The Shared Memory Parallelisation of an Ocean Modelling Code Using an Interactive Parallelisation Toolkit

C.S. Ierotheou, S. Johnson, P. Leggett, and M. Cross

Parallel Processing Research Group, University of Greenwich, London SE10 9LS, UK
Email: {c.ierotheou,s.johnson,p.leggett,m.cross}@gre.ac.uk

Abstract. This paper briefly describes an interactive parallelisation toolkit that can be used to generate parallel code suitable for either a distributed memory system (using message passing) or a shared memory system (using OpenMP). This study focuses on how the toolkit is used to parallelise a complex heterogeneous ocean modelling code within a few hours for use on a shared memory parallel system. The generated parallel code is essentially the serial code with OpenMP directives added to express the parallelism. The results show that substantial gains in performance can be achieved over the single thread version with very little effort.

1. Introduction

If oceanographers are to be allowed to continue to do relevant research then they will need to utilise powerful parallel computers to assist them in their efforts. Ideally, they should no be burdened with the task of porting their applications onto these new architectures but instead be allowed to focus their efforts on the quality of ocean models that are required. For example, the development of models that include features with both small spatial scales and large time scales and cover as large a geographic region as possible. The unfortunate reality is that a significant effort is often required to manually parallelise their model codes and this often requires a great deal of expertise. One suggestion is to substantially reduce the effort required for the parallelisation with the introduction of an effective parallelisation toolkit.

Today the shared memory and distributed memory programming paradigms are two of the most popular models used to transform existing serial application codes to a parallel form. For a distributed memory parallelisation of these types of codes it is necessary to consider the whole program when using a Single Program Multiple Data (SPMD) paradigm. The whole parallelisation process can be very time consuming and error-prone. For example, data placement is an essential consideration to efficiently use the available distributed memory, while the placement of explicit communication calls requires a great deal of expertise. The parallelisation on a shared memory system is only relatively easier. The data placement may appear to be less crucial than for a distributed memory parallelisation, but the parallelisation process is still error-prone, time-consuming and still requires a detailed level of expertise. The main goal for developing tools that can assist in the parallelisation of serial application codes is to embed the expertise and the automated algorithms to perform much of the tedious, manual and sometimes error-prone work, and in a small fraction of the time that would otherwise be taken by a parallelisation expert doing the same task manually. In addition to this, the toolkit should be capable of generating generic, portable, parallel source code from the original serial code [1, 2]. The toolkit discussed here was developed at the University of Greenwich and has been supplemented by a directive generation module [3] from Nasa Ames Research Center. The toolkit can generate either SPMD based parallel code for distributed memory systems or loop distributed directive-based parallel code for shared memory systems.

The aim of this paper is to report on how the toolkit was used and what level of effort was required to parallelise an ocean model code (typified by the Southampton and East Anglia model [4, 5]) for a shared memory based parallel system. A similar manual effort has not been undertaken due to the high cost associated with such a task. Finally, some remarks are made about the quality of the generated code and the parallel performance achieved on a test case.

2. The interactive parallelisation toolkit

The toolkit used in this study has been used to successfully parallelise a number of application codes for distributed memory systems [6, 7] based on distributing arrays across a processor topology. For an SPMD, distributed memory based parallelisation, the mesh over which these equations are solved is used as the basis for the partitioning of the data. The quality of the parallel source code generated benefits from many of the features provided by the toolkit. For example, the dependence analysis is fully interprocedural and value-based (i.e. the analysis detects the flow of data rather than just the memory location accesses) [8] and allows the user to assist with essential knowledge about program variables [9]. There are many reasons why an analysis may fail to accurately determine a dependence graph, this could be due to incorrect serial code, a lack of information on the program input variables, limited time to perform the analysis and limitations in the current state-of-the-art dependence algorithms. For these reasons it is essential to allow user interaction as part of the process, particularly if scalability is required on a large number of processors. For instance, a lack of knowledge about a single variable that is read into an application can lead to a single unresolved equation in the dependence analysis. This can lead to a single assumed data dependence that serialises a single loop, which in turn greatly affects the scalability of the application code. The placement and generation of communication calls also makes extensive use of the interprocedural capability of the toolkit as well as the merging of similar communications [10]. Finally, the generation of readable parallel source code that can be maintained is seen as a major benefit. The toolkit can also be used to generate parallel code for shared memory based systems, by inserting OpenMP [11] directives into the original serial code. This approach also makes use of the very accurate interprocedural analysis and also benefits from a directive browser (section 5) to allow the user to interrogate and refine the directives automatically placed within the code.

3. The Southampton-East Anglia (SEA) model

The SEA model [4, 5] was developed as part of a collaborative project between the Southampton Oceanography Centre and the University of East Anglia. The model is based on an array processor version of the Modular Ocean Model with a reduced set of options (MOMA) [12]. Although MOMA was not itself a parallel code, it could be arranged such that parallelism was exploited and this is what lead to the subsequent development of the SEA model code. The SEA model code can be configured with the aid of a C-preprocessor to execute in parallel on a distributed memory system

[13]. Work has been done relating to the parallelisation of the SEA code onto distributed memory systems [4, 14] but this will not be discussed here. Instead, a serial version of the SEA model code was used as the starting point for this study.

The surface of the model ocean is assumed to be split into a 2-D horizontal grid (i,j). Each (i,j) is used to define a volume of water that extends from the surface of the ocean to the ocean floor (k-dimension). The k-dimension is represented by a series of depths. The main variables solved by the model include the barotropic velocities and associated free-surface height fields as well as the baroclinic velocities and tracers. All other variables can be derived from these. The governing equations are discretised using a finite difference formulation in which the velocities are offset from the tracers and free-surface heights [15].

4. Parallelising a code using OpenMP

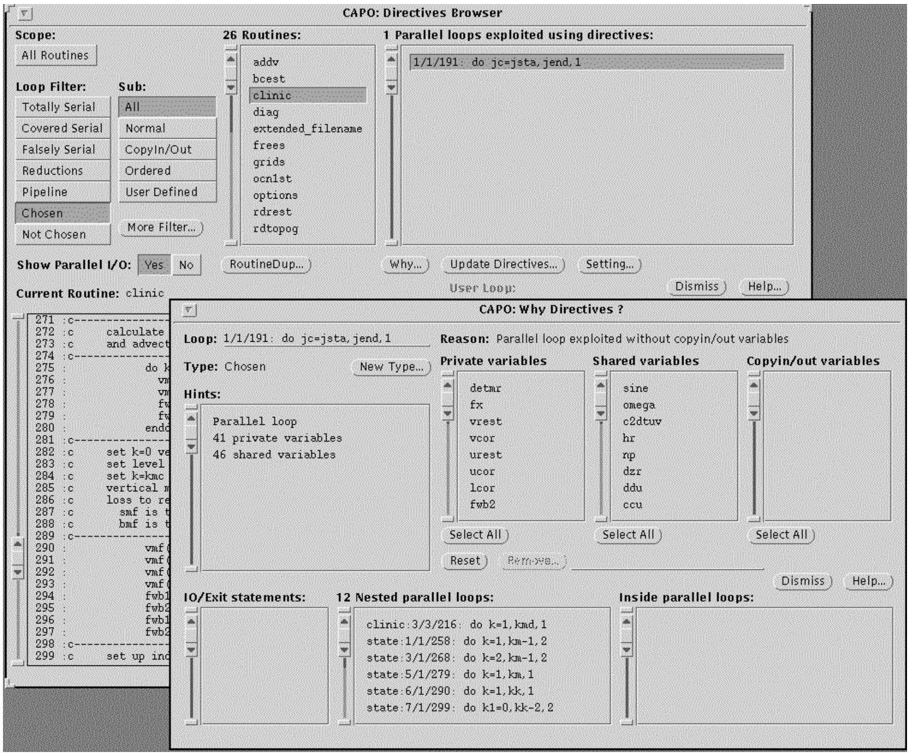
There are a number of different types of parallelism that can be defined using OpenMP, such as task or loop based parallelism. Here we focus on loop based parallelism and the issues in identifying privatisable variables. However, even in utilising these techniques there are essentially two practitioners – those that are CFD specialists (these include authors of the software) and those that are parallelisation experts. The CFD specialists have a deep level understanding of the code, the algorithms and their intended applicability, as well as the data structures used. This aids in identifying parallel loops due to independent calculation for cells in the mesh and also for identifying privatisable variables as they are often used as workspace or temporary arrays in CFD codes. These users tend to make use of implicit assumptions or knowledge about the code during the parallelisation. The parallelisation experts however, have the know how to carry out what is effectively an implicit dependence analysis of the code to examine the data accesses and therefore provide a more rigorous approach to the whole parallelisation process. In doing so, they still typically make assumptions due to the difficulty in manually performing a thorough investigation. Ideally, a user is required that has skills drawn from both disciplines to perform the parallelisation.

Adopting a formal approach to parallelising a code with loop distribution using OpenMP requires a number of considerations. For instance, it is necessary to carry out a data dependence analysis of the code as accurately as it is practical. As well as detecting parallel loops, the analysis should include the ability to identify the scope of variable accesses for privatisation. This means that the analysis must extend beyond the boundaries of a subroutine and therefore be interprocedural. This in itself can be a daunting task when tackled manually by a parallelisation expert, but can be implemented automatically as part of a parallelisation tool. The identification of PARALLEL regions using interprocedural information and the subsequent legal fusion of a series of these regions can be carried out automatically by the toolkit in order to reduce as much as possible the overheads associated with starting up and synchronising PARALLEL regions. In addition, the automatic identification, placement and legal use of NOWAIT clauses for DO loops can help to greatly reduce the overheads for synchronising amongst threads. These tasks require a very detailed analysis of the code and are sometimes viewed as too complex to implement manually, particularly when attempting to consider interprocedural effects.

5. The parallelisation of the SEA model code

The parallelisation of the SEA model code was achieved by executing the toolkit on a supported platform and performing a few simple processes within the toolkit. The first step is to read in the serial source code. The SEA model code is written in FORTRAN and contains nearly 8200 lines of source code. The next step was to perform a dependence analysis of the code using the analyser and this took a few minutes. Inspection of the unresolved questions identified by the analyser was followed by the user addition of some simple knowledge relating to the arrays storing the vertical depths ($0 \leq KMU \leq 32$ and $0 \leq KMT \leq 32$). A subsequent incremental analysis was performed in order to re-evaluate any uncertainties that remained using the additional user knowledge. A database is saved at this stage to allow the user to return at a later date to this point in the parallelisation process. The user can then use the toolkit to automatically generate OpenMP code without examining the quality of the code. Not surprisingly, the performance obtained for this version of the code was comparable to using the SGI auto parallelisation flag `-apo`. The parallel performance of this code was very poor with a speed up of 1.5 for 8 threads.

After reloading the saved database, the user is allowed to investigate in detail any serial loops in the code by using the interactive directives browser (Figure 1). This investigation is essential as it could lead to further parallelism being identified by the toolkit with the additional knowledge provided by the user. Equally important is the failure to look at such loops and allowing them to be executed in serial, as this can significantly affect the scalability of the code (Amdahl's law [16]). The directives browser presents information to allow the user to focus on the precise reasons why, for example, a loop cannot execute in parallel. The browser also provides information about which variables cause recurrences and hence prevent parallelism and which variables could not be legally privatised and therefore inhibit parallelism. This amount of detail in the browser enables both the parallelisation expert and CFD specialist (through their knowledge of independent updates in the algorithms and knowledge of workspace arrays) to address these issues. One possible course of action is to refine the dependence graph. For example, in the SEA code some array accesses define a multi-step update of the solution variables that can be executed in parallel. This may have been too complex to be detected by the dependence analysis alone. Figure 2 shows pseudo code for just such a case in the SEA code. The current state of the analysis has failed to capture this very specific case involving the `v` array. The serialising array dependence is presented to the CFD expert (through the browser) who is then able to inform the tools (by deleting the assumed dependencies) that this is not the case as the v array updates and uses are independent, thereby enabling the `i` and `j` loops to be defined as parallel. This refinement process could be repeated as often as necessary until the user is satisfied with the level of parallelism defined and the toolkit can generate the OpenMP code. The time taken to parallelise the SEA model and generate OpenMP code using the toolkit on an SGI workstation was estimated to be about 2 hours, most of which was spent trying to exploit further parallelism by using the toolkit's interactive browsers. Figure 3 shows an example of the quality of the code generated by the toolkit for the `clinic` subroutine. In this subroutine alone, it highlights the toolkit's ability to define parallelism at a high level (i.e. at the `j` loop level which includes 12 nested parallel loops as well as calls to subroutine `state`), the scope of all scalar and array variables and the automatic identification of THREADPRIVATE common blocks.



```
26 Routines:
addv
bcest
clinic
diag
extended_filename
frees
grids
ocnlst
options
rdrest
rdtopog

1 Parallel loops exploited using directives:
1/1/191: do jc=jsta,jend,1

Current Routine: clinic
:c--------------
:c     calculate
:c     and advect
:c--------------
:           do k
:             vm
:             vm
:             fw
:             fw
:           endd
:c--------------
:c     set k=0 ve
:c     set level
:c     set k=kmc
:c     vertical m
:c     loss to re
:c       smf is t
:c       bmf is t
:c--------------
:             vmf
:             vmf
:             vmf
:             vmf
:             fwb1
:             fwb2
:             fwb1
:             fwb2
:c--------------
:c     set up ind

Loop: 1/1/191: do jc=jsta,jend,1
Reason: Parallel loop exploited without copyin/out variables
Type: Chosen
Hints:
Parallel loop
41 private variables
46 shared variables

Private variables: detmr fx vrest vcor urest ucor lcor fwb2
Shared variables: sine omega c2dtuv hr np dzr ddu ccu
Copyin/out variables:

IO/Exit statements:

12 Nested parallel loops:
clinic:3/3/216: do k=1,kmd,1
state:1/1/258: do k=1,km-1,2
state:3/1/268: do k=2,km-1,2
state:5/1/279: do k=1,km,1
state:6/1/290: do k=1,kk,1
state:7/1/299: do k1=0,kk-2,2

Inside parallel loops:
```

Figure 1 Directive browser displaying loop classification and reasons for loop type

6. Performance of generated parallel code on a test case

An idealised topography for a pseudo global model that extends from 70°S to 70°N was used as the test case for this study. The topography was generated such that the horizontal resolution was modified from the default 182x73 resolution to 722x283, while the number of vertical levels was defined to be 32. It was also indicated that for such a global model cyclic periodic boundary conditions should be applied to the latitudinal boundaries. In the initial parallelisation without any user interaction there were 24 serial and 90 potentially parallel loops (from a total of 114 loops). The toolkit was able to generate code that exhibited a high degree of parallelism as the majority of loops selected for parallel execution were based on the outermost `J` loop. This ensured that for these loops there was enough computational work to offset the overhead in initialising and synchronising the threads. Following further user interaction (described in section 5) the number of potentially parallel loops had increased to 96.

```
nm=1
nc=2
np=3
do loop=1,9999999
  call STEP
enddo

SUBROUTINE CLINIC
do j=jsta,jend
  do i=ista,iend
    do k=1,kmc
       ... = v(k,i-1:i+1,j,nc),
             v(k,i-1:i+1,j,nm),
             v(k,i,j-1:j+1,nc),
             v(k,i,j-1:j+1,nm)
    enddo
    do k=1,kmc
      v(i,j,k,np) = ...
    enddo
  enddo
enddo
```

```
      SUBROUTINE STEP
      nnc=np
      nnm=nc
      nnp=nm
      np=nnp
      nc=nnc
      nm=nn
200   continue
        call CLINIC
        if(mixts.and.eb)THEN
          nc=nnp
          np=nnm
          mixts=.false.
          goto 200
        endif
```

Figure 2 Pseudo code showing the multi-step array accesses for `v`

```
      common /work/dpdx,dpdy,fue,fuw,fvn,fvs,vmf,vtf,fw,fwb1,
     & fwb2,rhoo,rhpo,rhpp,rhop,maskoo,maskpo,maskmo,...
!$OMP THREADPRIVATE(/work/)
...
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(detmr,fx,vrest,vcor,
!$OMP& urest,ucor,lcor,temp2,temp1,fxb,fxa,fuic,fvjc,k,bmf,
!$OMP& uvmag,smf,kmd,kmc,im1,ip1,i,boxar,boxa,jm1,jp1,j)
!$OMP& SHARED(omega,c2dtuv,acor,fkpm,dy4r,dx4r,dy2r,dx2r,grav,
!$OMP& np,dyp125,dxp125,nc,cdbot,nm,ista,iend,dy,dx,jsta,jend)
      do j=jsta,jend
        do i=ista,iend
          if(kmc.gt.0)then
            call state(t(1,i,j,1,nc),t(1,i,j,2,nc),...)
            call state(t(1,ip1,j,1,nc),t(1,ip1,j,2,nc),...)
            call state(t(1,ip1,jp1,1,nc),t(1,ip1,jp1,2,nc),...)
            call state(t(1,i,jp1,1,nc),t(1,i,jp1,2,nc),...)
```

Figure 3 Illustration of the automatic OpenMP code generated by the toolkit

Table 1 shows a breakdown of the frequency and type of OpenMP directives that were generated by the toolkit in the final generated code (referred to as "static opt0"). In summary, there were a total of 20 PARALLEL regions defined in the code, with just 25 of the 96 potentially parallel loops selected for parallel execution. In addition to the 3 REDUCTION loops that were identified there was also another reduction operation identified that did not conform to the OpenMP specification, but this was still handled by the code generator, choosing instead to use the CRITICAL directive. The parallel platform used was a 64 processor SGI Origin 2000, where each processor was a MIPS R12000 with a 300MHz clock speed. The code was compiled with `-mp` `-O3` flags and Figure 4 below shows the performance of the OpenMP code generated from the toolkit as a result of the parallelisation process described above. An efficiency of 87% on 32 processors and nearly 60% efficiency on 64 processors is quite promising given the fairly small, yet essential, user interaction that was needed.

Table 1 Breakdown of OpenMP directives generated by the toolkit

Total number of PARALLEL Regions	20
Total number of OMP DO Loops defined	25
PARALLEL regions with a single OMP DO loop	11
REDUCTION Loops	3
ATOMIC/CRITICAL Sections	1
Regions containing FIRSTPRIVATE variables	2

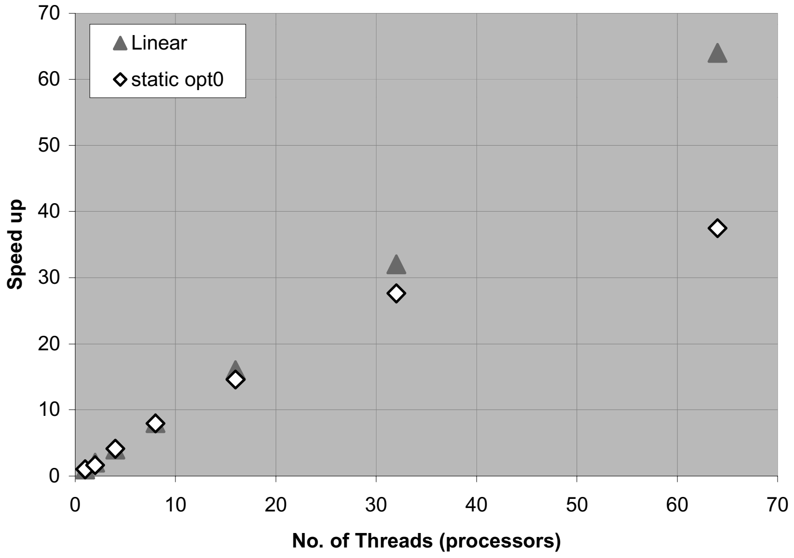


Figure 4 Performance of the OpenMP code generated by the toolkit on an SGI Origin

The possible causes for the degradation in performance could include (i) the execution of serial loops and the effect described by Amdahl's law; (ii) the overhead in starting up and subsequent synchronising of threads; (iii) the granularity of the computation as the number of threads are increased; (iv) the unbalanced workloads assigned to the threads. On examination of the code and with the aid of a profiler (ssrun and prof) a number of optimisations were identified in order to try and improve the performance of the code. Although the toolkit had already completed a fairly comprehensive task in reducing the overhead in starting and synchronising the threads, an additional manual attempt was made to further fuse PARALLEL regions together. This was only possible in a few loops that were contained within IF constructs. As a result of this optimisation the breakdown of the OpenMP directives in the code (Table 2) now show that the number of PARALLEL regions had reduced to 6 but they contained more OMP DO directives. In addition the number of

PARALLEL DO regions were also reduced to 4 as many were fused into larger regions. The performance of the optimised parallel OpenMP code is shown in Figure 5 (static opt1) and shows that there was a small improvement in performance. However, the manual optimisation of fusing the loops did not appear to make a significant contribution to improving the parallel performance.

Table 2 Breakdown of optimised OpenMP directives generated by the toolkit

Total number of PARALLEL Regions	: 10
Total number of OMP DO Loops defined	: 25
PARALLEL regions with a single OMP DO loop	: 4
REDUCTION Loops	: 3
ATOMIC/CRITICAL Sections	: 1
Regions containing FIRSTPRIVATE variables	: 2

Another possible reason for the limited scalability when using 64 threads is the load imbalance that may exist due to the very nature of the problem being solved. Although there is a maximum of 32 cells in the vertical direction, these vary in line with the ocean depth and are defined using an array such that the iterations over the depth (k-iterations) are different for each i,j cell. In this way, computations are performed only in the ocean regions and only to the necessary depth. Therefore, the unbalanced workloads on each processor together with the reduction in effective computation performed as the number of processors increased contributed significantly to the overall performance. In order to try and redress the load imbalance, the j-iterations need to be distributed in such a way as to provide an even balance of ocean grid cells for each thread. The default scheduling of iterations for a parallel OMP DO is referred to as "static" whereby each thread is assigned a chunk or fixed number of iterations on which to operate. An alternative strategy is to dynamically allocate a number of iterations to a thread as soon as it becomes free, thereby ensuring that all the threads are kept busy. The default chunk size is a single iteration for each thread should operate on at any one time. Using a dynamic scheduling process carries a higher overhead than a static one since the allocation of iterations is continually determined during the parallel execution whereas for a static schedule the allocation of iterations is performed once at the start of the DO loop. Figure 5 shows the effect of using a dynamic scheduling process to execute the 5 parallel loops in the time step loop (dynamic opt1). The performance on 64 threads has now increased to an impressive 75%. In spite of the additional overhead in using a dynamic schedule, it is far outweighed by the superior load balancing achieved. Solving the same problem with a coarser mesh and using the message passing version that has a strategy to deal with the load imbalance [4] shows a similar performance trend.

7. Conclusions

In this paper we have demonstrated that parallelisation tools can be beneficial to both CFD specialists and parallelisation experts by performing most of the tasks needed to parallelise real-world application codes typified by the SEA ocean modelling code. This work also demonstrates the crucial need for an interactive system where the toolkit not only provides valuable information to the user, but also that the user can offer knowledge to supplement and direct the parallelisation process. The interaction is carried out through the use of browsers that filter the vast information acquired by the toolkit and this is presented to the user in a more focused manner. It has also been shown that the generated code is efficient and scalable and that the parallelisation can be done extremely quickly. In comparison, the manual parallelisation of the code using OpenMP has not been undertaken to date because of the prohibitive cost of effort and the expertise that is needed to complete the task.

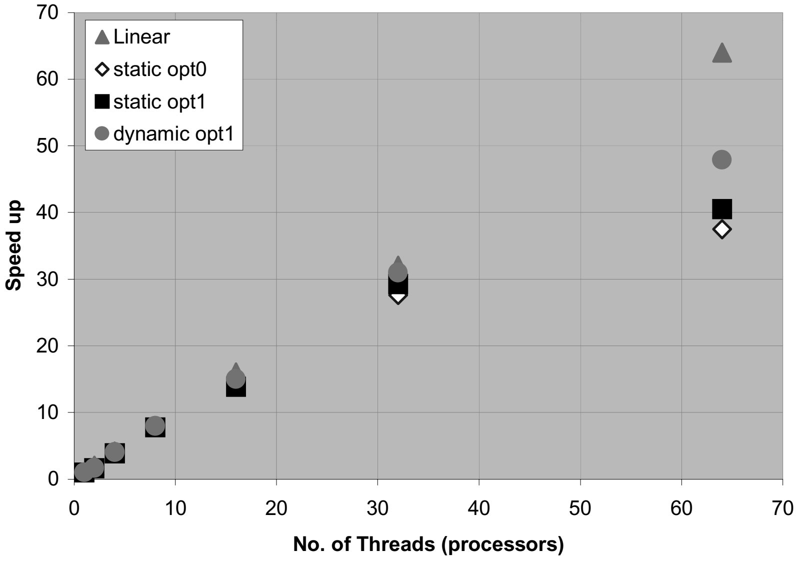


Figure 5 Performance of the optimised OpenMP code generated by the toolkit on an SGI Origin.

8. Acknowledgements

The authors wish to thank the developers of the SEA model code for making the source code available. The authors also wish to thank the many people at the University of Greenwich and NASA Ames who have helped in the development of this toolkit, in particular to Haoqiang Jin (NASA), Jerry Yan (NASA) and also Mallick Arigapudi (SGI) for allowing use of the SGI Origin 2000.

References

1. Evans E.W., Johnson S.P., Leggett P.F., Cross M., Automatic and Effective Multi-Dimensional Parallelisation of Structured Mesh Based Codes. Parallel Computing, 26, 677-703, 2000.
2. Evans E.W., Johnson S.P., Leggett P.F. and Cross M., The automatic code generation of asynchronous communications embedded within a parallelisation tool. Parallel Computing, 23, 1493-1523, 1997.

3. Jin H., Frumkin M., and Yan J. Automatic generation of OpenMP directives and it application to computational fluid dynamics codes. International Symposium on High Performance Computing, Tokyo, Japan, p440, 2000
4. Beare, M.I. and Stevens D.P., Optimisation of a parallel ocean general circulation model, Annales Geophysicae, 15, 1369-1377, 1997.
5. http://www.mth.uea.ac.uk/ocean/SEA/
6. Ierotheou C.S., Johnson S.P., Cross M. and Leggett P.F., Computer aided parallelisation tools (CAPTools) - conceptual overview and performance on the parallelisation of structured mesh codes. Parallel Computing, 22, 197-226, 1996.
7. Johnson S.P., Ierotheou C.S. and Cross M., Computer Aided Parallelisation Of Unstructured Mesh Codes. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Editors H.R.Arabnia et al, publisher CSREA, vol. 1, 344-353, 1997.
8. Johnson S.P., Cross M. and Everett M., Exploitation of Symbolic Information In Interprocedural Dependence Analysis. Parallel Computing, 22, 197-226, 1996.
9. Leggett P.F., Marsh A.T.J., Johnson S.P. and Cross M., Integrating user knowledge with information from parallelisation tools to facilitate the automatic generation of efficient parallel Fortran code. Parallel Computing, 22, 259-288, 1996.
10. Johnson S.P., Ierotheou C.S. and Cross M., Automatic parallel code generation for message passing on distributed memory systems. Parallel Computing, 22, 227-258,1996.
11. http://www.openmp.org/
12. Webb, D.J., An ocean model code for array processor computers, Computers and Geosciences, 22 ,569-578, 1996.
13. Pacanowski, R.C., MOM2 documentation, user's guide and reference manual, GFDL Ocean Group Technical Report No.3, GFDL/NOAA, Princeton University, Princeton, NJ, 1995.
14. Rodrigues J.N., Johnson S.P., Walshaw C. and Cross M., An automatable generic strategy for dynamic load balancing in parallel structured mesh CFD code., Parallel Computational Fluid Dynamics, D.Keyes Editor, 345-353, 2000.
15. Mesinger F. and Arakawa A., Numerical methods used in atmospheric models, GARP publications series, No.17, World Meteorological Organisation, 1976.
16. Amdahl G. Validating the single processor approach to achieving large scale computing capabilities. AFIPS conference proceedings,30, 83-485, 1967.