Checkpointing Aided Parallel Execution Model and Analysis

Laura Mereuta and Éric Renault

GET / INT — Samovar UMR INT-CNRS 5157
9 rue Charles Fourier, 91011 Évry, France
Tel.: +33 1 60 76 45 56, Fax: +33 1 60 76 47 80
{laura.mereuta,eric.renault}@int-evry.fr

Abstract. Checkpointing techniques are usually used to secure the execution of sequential and parallel programs. However, they can also be used in order to generate automatically a parallel code from a sequential program, these techniques permitted to any program being executed on any kind of ditributed parallel system. This article presents an analysis and a modelisation of an innovative technique — CAPE — which stands for Checkpointing Aided Parallel Execution. The presented model provides some hints to determine the optimal number of processors to run such a parallel code on a distributed system.

1 Introduction

Radical changes in the way of taking up parallel computing has operated during the past years, with the introduction of cluster computing [1], grid computing [2], peer-to-peer computing [3]... However, if platforms have evolved, development tools remain the same. As an example, HPF [4], PVM [5], MPI [6] and more recently OpenMP [7] have been the main tools to specify parallelism in programs (especially when supercomputers were the main issue for parallel computing), and they are still used in programs for cluster and grid architectures. Somehow, this shows that these tools are generic enough to follow the evolution of parallel computing. However, developers are still required to specify almost every information on when and how parallel primitives (for example sending and receiving messages) shall be executed.

Many works [8,9] have been done in order to automatically extract parallel opportunities from sequential programs in order to avoid developers from having to deal with a specific parallel library, but most methods have difficulties to identify these parallel opportunities outside nested loops. Recent research in this field [10,11], based on pattern-maching techniques, allows to substitute part of a sequential program by an equivalent parallel subprogram. However, this promising technique must be associated an as-large-as-possible database of sequential algorithm models and the parallel implementation for any target architectures for each of them.

At the same time, the number of problems that can be solved using parallel machines is getting larger everyday, and applications which require weeks (or

R. Perrott et al. (Eds.): HPCC 2007, LNCS 4782, pp. 707–717, 2007.
© Springer-Verlag Berlin Heidelberg 2007

months, or even more...) calculation time are more and more common. Thus, checkpointing techniques [12,13,14] have been developed to generate snapshots of applications in order to be able to resume the execution from these snapshots in case of problem instead of restarting the execution from the beginning. Solutions have been developed to resume the execution from a checkpoint on the same machine or on a remote machine, or to migrate a program in execution from one machine to another, this program being composed of a single process or a set of processes executing in parallel.

Through our researches, we try to address a different problem. Instead of securing a parallel application using checkpointing techniques, checkpointing techniques are used to introduce parallel computing inside sequential programs, i.e. to allow the parallel execution of parts of a program for which it is known these parts can be executed concurrently. This technique is called CAPE which stands for Checkpointing Aided Parallel Execution. It is important to note that CAPE does not detect if parts of a program can be executed in parallel. We consider it is the job of the developer (or another piece of software) to indicate what can be executed in parallel. CAPE consists in transforming an original sequential program into a parallel program to be executed on a distributed parallel system. As OpenMP already provides a set of compilation directives to specify parallel opportunities, we decided to use the same in order to avoid users from learning yet another API. As a result, our method provides a distributed implementation of OpenMP in a very simple manner. As a result, CAPE is a good alternative to provide a distributed implementation of OpenMP in a very simple manner.

Moreover, CAPE provides three main advantages. First, there is no need to learn yet another parallel programming environment or methodology as the specification of parallel opportunities in sequential programs is performed using OpenMP directives. Second, CAPE inherently introduces safety in the execution of programs as tools for checkpointing are used to run concurrent parts of programs in parallel. Third, more than one node is used only when necessary, i.e. when a part of the program requires only one node to execute (for example if this part is intrinsincly sequential), only one node is used for execution.

Other works have presented solutions to provide a distributed implementation of OpenMP [15]. Considering that OpenMP has been designed for shared-memory parallel architectures, the first solution was consisting in executing OpenMP programs on top of a distributed shared memory based machine [16]. More recently, other solutions have emerged, all aiming at transforming OpenMP code to other parallel libraries, like Global Arrays [17] or MPI [18].

This article aims at presenting the model we developed to analyze the behaviour of CAPE implementations. Using this model, we determine the optimal number of nodes one should use to minimize the execution time of a paralle program based on CAPE. The article is organized as follows. First, we present CAPE, our method to make a parallel program from a sequential program and demonstrate that if Bernstein's conditions are satisfied then the execution of a program using CAPE is equivalent to the sequential execution of the same program. Then, we introduce a model to describe the different steps involved

in the execution of a program with CAPE; from the temporal equation derived from the model, we determine the optimal number of processors one should use to minimize the execution time of a given program.

2 CAPE

CAPE, which stands for Checkpointing Aided Parallel Execution, consists in modifying a sequential program (for which parts are recognized as being executable in parallel) so that instead of executing each part the one after the other one on a single machine, parts are automatically spread over a set of machines to be executed in parallel. In the following, we will just present the principle of CAPE to make the rest of the article readable. A deeper presentation is available in [19,20], which includes a proof of the concept.

As lots of works are on the way for distributing a set of processes over a range of machines, in the following we consider that another application (like Globus [21], Condor [22] or XtremWeb [23]) is available to start processes on remote nodes and get their results from there. In this section, this application is called the "dispatcher". CAPE is based on a set of six primitives:

– create (filename) stores in file "filename" the image of the current process. There are two ways to return from this function: the first one is after the creation of file "filename" with the image of the calling process; the second one is after resuming the execution from the image stored in file "filename". Somehow, this function is similar to the fork ( ) system call. The calling process is similar to the parent process with fork ( ) and the process resuming the execution from the image is similar to the child process with fork ( ). Note that it is possible to resume the execution of the image more than once; there is no such equivalence with the fork ( ) system call. The value returned by this function has a similar meaning as those of the fork ( ) system call. In case of error, the function returns -1. In case of success, the returned value is 0 if the current execution is the result of resuming its execution from the image and a strictly positive value in the other case (unlike the fork ( ) system call, this value is not the PID of the process resuming the execution from the image stored in the file).
– diff (first, second, delta) stores in file "delta" the list of modifications to perform on file "first" in order to obtain file "second".
– merge (base, delta) applies on file "base" the list of modifications from file "delta".
– restart (filename) resumes the execution of the process which image was previously stored in "filename". Note that, in case of success, this function never returns.
– copy (source, target) copies the content of file "source" to file "target".
– wait_for (filename) waits for any merges required to update file "filename" to complete.

The following presents two cases where CAPE can be applied automatically with OpenMP: the first case is the “parallel sections” construct and the second case is the “parallel for” construct.

In order to make the piece of code in Fig. 1 and Fig. 2 readable, operations for both the modified user program and the dispatcher have been included, and operations executed by the dispatcher are *emphasized*. Operations performed by the dispatcher are initiated by the modified user program while sending a message to the dispatcher. Operations are then performed as soons as possible but asynchronously regarding the modified user program.

Error cases (especially when saving the image of the current process) are not represented in Fig. 1(b). In some cases, a test is performed just after the creation of a new image in order to make sure it is not possible to resume the execution from the image direcly. This kind of image is used to evaluate variables updated by a specific piece of code and its execution is not intended to be effectively resumed.

2.1 The “Parallel for” Construct

In the case of for loops (see Fig. 1(a) and Fig. 1(b)), the first step consists in creating an “original” image for both having a reference for the “final” image

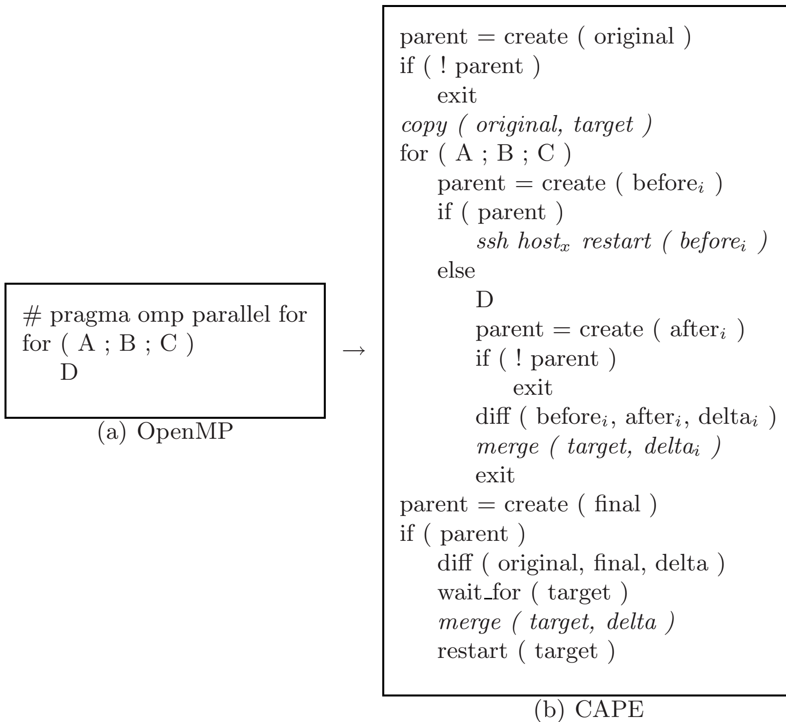

Fig. 1. Templates for “Parallel for”

and having a basis for the "target" image which is used at the end to resume the execution of the program sequentially after having executed the whole loop in parallel. Thus, once created, the "original" image is copied to a "target" image which will be updated step by step using deltas after the execution of each loop iteration.

The second step consists in executing loop iterations. After executing instruction A and checking condition B is true, a "before" image is created. When returning from function create (before), two cases are possible: either the function returns after creating the image, or the function returns after resuming from the image. In the first case, the dispatcher is asked to restart the image on a remote part and then the execution is resumed on the next loop iteration, i.e. executing C, evaluating B... In the second case, the loop iteration is executed and another image is saved in order to determine what are the modifications involved by the execution of this loop iteration (these modifications are then merged asynchronously to the "target" image by the dispatcher). Once all loop iterations have been executed, the third step consists in creating the "final" image which difference from the "original" image is used in order to set in image "target" modifications involved when executing C and evaluating B for the last time. Moreover, it ensures that the current frame in the execution stack is correct. This is the reason why it is necessary to wait for all other merges to be performed before including the one from "final". When restarting image "target", the execution resumes after create (final). As the process resumes its execution from an image, the last four lines in Fig. 1(b) are not executed the second time.

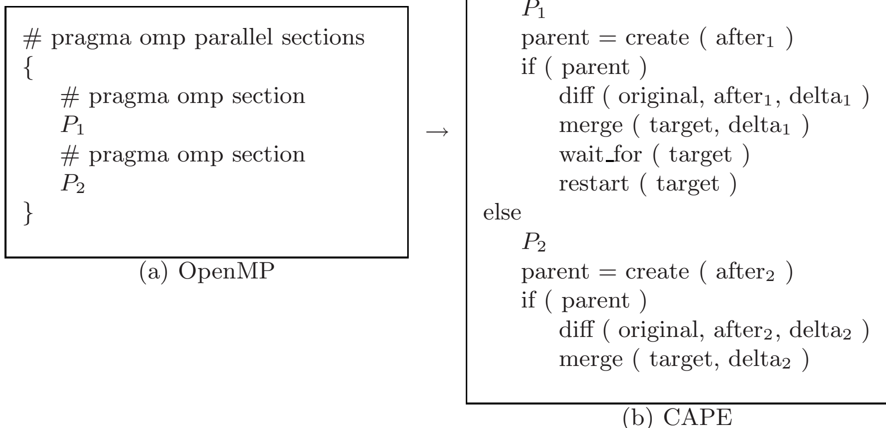

Fig. 2. Templates for "Parallel sections"

2.2 The "Parallel Sections" Construct

Let P_1 and P_2 be two parts of a sequential program that can be executed concurrently. Fig. 2(a) presents the typical code one should write when using OpenMP and Fig. 2(b) presents the code to substitute to run P_1 and P_2 in parallel with CAPE.

The first step consists in creating an "original" image used to resume the execution on a distant node, calculate the delta for each part executed in parallel and build the "target" image to resume the sequential execution at the end.

The second step consists in executing parts and generating deltas. Thus, the local node asks the dispatcher to resume the execution of the "original" image on a distant node. Parts are executed, two "after" images are generated to produce two "delta" files; then, these "delta" files are merged to the "target" image; all these operations are executed concurrently. The main difficulty here is to make sure that both the current frame in the execution stack and the set of processor registers are consistent. However, this can be easily achieved using a good checkpointer.

The last step consists in making sure all "delta" files have been included in the "target" image and then restarting the "target" image in the original process.

3 Parallel Model

This section presents an exact analysis of the model of our distributed implementation of OpenMP using CAPE. We are interested in the optimal number of processors for which the processing time is minimal. The model presented below has been done on modelling the temporal equations for each state of the model under various assumptions regarding the master-slave paradigm.

In our model, there are two kinds of processors. The master is in charge of distributing and collecting the data and the slaves are devoted to process the data. By convention, the master node is allocated process #0 and the slaves are allocated process number 1 to P. As a result, the total number of processes is $P + 1$.

Let $\Delta^{'}$ (resp. $\Delta^{''}$) be the amount of memory sent to (resp. received from) the slaves. In order to simplify the model, it is considered that these amounts of data are the same for all slaves. This is effectively the case for $\Delta^{'}$ as the image of the original process do not change a lot from one loop iteration to another; we also consider it is the case for $\Delta^{''}$ considering data and processes are equitably shared among slaves processes.

In the model, let C be the time to create a checkpoint, S be the time to send a message, N be the network latency (N is inversely proportional to the throughput) and I be the time to receive a message to update the local image of the process. Finally, let t_1 be the sequential execution time and t_P be the execution time for each of the P processors. Considering that the amount of work is equitably shared among processors, $t_P = \frac{t_1}{P}$.

Fig. 3 presents the different steps involve in the processing of the work through several processes and the relationship between the master and the slaves. Let

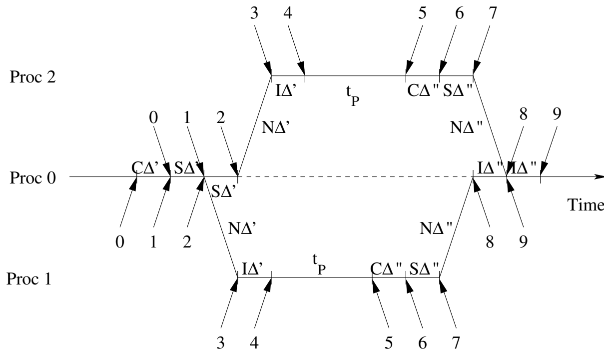

Fig. 3. The parallel model

$\Pi_{p,e}$ be the time when processor p ($p \in [0, P]$) reaches state e. As a result, with our model, we want to find the optimal value for P that makes $\Pi_{P,9}$ minimum.

From the analysis of the behaviour of our implementation of OpenMP for distributed memory architecture with CAPE and in accordance to Fig. 3, we can define $\forall p \in [0, P]$ the temporal equations (1) to (10).

$$\Pi_{p,0} = 0 \tag{1}$$
$$\Pi_{p,1} = \Pi_{p,0} + C\Delta^{'} \tag{2}$$
$$\Pi_{p,2} = \Pi_{p-1,2} + S\Delta^{'} \tag{3}$$
$$\Pi_{p,3} = \Pi_{p,2} + N\Delta^{'} \tag{4}$$
$$\Pi_{p,4} = \Pi_{p,3} + I\Delta^{'} \tag{5}$$
$$\Pi_{p,5} = \Pi_{p,4} + t_P \tag{6}$$
$$\Pi_{p,6} = \Pi_{p,5} + C\Delta^{''} \tag{7}$$
$$\Pi_{p,7} = \Pi_{p,6} + S\Delta^{''} \tag{8}$$
$$\Pi_{p,8} = \max(\Pi_{p,7} + N\Delta^{''}, \Pi_{p-1,9}) \tag{9}$$
$$\Pi_{p,9} = \Pi_{p,8} + I\Delta^{''} \tag{10}$$

Equations (3) and (9) can be normalized with: $\Pi_{0,2} = \Pi_{1,1}$ and $\Pi_{0,9} = \Pi_{P,2}$. In fact, both relations are due to the fact that all messages from the master must have been sent to the slaves before the master can receive the returning data from the slaves.

The set of equations from (1) to (10) can be reduced to the following ones:

$$\Pi_{p,2} = (C + pS + N)\Delta^{'}$$
$$\Pi_{p,7} = \Pi_{p,2} + (N + I)\Delta^{'} + t_P + (C + S)\Delta^{''}$$
$$\Pi_{p,9} = \max(\Pi_{p,7} + N\Delta^{''}, \Pi_{p-1,9}) + I\Delta^{''}.$$

4 Program Execution Time

The time needed to perform the execution of a program with CAPE is given by $\Pi_{P,9}$, ie. it is required the last processor has finished its work and returned it to the master node. In the following, we provide an expression for $\Pi_{P,9}$ with no recursion and that do not depend upon the value of p.

First, we need to determine the value for $\Pi_{P,8}$. From (9) and (10), we can see that the value for $\Pi_{P,8}$ can be expressed using the following recursion:

$$\Pi_{P,8} = \max \begin{cases} \Pi_{P,7} + N\Delta^{''} \\ \Pi_{P-1,8} + I\Delta^{''} \end{cases} \tag{11}$$

After replacing the value for $\Pi_{P-1,8}$ by the corresponding expression provided for $\Pi_{P,8}$ in (11), we obtain (12) for $\Pi_{P,8}$.

$$\Pi_{P,8} = \max_{i=0}^{P-2} \begin{cases} \Pi_{P-i,7} + (N + iI)\Delta^{''} \\ \Pi_{1,8} + (P-1)I\Delta^{''} \end{cases} \tag{12}$$

Then, $\Pi_{1,8}$ can be substituated its value using (9). After reduction, we have the following expression for $\Pi_{1,8}$:

$$\Pi_{P,8} = \max_{i=0}^{P-1} \begin{cases} \Pi_{P-i,7} + (N + iI)\Delta^{''} \\ \Pi_{P,2} + (P-1)I\Delta^{''} \end{cases}$$

After performing a variable change $(P - i \rightarrow i)$, $\Pi_{P-i,7}$ and $\Pi_{P,2}$ are substituated by their respective values. From the new expression for $\Pi_{1,8}$, $\Pi_{1,9}$ can be provided $\forall p \in [0, P]$ using (10):

$$\Pi_{P,9} = \max \begin{cases} (C + pS + N + I)\Delta^{'} + \frac{t_1}{P} + (C + S + N + (P - p + 1)I)\Delta^{''} \\ (C + PS)\Delta^{'} + PI\Delta^{''} \end{cases} \tag{13}$$

The set of expressions that depends upon the value of p in (13) can be reduced to two equations only. In fact, while taking p as a variable, this set of equations can be written:

$$(S\Delta^{'} - I\Delta^{''})p + (C + N + I)\Delta^{'} + \frac{t_1}{P} + (C + S + N + (P + 1)I)\Delta^{''} \tag{14}$$

This is the equation of an affine function which maximum depends on the coefficient of p. As a matter of fact, we have two different interpretations for expression (14) according to the sign of $S\Delta^{'} - I\Delta^{''}$. If $S\Delta^{'} < I\Delta^{''}$, expression (14) is maximized when $p = 0$; if $S\Delta^{'} > I\Delta^{''}$, expression (14) is maximized when $p = P$.

As result, (15) for $\Pi_{P,9}$ does not depend upon the value of p and is not recursive.

$$\Pi_{P,9} = \max \begin{cases} \max(S\Delta^{'}, I\Delta^{''})P + \frac{t_1}{P} + (C + N + I)(\Delta^{'} + \Delta^{''}) + S\Delta^{''} \\ (S\Delta^{'} + I\Delta^{''})P + C\Delta^{'} \end{cases} \tag{15}$$

5 Optimal Number of Nodes

In the previous sections, we have defined a model to evaluate the performance of the execution of a parallel program based on the framework developed with CAPE and we have determine the time needed to perform the execution of a program. In this section, we use the model defined above to determine the optimal number of processors that one should use in the parallel machine in order to minimize the execution time of the parallel program.

Let P^* be the optimal number of processors that minimizes the execution time, ie. the number of processors that satisfies the following expression:

$$\Pi_{P^*,9} \leq \Pi_{P,9} \qquad \forall P$$

Let $f_1(P)$ and $f_2(P)$ be the first and the second expressions in (15) respectively, ie:

$$f_1(P) = \max(S\Delta^{'}, I\Delta^{''})P + \frac{t_1}{P} + (C + N + I)(\Delta^{'} + \Delta^{''}) + S\Delta^{''} \qquad (16)$$

$$f_2(P) = (S\Delta^{'} + I\Delta^{''})P + C\Delta^{'}$$

The derivation in P of f_1 leads to the following expression:

$$f_1'(P) = \max(S\Delta^{'}, I\Delta^{''}) - \frac{t_1}{P^2} \qquad (17)$$

One can demonstrate that there are only two zeros for this expression. Moreover, only one of both resides in $\mathcal{R}^+$. This means that there is just one inflection point in $\mathcal{R}^+$ for f_1. As f_2 is an affine function and as the coefficient for P in this expression is bigger than the one in f_1, there is one and only one intersection point between f_1 and f_2. As a result, the maximum for (15) is provided by f_1 before the intersection of both expressions and is provided by f_2 in the other case. In a more formal way:

$$P^* = \min \begin{cases} f_1 = f_2 \\ f_1' = 0 \end{cases}$$

In order to determine the intersection of both expressions, we compute $f_1(P) - f_2(P) = 0$ which leads to expression (18).

$$\min(S\Delta^{'}, I\Delta^{''})P^2 - \left[(C + S)\Delta^{''} + (N + I)(\Delta^{'} + \Delta^{''})\right] P - t_1 = 0 \qquad (18)$$

Note that the maximum in (16) has been transformed into a minimum in (18) using the following relation:

$$x + y = \min(x, y) + \max(x, y) \qquad \forall x, y$$

Equation (18) is a second degree equation in P. Considering that C, S, N, I, $\Delta^{'}$, $\Delta^{''}$ and t_1 are all positive constants, the associated discrimant is necessarily strictly positive and there is one and only one solution in $\mathcal{R}^+$.

In order to determine the value of P^* in the second case, we have to determine the zero of $f_1'(P) = 0$. From (17), $f_1'(P) = 0$ can be transformed into a second degree equation of the form $ax^2 + b = 0$ which provides a single obvious solution in $\mathcal{R}^+$.

6 Conclusion

This article is based on an original way of transforming a sequential program into a parallel program, CAPE for Checkpointing Aided Parallel Execution, that uses checkpointing techniques rather than shared memory or any message-passing library. Then, we introduced a model to evaluate the performane of the distributed execution. In order to do so, we defined the parameters of the model and provided the temporal equations. Finally, we determined the optimal number of processors that should be used to minimize the execution time.

In the current model, we considered that the time spent by each slave processor is the same on all slaves. In the future, we will consider that these time may vary a little bit from one process to another.

References

1. Buyya, R.: High Performance Cluster Computing: Architectures and Systems, vol. 1. Prentice-Hall, Englewood Cliffs (1999)
2. Foster, I., Kesselman, C., Tuecke, S.: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. The International Journal of High Performance Computing Applications 15(3), 200–222 (2001)
3. Leuf, B.: Peer to Peer. In: Collaboration and Sharing over the Internet, Addison-Wesley, Reading (2002)
4. Loveman, D.B.: High Performance Fortran. IEEE Parallel & Distributed Technology: Systems & Applications 1(1), 25–42 (1993)
5. Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Network Parallel Computing. Scientific and Engineering Computation Series. MIT Press, Cambridge (1994)
6. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference, 2nd edn. Scientific and Engineering Computation Series. MIT Press, Cambridge (1998)
7. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 2.5 Public Draft (2004)
8. Allen, J.R., Callahan, D., Kennedy, K.: Automatic Decomposition of Scientific Programs for Parallel Execution. In: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Munich, West Germany, pp. 63–76. ACM Press, New York (1987)
9. Feautrier, P.: Automatic parallelization in the polytope model. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 79–103. Springer, Heidelberg (1996)
10. Barthou, D., Feautrier, P., Redon, X.: On the Equivalence of Two Systems of Affine Recurrence Equations. In: Monien, B., Feldmann, R.L. (eds.) Euro-Par 2002. LNCS, vol. 2400, pp. 309–313. Springer, Heidelberg (2002)
11. Alias, C., Barthou, D.: On the Recognition of Algorithm Templates. In: Knoop, J., Zimmermann, W. (eds.) Proceedings of the 2nd International Workshop on Compiler Optimization meets Compiler Verification, Warsaw, Poland, pp. 51–65 (2003)
12. Web page: Ckpt (2005), http://www.cs.wisc.edu/~zandy/ckpt/

13. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments. In: Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation, Boston, MA, pp. 361–376 (2002)
14. Plank, J.S.: An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee (1997)
15. Merlin, J.: Distributed OpenMP: extensions to OpenMP for SMP clusters. In: EWOMP 2000. 2nd European Workshop on OpenMP, Edinburgh, UK (2000)
16. Karlsson, S., Lee, S.W., Brorsson, M., Sartaj, S., Prasanna, V.K., Uday, S.: A fully compliant OpenMP implementation on software distributed shared memory. In: Sahni, S.K., Prasanna, V.K., Shukla, U. (eds.) HiPC 2002. LNCS, vol. 2552, pp. 195–206. Springer, Heidelberg (2002)
17. Huang, L., Chapman, B., Liu, Z.: Towards a more efficient implementation of OpenMP for clusters via translation to global arrays. Parallel Computing 31(10–12), 1114–1139 (2005)
18. Basumallik, A., Eigenmann, R.: Towards automatic translation of OpenMP to MPI. In: Proceedings of the 19th annual international conference on Supercomputing, pp. 189–198. ACM Press, Cambridge, MA (2005)
19. Renault, E.: Parallel Execution of For Loops using Checkpointing Techniques. In: Skeie, T., Yang, C.S. (eds.) Proceedings of the 2005 International Conference on Parallel Processing Workshops, Oslo, Norway, pp. 313–319. IEEE Computer Society Press, Los Alamitos (2005)
20. Renault, E.: Automatic Parallelization made easy using Checkpointing Techniques. In: Oudshoorn, M., Rajasekaran, S. (eds.) Proceedings of the 18th International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV (2005)
21. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. The International Journal of Supercomputer Applications and High Performance Computing 11(2), 115–128 (1997)
22. Litzkow, M., Livny, M., Mutka, M.: Condor - A Hunter of Idle Workstations. In: The 8th International Conference on Distributed Computing Systems, San Jose, CA, pp. 104–111. IEEE Computer Society Press, Los Alamitos (1988)
23. Fedak, G., Germain, C., Néri, V., Cappello, F.: XtremWeb: A Generic Global Computing System. In: Buyya, R., Mohay, G., Roe, P. (eds.) Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid, Brisbane, Australia, pp. 582–587. IEEE Computer Society Press, Los Alamitos (2001)