CAPTools Project: Evaluation and Application of the Computer Aided Parallelisation Tools

FINAL REPORT

David O'Neal
National Center for Supercomputing Applications
University of Illinois, Champaign, IL

Richard Luczak
University of Tennessee, Knoxville
Aeronautical Systems Center, Wright-Patterson Air Force Base, OH

Michael White
Ohio Aerospace Institute
Air Force Research Laboratory, Wright-Patterson Air Force Base, OH

Abstract

This report extends a previous study of the Computer Aided Parallelisation Tools software package (http://captools.gre.ac.uk/) developed by the University of Greenwich, London.[1] Product effectiveness, deployment and training requirements, and the more general issue of continued support for the project are all addressed by this installment.

Scalability and the level of effort required to achieve it are considered first. An informative inventory of essential features and basic usage strategies follows directly. A review of project accomplishments leads into a discussion of appropriate courses of action, and key recommendations are made in closing.

[1] D. O'Neal, R. Luczak and M. White, *CAPTools Project: Evaluation and Application of the Computer Aided Parallelisation Tools*, proceedings of the DoD High Performance Computing Modernization Program Users Group Conference, Monterrey, CA, 1999.

Overview

During the past year, we have continued our work with the University of Greenwich on an evaluation of their CAPTools product. ASC PET funding for the project was awarded to the University of Illinois and the University of Tennessee in May of 1999, but negotiations with Greenwich stalled at an inopportune time and as a result, no funding was provided to UG for contract year 4 (CY4). However, UG was able to maintain a basic level of support through attenuation of CY3 funds. As of this writing, it appeared likely that all three universities would receive awards for CY5.

The origins of the ASC PET CAPTools Project were examined in our first paper.[1] This publication may be viewed at the NCSA CSM Group website.

http://www.ncsa.uiuc.edu/EP/CSM/publications/1999/UCG99_CAPTools.pdf

Unless noted to the contrary, all references to the CAPTools product correspond to a single integrated IRIX installation consisting of (i) the latest *capo* executable (1.0.3 Beta-022) furnished by NASA Ames and (ii) the message passing libraries, scripting tools, and manuals from the standard package developed by the University of Greenwich (2.1 Beta). All timing tests were performed on the Origin clusters at NCSA and ASC.

We begin this final report with a brief overview of Amdahl's Law of Parallelization. This section serves two purposes. It lends definition to basic terms used throughout the article and it establishes a method for evaluating the effectiveness of CAPTools-generated parallel source codes. Notes describing the basic care and feeding of the software (Usage) lead into a discussion of more severe limitations (Caveats). Fundamental descriptions of each evaluation code precede more detailed observations regarding scalability (Applications). Significant findings are then summarized and final recommendations are made in closing.

Amdahl's Law of Parallelization

The premise of Amdahl's Law is that every algorithm has a sequential part that ultimately limits the speedup that can be achieved by a multiprocessor implementation. In this context, *speedup* is a ratio of execution times for single processor and multiprocessor runs. The classical form of Amdahl's Law is usually stated as follows:[2]

> *If the serial component of an algorithm accounts for 1/S of its execution time, then the maximum speedup that can be attained by running it on a parallel computer is S.*

More detailed analyses generally begin with a characterization of a program solely in terms of its operation count. It is assumed that some portion q of these operations can be executed in parallel by p processors and that the remaining operations must be executed sequentially.

For a given problem size, if the total operation count and single processor performance of the code are presumed constant for any value of p, then a simple expression for *speedup* may be written solely in terms of p and q.

$$S(p,q) = (q/p + 1 - q)^{-1}$$

This is Amdahl's Law of Parallelization. It is important to note that S reflects an *idealized speedup* value, also known as *false speedup*, because parallel executions are known to involve overheads that invalidate the aforementioned presumptions. Nevertheless, this function does serve as an upper bound for more realistic speedup models that incorporate operation counts and performance levels that do depend on the partition size.[3]

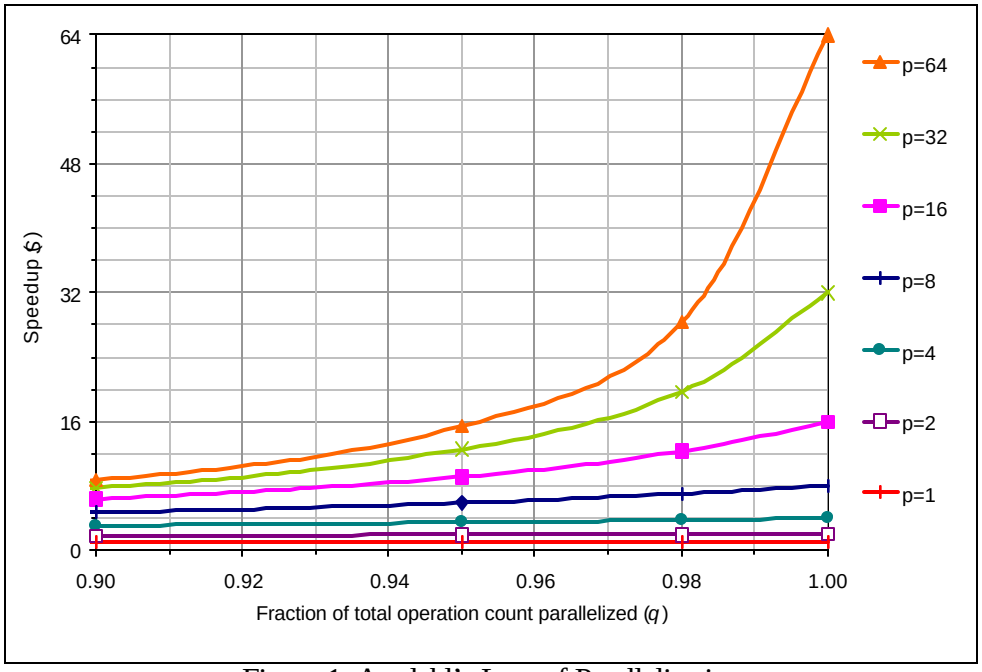

Figure 1. Amdahl's Law of Parallelization

Estimated speedups derived from timing measurements for each of our evaluation codes are used to estimate the portion of the total operation count parallelized by CAPTools. Reference is made to the curves of Fig. 1. Note that in order to make the interesting part of the speedup curves easier to interpret, values for $q < 90\%$ have been cropped from the chart. Speedups associated with the missing portions of these curves are considered to be unacceptably poor and in some cases are only reported as such without accompanying data. Reference is also made to the *efficiency* of parallel codes. This value is taken to be the ratio of speedup to the processor count, or $e = S/p$. Readers may also be interested in another interpretation of these concepts that was used to develop throughput estimates for computers featuring hierarchical memory systems.[4]

Usage Notes

This section describes limitations and general usage characteristics associated with the current version of the CAPTools product. Both new and experienced users should find it informative. Note that at the time of this writing, the release of Beta version 2.2 was imminent.

CAPTools supports standard Fortran 77 syntax (an F90 parser is in development). Fortran statements that deviate from F77 standard may need to be rewritten. An unusual I/O format and a number of instances of array syntax were encountered and (justifiably) rejected by CAPTools during testing.

Source files containing conditional compilation directives must be preprocessed prior to loading. The CAPTools parser treats compiler directives as comments and therefore all program statements are loaded. Unless an error occurs as a side effect of some unwanted inclusion, no warning is provided to the user.

A new feature called the Undefined Symbol Browser was added recently. In the past, all external references had to be resolved before a source code could be loaded successfully. For example, stubs corresponding to library calls had to be provided by the user, and then these same stub routines would have to be commented back out of the CAPTools-generated source code prior to compilation.

The new USB window supports examination and control of undefined symbols and their effect on subsequent dependency analyses. Help is currently not available for this feature, but we expect that usage details will appear in the next release. If the default assumptions are accepted, the Analyser window is automatically raised. This is also the case when a source without undefined symbols has been successfully loaded. A better choice might be to raise the READ Knowledge editor as its use is indicated first, or perhaps no default selection should be made.

User knowledge must be added prior to analysis in order to be considered. Integer logic associated with variables appearing in READ statements can be simplified merely by indicating the sign of these inputs. More complex criteria can also be developed using standard F77 syntax. Of course it may not be possible to make such determinations for all variables, but in general, user knowledge should be specified whenever possible.

After an input source has been loaded, the Analyser button becomes active. Two distinct algorithms for determining dependencies are supported (Banerjee, Omega). Any combination of three dependency test options (Scalar, Exact, Disproofs) may be performed within any combination of three contexts (Interprocedural, Knowledge, Logic). A group of preset selections is also provided (Basic, Intermediate, Full). Ultimately a full analysis should be completed, but when working with large sources, an incremental approach aimed at memory conservation may be used. Performance of the Analyser may be improved by (i) saving the database, (ii) exiting

and re-entering the system, and (iii) reloading the database file upon completion of each preset level of analysis. The Knowledge Base window should also be reviewed and edited between stages. Note that changes to the Knowledge Base are applied to subsequent analyses.

The dependency analysis kernel presumes static allocation of all variables (as if SAVE statements were implemented everywhere). The use of this programming technique is often found in older programs and it is still supported by modern compiling systems so it can't be ignored. However, it should be possible to deny this consideration. Among other things, static allocation implies that the addresses of local variables are fixed and so when a subroutine (function) is exited, its local variables persist and therefore all calls that might be executed more than once will spawn dependencies that should be removed if static allocation is not presumed.

Configurable windows are provided for filtering and editing dependencies. The pruning of false dependencies might be regarded as an option, but the scalability of the final code can be greatly affected by the presence of such dependencies. The function of the Directives Browser (OpenMP) and the Dependency Graph Viewer (message passing) should be well understood before electing to forego their use. Currently, the only guidance provided for any of the directives-oriented features appears in the form of a README file.

An OpenMP source may be generated immediately after completing any dependency analysis, but prior use of the Directives Browser is highly recommended. After loading a source code and completing an analysis, users should proceed directly to the Directives Browser. Dependencies inhibiting loop parallelization should be viewed and edited there. Partitioning is not required.

Conversely, message passing codes must be partitioned and communication logic must also be developed within the environment before a new source code can be generated. The compilation and execution of CAPTools-generated message passing codes is also somewhat more demanding of the user, but the difference is approximately the same as that between compiling and running any given OpenMP source and its MPI equivalent.[5, 6]

CAPTools-generated message passing codes contain calls to various library routines developed by UG. At the core of this design lies CAPLib, a library of communication routines built upon various message passing primitives (MPI, PVM, shmem). This wrapper-like layer was implemented before MPI had emerged as a de facto standard interface. A nice paper describing CAPLib was recently made available at the CAPTools website.[7]

Caveats

Consideration of the following commandment for CAPTools users is of utmost importance:

Thou shalt not save the database after generating source code.

In the current version of CAPTools, the contents of the database are corrupted during source dumps associated with code generation. Although it is possible to continue to working within a session after generating source code, it is not recommended. Oddly enough, this rather severe shortcoming has remained undocumented for at least six months.

Another insidious bug originates from the so-called Openwin libraries that are an integral part of the CAPTools environment. After successfully starting and then exiting a CAPTools session, subsequent attempts to start new sessions unexpectedly terminate during initialization. The most reliable workaround for the problem is to touch the libxview.a and libolgx.a library files, but of course only the owner of these files can apply it.

Because of the aforementioned database corruption problem, users are forced to exit and re-enter CAPTools after generating source code. The Openwin problem then will often prevent immediate re-entry. This is definitely not a good situation where new users are involved. UG has been working on a new build of the windowing libraries, and a File menu feature that supports multiple database loads is in the works. We fully expect these items to be resolved soon.

A well-documented characteristic worthy of mention here is that CAPTools allows progression through the stages of the parallelization process in a forward manner only. Retreating from the current state of the database depends entirely on the availability of a previously saved version of the database. Users should consider the costs and benefits of preserving numerous versions of the database at various key points of progress, i.e. just after completing some significant task.

A methodical approach based on the generation and validation of source codes following a reasonable number of database modifications should be adopted. The objective is to balance the cost of performing unessential validations against the loss of work associated with a given regression step, presumably brought about by invalid results.

Applications

Basic characteristics of the simulation codes selected for evaluation are followed by detailed descriptions of the individual porting efforts and scalability measurements taken with the final versions of the new parallel application codes.

We begin this section with a table of information summarizing each of our evaluation codes in terms of its geometry, subroutine and function count, number of statements before and after the application of CAPTools and the cases (partition sizes) for which results were validated.

Application	Model Geometry	Subroutines and Functions	F77 Lines In	OMP Lines Out	MPI Lines Out	OMP Validation	MPI Validation
R-Jet	3D	63	7655	7471	19284	1,2,4,8,16,32	1,4,9,16,25,36,49,64
FDL3DI	3D	76	9964	15330	11993	1-8,16,32	1-4
N-Body	3D	2	195	207	376	1-6,8,16,32,64	1-6
PFEM	2D	16	2073	1934	2357	1-8,16,32,64	1-8,16,32,64

Table 1. Summary of general information regarding CAPTools test applications

The first two applications (R-Jet and FDL3DI) were provided by AFRL. The others (N-Body and PFEM) are research codes out of Rice University and Carnegie Mellon University respectively.

In the following sub-sections titled by application name, brief descriptions of the functionality of each application precede related details regarding the porting effort. Speedup curves are also provided for each experiment.

R-Jet
Air Force Research Laboratory

R-Jet is a hybrid, high-order, compact finite difference spectral method for simulating vortex dynamics and breakdown in turbulent jets. While the code is explicit in time, the compact finite difference scheme requires inversion of tridiagonal matrices, giving rise to the same sort of parallelization problems exhibited by implicit methods.

The R-Jet program was only recently implemented as a serial code and so a parallel version had not been written when CAPTools became available. However, R-Jet was designed to support the solution of radial and axial derivatives by either an explicit differencing scheme or by the compact method described above, which ultimately proved to have a profound effect on scalability.

R-Jet employs a programming technique in which complex-valued arrays are mapped onto pairs of real arrays for subsequent input to FFT routines. This approach was problematic for earlier versions of CAPTools. Dummy routines were needed to maintain the geometry of the data structures. CAPTools was later amended to eliminate the underlying restriction, but we

continued to maintain the new code because its use resulted in a significant reduction in the time required to complete a dependency analysis.

CAPTools was overly cautious in its assessment of the utilization of a number of work arrays. The pruning of a number of false dependencies was performed with the Dependency Graph browser (see Usage section). Another unexpected result was the placement of message passing statements outside of existing IF-THEN constructs whose use was required for proper masking of data exchange. For some of our test problems, this resulted in the generation of spurious messages. It was fixed by applying the aforementioned IF-THEN statements to the message passing calls in the output source code.

The most significant deficiency observed with respect to the R-Jet code was the way in which the collection of summary information needed to generate the final solution and restart files had been implemented. The code generated by CAPTools represented a sort of *bucket brigade* arrangement in which variables were handed down one at a time from processor to processor until they reached PE0 and were written out to disk. For larger partition sizes, problems with some of the default MPI limit settings were encountered.

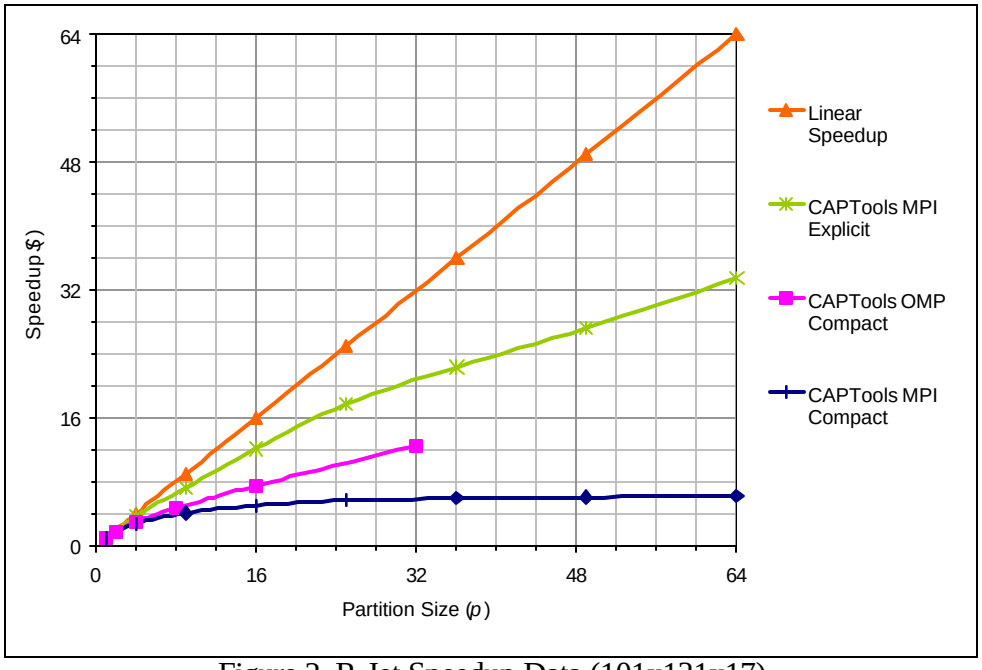

Figure 2. R-Jet Speedup Data (101x131x17)

We chose to replace the bucket brigade code with a simpler and more efficient implementation in which each processor was programmed to open a uniquely named temporary file into which local data was subsequently streamed using unformatted block writes. The master processor (PE0) was then used to gather and construct the global solution and restart files by post-processing the

succession of temporary files. The new method outperformed the CAPTools-generated code for all cases involving more than a few processors, where it remained competitive.

The MPI curves shown by Fig. 2 illustrate the scalability of the resultant message passing code for a 101x131x17 test case partitioned in the first two dimensions. Speedups for the compact differencing scheme level off at around 25 processors due to the serial dependence implied by the tridiagonals, but the efficiency of the explicit differencing algorithm exceeds 50% through full range of partition sizes tested ($p = n^2$ for n ranging from 1 to 8). The estimated fraction of the total operation count parallelized by CAPTools exceeded 97% for the latter case.

Reference is also made to a set of OpenMP experiments performed last fall with the compact differencing source and an earlier version of the CAPTools 2.1 Beta release. Results were presented by Greenwich at Supercomputing 1999.

http://www.gre.ac.uk/~lp01/sc99/openmp/page2.html

The OpenMP (OMP) curve of Fig. 2 was drawn from the same data that appears on the reverse side of the UG handbill. It indicates that between 92% and 95% of the total operation count was parallelized by CAPTools.

FDL3DI
Air Force Research Laboratory

The Flight Dynamics Laboratory 3D Implicit code is used to study aeroelastic effects. It was developed by the AFRL Basic CFD Research Group at Wright-Patterson Air Force Base. FDL3DI is a high-frequency Navier-Stokes model featuring a one-dimensional structural solver component. Collective use of numerous variations of the FDL3DI codes maintained by AFRL research staff is heavy. Interest in developing a recipe for converting all of these sources to parallel form is what led to the inclusion of the FDL3DI code in this study.

An existing parallel implementation based on a Chimera overset grid decomposition scheme was modified by Wright State University to run in a serial fashion for our experiments. The original multiprocessor design by K. Tomko (WSU) was quite clever, taking full advantage of the knowledge that FDL3DI was already configured to handle overset grids. By characterizing arbitrary mesh decompositions as overset grids, the task of parallelizing FDL3DI was reduced to a geometry problem. A new tool capable of imposing such decompositions had to be developed from scratch, but very few changes to the original code were required.

The serialized WSU code was then used as input to CAPTools. Both OpenMP and message passing sources were produced. The Directives Browser showed that all but a few loops were automatically parallelized without any additional work, but as indicated by the speedup curves of Fig. 3, the resulting code was only able to keep a few processors busy. However, the effort required to achieve this improvement was insignificant.

Work on the message passing version of FDL3DI was even less satisfactory. After completing a full analysis, an array index was selected for partitioning. CAPTools needs this information to create a template for affecting other similarly dimensioned program arrays including those of lower rank. Upon completing this phase, maskings of all newly partitioned arrays were computed and the corresponding communications statements were generated. Attempts to apply the overlapping communications and memory reduction features resulted in segmentation faults.

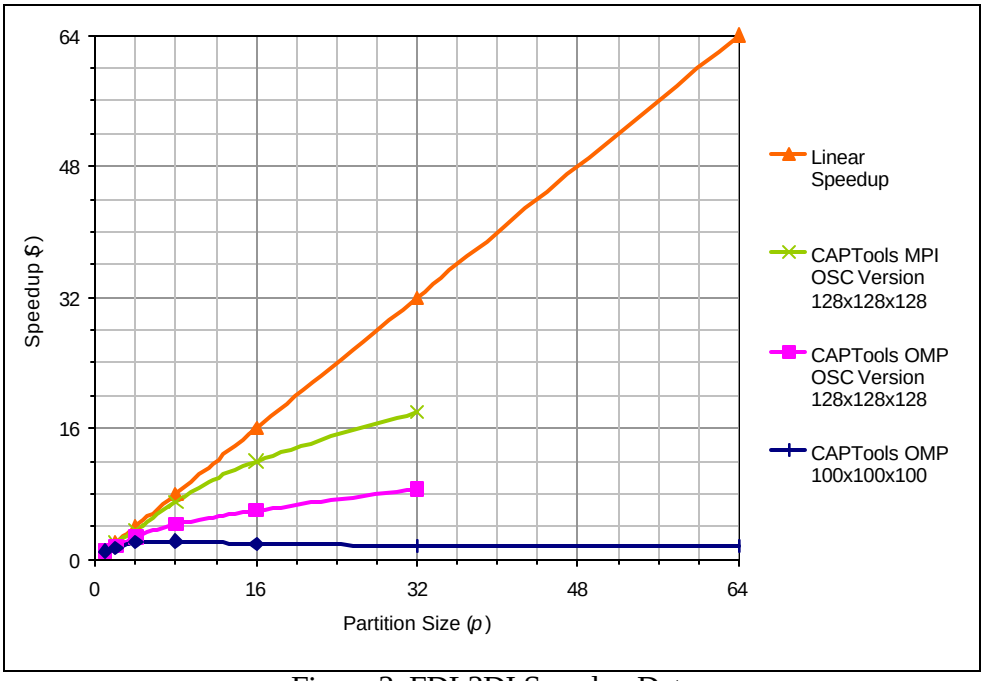

Figure 3. FDL3DI Speedup Data

Tests of the message passing executables created from 1- and 2-dimensional partitionings of the data were not successful. Iteration timings that consumed about a minute of CPU while running on a single processor required over 2 minutes when 2 processors were used, and over 4 minutes when 4 processors were used. The need to increase one of the default per-processor limits, i.e. MPI_MSGS_PER_PROC, gave indication that a large number of small messages were being dumped onto the network. This in turn suggests the presence of a data locality problem.

Unnecessary replications meant that the physical node memories were also being oversubscribed. Nevertheless, all available network topologies were tested. Sources generated from default and pruned dependency graphs were checked. The pipe configuration coupled with the source produced from the pruned graph resulted in the best timings (approximations noted above). A negative trend was clearly established and so testing was suspended at this point. An acceptable level of performance could not be achieved for the given input source.

But reference is also made to another set of experiments with a newer version of the FDL3DI code conducted by members of the Ohio Supercomputer Center research staff. Results were presented by the University of Greenwich at Supercomputing 1999 in the form of two handbills.

http://www.gre.ac.uk/~lp01/sc99/openmp/page2.html
http://www.gre.ac.uk/~lp01/sc99/mp/page2.html

The OSC tests were carried out with a version of the FDL3DI source that features modified data structures in the form of supplemental workspace arrays used to carry out computational sweeps along pairs of axes. A slightly finer discretization (128x128x128) of the same problem was solved. The dependency graph pruning and data partitioning techniques applied to each case were essentially the same, which meant that the effort given to tuning each of these codes was also approximately the same. Note though that the results of the OSC tests were much more positive than ours.

Timing data presented for the OSC OpenMP tests suggests that about 90% of the total operation count was parallelized by CAPTools. Speedup data for the OSC message passing runs indicates that 96% to 98% of the total operation count was parallelized by CAPTools. This case provides clear indication that the efficiency of CAPTools-generated source is highly dependent on the way in which the original code allocates and uses its data structures.

N-Body
Rice University

N-Body is a three-dimensional n-body model with constraints. It is used to solve problems arising in electromagnetic applications. N-Body determines equilibrium positions for an arbitrary set of points interacting on the surface of a unit sphere according to a $1/d^2$ force law (here d denotes the distance between two points). The result is a uniform distribution of points in three dimensions.

A serial version of the N-Body code was used as input to CAPTools. A hand-written MPI code was also developed for comparison purposes. The dependency analyses and the generation of the OpenMP code were accomplished in a fully automatic way. Partitioning phases associated with the message passing model required a minimum of user input. Both CAPTools-generated codes produced correct output.

Identical initial distributions and convergence criteria were used for all experiments. Measurements were taken for problems consisting of 100 and 1,000 points running on various partition sizes. Approximately 10^5 and 10^6 iterations respectively were required to achieve equilibrium. Note that N-Body doesn't involve a computational mesh. Positional updates for each particle depend only on the governing ordinary differential equation and therefore, a full topology was specified for the message passing runs.

Scalability measurements for the OpenMP and handwritten MPI versions of the code are presented in Fig. 4. Results associated with the OpenMP model were excellent. For the smaller problem, good scalability was observed through 16 processors. The larger problem scaled well through 64 processors. The estimated portion of the total operation counts parallelized by CAPTools ranged from 90% to 99%.

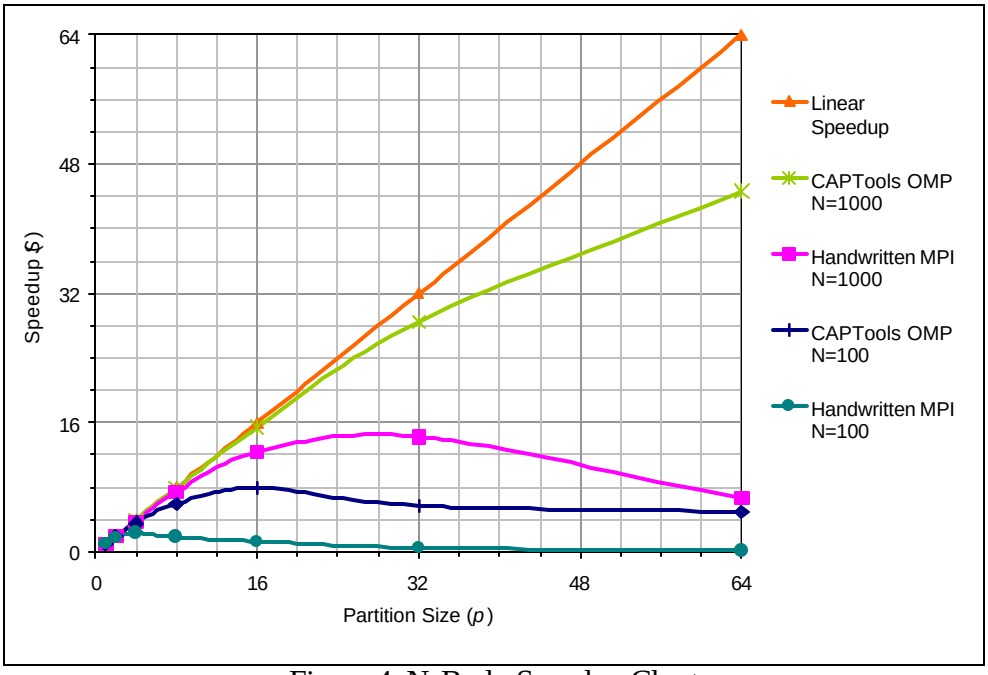

Figure 4. N-Body Speedup Chart

A few runs of the CAPTools-generated message passing code were also performed for various partition sizes. Although correct results were produced, we were unable to achieve an acceptable level of scalability for either problem size. These results are not shown in Fig. 4.

The N-Body tests revealed a specific limitation of the current CAPTools message passing model. Simple communication patterns may be implemented in overly general fashions. In one case, we found that data arrays were being transmitted a single element at a time inside of a loop. In another, a reduction operation wasn't recognized as such. It had been written in terms of CAPLib primitives, e.g. CAP_SEND and CAP_RECEIVE.

These items imply that hand tuning of message passing source codes may be required in order to realize acceptable levels of performance. Where the communication of data arrays is involved, at least some of these changes may be completely straightforward and very effective, but collective operations are generally more complicated. Knowledge of message passing concepts is certainly prerequisite for such explicit optimizations.

We should also mention that the memory reduction feature associated with the message passing model was successfully applied. Note that if memory reduction had not been applied, all data arrays would have been replicated across all processor memories. While data replication might be acceptable early in the design cycle, it doesn't represent a valid approach for a final product.

PFEM
Carnegie Mellon University

The Parallel Finite Element Method code is a research-level finite element code suitable for solving highly nonlinear boundary value problems in two dimensions. The use of nonconforming finite element geometries facilitates a simple domain decomposition strategy based on two-colorings. Each subdomain consists of approximately half of the total number of elements. All of the element calculations within each subdomain are effectively uncoupled.[8]

The PFEM model was selected for evaluation for a number of reasons. A hand-coded directive-driven source code based on the Cray Research microtasking model (CRI) was available for comparison purposes. PFEM featured a complete set of validation problems and timer outputs that saved us a bit of time and effort. We were also interested in testing CAPTools' ability to deal with automatic array allocation, a feature that figured prominently in the design of PFEM.

The CRI parallel source code was used as input to CAPTools. Existing directives were converted into CAP comments by the parser. Unresolved references to a couple of system functions (SECOND and GETENV) were encountered and properly scoped by the new Undefined Symbol Browser. User knowledge reflecting a positive element count and tolerance value was added, and then a full analysis was performed.

A few circular (self-dependent) references associated with the static treatment of local variables were pruned from the dependency graph (see Usage section), the database was saved, and then just prior to exiting the program, a new OpenMP source code was generated (see Caveats section). Following a simple makefile change (the addition of a compiler switch) we were able to build and run the new OpenMP program. Work on the message passing model, however, proved to be much more complicated by comparison.

Upon restarting CAPTools, the database that had been saved prior to generating the OpenMP source was loaded, giving us a bit of a head start with the message passing version. Partitioning was specified with respect to the first index of a single array, the geometry of which was representative of all of the primary data arrays. An unstructured mesh was then selected and partitioning was initiated. Within the context of CAPTools, indirect addressing corresponds to the presence of an unstructured mesh. Furthermore, unstructured meshes correspond to fully connected messaging topologies.

After confirming array bound values for a short list of references, all of the primary data arrays were automatically partitioned. A few indexing arrays were necessarily deleted from the list of

partitioned arrays. Indeed, when this step was omitted, the resultant executable produced incorrect output (this was recently repaired by UG). Masking and communication calculations were completed using the default system settings, the final database was saved, and a new message passing source code was created just prior to exiting CAPTools.

Our attentions then turned to the set of *cap* utility scripts, e.g. *capmake* and *caprun*, that were developed to support proper compilation and execution of CAPTools-generated message passing codes. We chose to modify our PFEM makefile to call the *capmake* script with the desired arguments. A minor problem was observed while attempting to pass options through to the underlying compiler, but a suitable workaround was found and we were then able to complete the initial build. The result was a single executable that could be configured at run time to accommodate any problem or partition size.

Follow-up experiments with the available message passing optimization buttons were also carried out. Enabling the use of asynchronous calls (overlapping communications) produced no changes in the output source code. Attempts to apply the memory reduction feature resulted in segmentation faults, but we later found that this feature is not supported for unstructured meshes in the current release and so the button should not have been active. See the Release Notes page for CAPTools version 2.0 Beta dated October 23, 1998, for further details.[9]

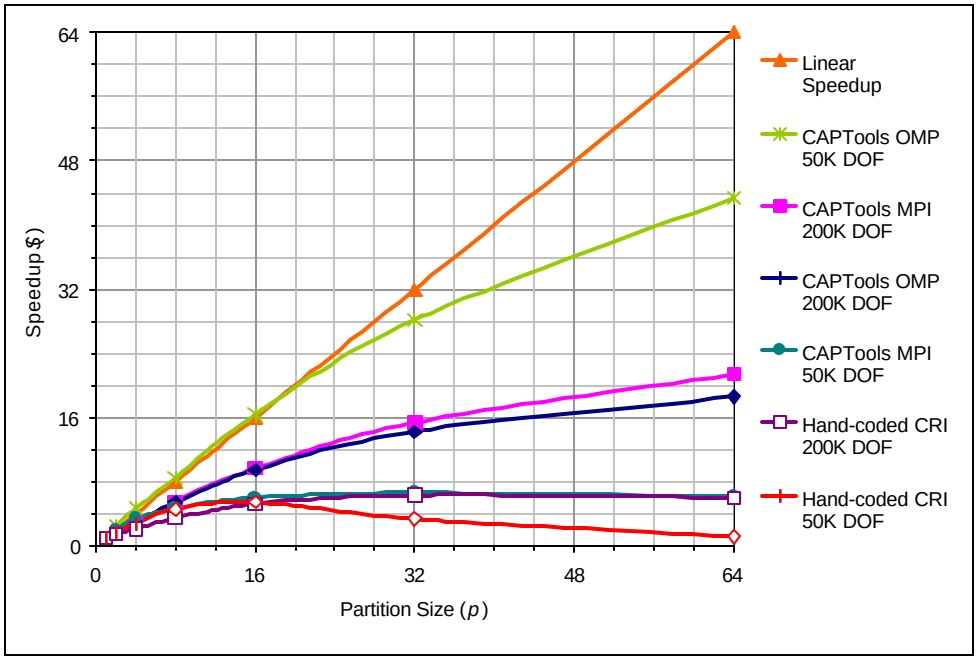

Figure 5. PFEM Speedup Chart

Initial timings were surprising. The message passing version was quite a bit more efficient than the corresponding OpenMP code, particularly for larger partition sizes. Estimated fractions of the

total operation count parallelized by CAPTools were in the range of 95% to 97% for the message passing model, but only around 90% for the OpenMP generator. Because PFEM was originally designed for execution on a multiprocessor computer, we had expected more from the OpenMP port. This led us to take a closer look at the source from within the Directives Browser.

Cursory inspection revealed that a pair of scattering loops had not been parallelized because of the presence of indirect addressing arrays. But these were permutation arrays that did not involve any true dependencies so the inhibitors were explicitly removed using the Directives Browser and a new source was generated. The desired effect was then implemented automatically.

Scalability of the new OpenMP executable was much improved. The estimated fraction of total operations parallelized by CAPTools was now exceptional, ranging from 95% to 98%. Note the superlinear behavior of the 50K DOF OMP curve shown in Fig. 5.

Summary

The CAPTools product was evaluated with respect to its ability to deal with two significant DoD codes and two academic research programs. A total of 15 cases were considered including three datasets produced by the Ohio Supercomputer Center. At least some measure of speedup was observed for all seven of the CAPTools-generated OpenMP codes while only 5 of 8 CAPLib experiments met with similar results (speedup data for the other 3 cases was not presented). The effort required to achieve these results varied from case to case, but the OpenMP tests were consistently less demanding both in terms of required parallel programming expertise and time to complete.

The memory reduction feature could not be applied to one of the test codes (PFEM) due to the presence of an unstructured mesh. In another case (FDL3DI), attempts to apply the feature resulted in segmentation faults. When memory reduction is not (or cannot be) applied, the entire dataset is replicated across the partition.

The CAPLib model requires much more from the user in every way. Detailed knowledge of the input source code is presumed. Issues related to data distribution must be considered prior to any other parallelization steps. A thorough understanding of parallel programming concepts is required in order to guide CAPTools through the process.

CAPLib source code is also much more difficult to debug, even for the experienced programmer. The absence of a compact and detailed CAPLib document describing the application interface contributes to the problem. Unwanted modification to the original formatting contained by input source codes renders output from the *diff* utility useless. Nearly all statements are affected. As a result, nontrivial changes can only be observed through side-by-side comparison of entire programs, and that is an extremely tedious task.

The effectiveness of the CAPLib model is also highly dependent on the presence of a well-defined mesh. Our work with the N-Body code made this perfectly clear. In contrast, OpenMP codes are not subject to this restriction, but are instead dependent on the presence of large arrays. Results for the OpenMP version of N-Body were exceptional.

For two of our test codes, CAPTools implemented block-oriented communications as loops around single element transmissions. This limitation is reportedly slated for improvement in the next release. Until then, users may need to modify message passing output source codes *by hand* in order to achieve acceptable levels of performance.

We also observed in some cases that CAPTools did not recognize situations that called for the use of collective communication calls. Instead, less efficient code was written in terms of CAPLib primitives. This is a much more difficult problem that is not currently scheduled for near-term improvement. Here we have indication of a long-term need to understand the CAPLib interface and implement optimizations explicitly when working with message passing models.

Recommendations

Users faced with the challenge of parallelizing a FORTRAN 77 code should certainly consider the CAPTools package first. Work may be required to successfully load or analyze any given input file, but once accomplished, the value of the information represented by the Call Graph, the Dependency Graph and the Directives Browser is significant. Transformation Menu options may also be used to affect serial optimizations such loop interchanges, splits and skews. The effort required to achieve this minimal level of progress is considered worthwhile for any porting project.

Where the option exists, users are advised to work with the OpenMP model. Compilers supporting OpenMP are usually associated with distributed shared memory platforms and so we are indirectly recommending that users target DSM machines like the SGI Origin 2000. The simplicity of this approach is most attractive. OpenMP source may be generated immediately after completing an analysis. If formatting changes are neglected, the output files look very much like the input files, thus reducing the impact of any subsequent debugging or tuning efforts. The build process is also completely straightforward. For two of the four test codes considered by this evaluation, the corresponding OpenMP executables were the most efficient. As previously noted, this is currently the only practical choice for meshless codes. The CAPO (CAPTools OpenMP) program out of NASA Ames has been a robust and practical tool for many months now. A CAPO installation based on the configuration developed at NCSA should be deployed across all of the HPCMP centers.

We are less enthusiastic about the CAPTools message passing model. It is most effective in the hands of an experienced programmer, but this type of DoD user is not inclined to develop message passing logic that depends on a set of proprietary libraries. Novice programmers may

find the CAPLib interface less complicated than MPI for example, but such users aren't well suited to the demands imposed by the model, especially when debugging or tuning is required (as is all too often the case). The former situation isn't likely to change, and the possibility of a member of the latter group producing a significant result within a reasonable amount of time is also considered to be remote.

Therefore, we recommend that funding to support the development of the OpenMP model should be continued. Deliverables associated with the current version of the UG proposal for CY5 should be interpreted accordingly. Specific items of interest include core support, updates of the Openwin libraries, repair and enhancement of database management features, and improvements to the documentation system with respect to OpenMP.

On the PET side of the equation, we recommend the award of an additional 15-20% FTE position to develop a comprehensive success story over the next 9 months. The *suppressor model* and one other highly visible DoD code would be targeted. This relatively small adjustment to the budget would yield a significant feature presentation for ASC PET.

Support for the implementation of a comprehensive CAPTools tutorial based on the OpenMP model is also recommended. Another 5% FTE (minimum) is indicated. NCSA and UTK are each qualified to produce such a document. On a related note, both have recently made new CAPTools resource pages available.

http://www.ncsa.uiuc.edu/EP/CSM/software/captools/
http://www.asc.hpc.mil/PET/PTES/CAPTools/

As a final remark, the collaboration between UG and NASA Ames has been very successful, but we would still like to see a single, integrated version of the CAPTools package emerge. The organization behind the product should present itself to the community in the simplest possible terms. Impediments associated with acquisition, installation, and maintenance of complementary versions of the same product should certainly be removed. Further consideration of interoperability issues (between models) is also warranted.

Acknowledgments

We wish to express our sincere thanks to Professor Mark Cross, Drs. Constantinos Ierotheou and Steven Johnson, and Peter Leggett of the Parallel Processing Research Group, University of Greenwich, London, England, UK, and to Dr. Henry Jin of the NASA Ames Research Center, Moffett Field, CA, without whose help this project would not have been possible.

We would also like to thank Drs. Miguel Visbal and Raymond Gordnier (AFRL) and Professor Karen Tomko (Wright State University) for their contributions to the project including the provision of the FDL3DI application.

References

1. D. O'Neal, R. Luczak, and M. White, *CAPTools Project: Evaluation and Application of the Computer Aided Parallelisation Tools*, proceedings of the DoD High Performance Computing Users Group Conference, Monterrey, CA, 1999.

2. G. Amdahl, *Validity of the single-processor approach to achieving large-scale computational capabilities*, proceedings of the AFIPS Conference, volume 30, page 483, AFIPS Press, 1967.

3. W. Schönauer, *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers*, self-edition, Karlsruhe, Germany, 2000.

4. D. O'Neal and J. Urbanic, *On Microprocessors, Memory Hierarchies, and Amdahl's Law*, proceedings of the DoD High Performance Computing Users Group Conference, Monterrey, CA, 1999.

5. *MPI: A Message Passing Interface Standard*, University of Tennessee, Knoxville, TN, May 5, 1994.

6. *OpenMP Fortran Application Program Interface*, Version 1.0, October, 1997.

7. P. Leggett, S. Johnson, and M. Cross, *CAPLib: A Thin Layer Message Passing Library to Support Computational Mechanics Codes on Distributed Memory Systems*, internal report, Parallel Processing Research Group, Center for Numerical Modelling and Process Analysis, University of Greenwich, London, UK, 2000.

8. D. O'Neal and R. Reddy, *The Parallel Finite Element Method*, in proceedings of the Cray User Group Inc., Spring Conference, Denver, CO, 1995.

9. *Computer Aided Parallelization Tools User's Guide*, Parallel Processing Research Group, University of Greenwich, London, UK, Version 2.0 Beta, October, 1998.