) workstations and PCs in a network (Lu et al, 1999; Sato et al, 1999). A range of proposals for modest extensions to the language are also expected to appear.

In addition to vendor-supplied parallel program development toolsets, there are a range of products from individual vendors to support the creation of MPI or OpenMP code.

These include the Kuck and Associates product line (Kuck and Associates, 1998) for shared memory parallelisation. Their *Visual KAP for OpenMP* operates as a companion preprocessor for the Digital Visual Fortran compiler. Although it has many user-controlled options, it essentially functions as a batch tool on the source files. The *FORGE* parallelization tools (APR, 1993) from Applied Parallel Research enable source code editing and offer control and data flow information, in combination with an HPF compiler. *VAST* (PSR, 1998) from Pacific Sierra Research helps a user migrate from Fortran 77 to HPF code by automatic code cleanup and translation. We are not aware of tools which explicitly support the development of programs that combine the OpenMP and MPI paradigms.

Interactive display of analysis results and the provision of user transformations was pioneered in the *ParaScope Editor*, for improving shared memory programs. *CAPTools* (Ierotheou et al, 1996) interacts with the user on the basis of a program data base, which the user may query and modify, in order to improve the automatic transformation of a sequential program to an MPI one. Finally, *SUIF Explorer* (Liao et al, 1998) supports parallelization by identifying the most important loops via profiling. It takes a different approach from POST to help the user perform this task, for example, applying program slicing to present information on dependence-causing variables.

6. Conclusions and Future Work

In this paper we have described a portable and extensible programming environment for the development of parallel Fortran applications under MPI and OpenMP. One of the challenges of this work is to provide support for both paradigms, including their combined deployment, within a single coherent environment. In addition, we are exploring the usefulness of several novel techniques for application parallelization. We have chosen not to attempt to create fully automatic solutions, but to implement and evaluate approaches that provide varying levels of automation under user control. The availability of a higher-level interface to the compiler's information, and the capabilities of the underlying system, enable us to create prototype systems of the kind described, within a reasonable time frame.

Within the ESPRIT Project POST, we have begun to work with application developers at several industrial sites in Europe, in order to obtain feedback on the functionality and features of our environment, and in order to acquire material for the case-based reasoning system.

Finally, the programming approaches we target are far from ideal. The use of a combined MPI-OpenMP approach requires the application developer to deal with two distinct and demanding programming models. Moreover, as already noted, it seems necessary to provide the user of a CC-NUMA machine with some additional control over the assignation of data and work to the system's nodes. We hope to contribute to the search for an improved unified model.

Acknowledgments

This work is part-funded by the ESPRIT project 29920, POST (Programming with the OpenMP STandard), whose name we have borrowed.

We thank Jerry Yan at NASA Ames for discussions on a number of related topics, including the NAS benchmarks, and David Lancaster at Southampton University for discussions and for providing benchmark results. We also thank Wolfgang Nagel and colleagues at the TU Dresden, who extended their VAMPIR performance analysis tool to enable closer investigation of the behaviour of some of the OpenMP codes.

References

Applied Parallel Research. *APR's FORGE 90 parallelization tools for High Performance Fortran.* APR, June 1993

E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez and J.I. Navarro. *Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study.* Proc. 28th Int. Conf. on Parallel Processing '99, Aizu, Japan, 1999

R. Blikberg and T. Sørevik. *Early experiences with OpenMP on the Origin 2000.* Proc. European Cray MPP meeting, Munich, Sept 1998

F. Bodin, Y. Mevel and R. Quiniou. *A user level program transformation tool*, Proc. Int. Conference on Supercomputing, 1998

S.W. Bova, C.P. Breshears, C. Cuicchi, Z. Demirilek and H.A. Gabb. *Dual Level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP.* Proc. Supercomputing '98, Orlando, 1998

T. Brandes. *Exploiting Advanced Task Parallelism in High Performance Fortran via a Task Library.* Proc. 5th Int. Euro-Par Conference (Euro-Par '99), LNCS 1685, Springer Verlag, 1999

C. Brunschen and M. Brorsson. *OdinMP/CCp – A Portable Implementation of OpenMP for C.* Proc. 1st European Workshop on OpenMP, Lund, Sweden, 1999

B. Chapman and M. Egg. *ANALYST: Tool Support for the Migration of Fortran Applications to Parallel Systems*, Proc. PDPTA' 97, Las Vegas, 1997

B. Chapman and P. Mehrotra. *OpenMP and HPF: Integrating Two Paradigms.* Proc. 4th Int. Euro-Par Conference (Euro-Par '98), LNCS 1470, Springer Verlag, 1998

B. Chapman, F. Bodin, L. Hill, J. Merlin, G. Viland and F. Wollenweber. *FITS—A Light-Weight Integrated Programming Environment.* To appear in Proc. Europar '99, Toulouse, 1999

T. Faulkner. *Performance Implications of Process and Memory Placement using a Multi-Level Parallel Programming Model on the Cray Origin 2000.* Available at URL `www.nas.nasa.gov/~faulkner`

L.A. Frøyland, F. Manne and N. Skjei. *2D seismic modelling on the Cray Origin 2000.* Internal report 1998-02-13 for Norsk Hydro. In Norwegian

C.S. Ierotheou, S.P. Johnson, M. Cross and P.F. Leggett. *Computer aided parallelisation tools (CAPTools)—Conceptual overview and performance on the parallelisation of structured mesh codes*, Parallel Computing, 22(2), 163–195, 1996

D. Keyes, D. Kaushik and B. Smith. *Prospects for CFD on Petafloops Systems.* ICASE Report No. 97-37, NASA Langley Research Center, 1997

J. Kolodner. *Case-Based Reasoning.* Morgan Kaufmann, 1993

Kuck and Associates. *KAP/Pro Toolset for OpenMP.* See `www.kai.com/kpts/`

D. Lancaster. *Results of GENESIS benchmark experiments.* Available at URL `http://gather.ecs.soton.ac.uk/`

S.-W. Liao, A. Diwan, R.B. Bosh Jr., A. Ghuloum and M.S. Lam. *SUIF Explorer: An interactive and interprocedural parallelizer.* Technical Report, Computer Systems Lab, Stanford University, 1998

H. Lu, C. Hu and W. Zwaenepol. *OpenMP on Networks of Workstations.* Proc. Supercomputing '99, Orlando FL, 1998

Y. Mevel *Environnement pour le portage de code oriente performance sur machines paralleles et monoprocesseurs.* Ph.D. Thesis, University of Rennes, France, 1999

OpenMP Consortium: OpenMP Fortran Application Program Interface, Version 1.0, October, 1997

C. Rodden and B. Brode. *VAST/Parallel: automatic parallelisation for SMP systems.* Pacific Sierra Research Corp., 1998

M. Sato, S, Satoh, K. Kusane and Y. Tanaka. *Design of OpenMP Compiler for an SMP Cluster.* Proc. First European Workshop on OpenMP, Lund, Sweden, 1999

A. Sawdey. SC-MICOM. Software and documentation available from `ftp://ftp-mount.ee.umn.edu/pub/ocean/`

Simulog SA, *FORESYS, FORtran Engineering SYStem, Reference Manual V1.5*, Simulog, Guyancourt, France, 1996

S.K. Singhai and K.S. McKinley. *A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality.* Computer Journal, 40(6), 340–355, 1997