NASA Technical Memorandum 87349

Increasing Processor Utilization During Parallel Computation Rundown

William H. Jones
Lewis Research Center
Cleveland, Ohio

Prepared for the
1986 International Conference on Parallel Processing
sponsored by the Institute of Electrical and Electronics Engineers, Inc.
St. Charles, Illinois, August 19–22, 1986

NASA

86N26914*# ISSUE 17 PAGE 2789 CATEGORY 59 RPT#: NASA-TM-87349
E-3101 NAS 1.15:87349 86/00/00 13 PAGES UNCLASSIFIED DOCUMENT
UTTL: Increasing processor utilization during parallel computation rundown
AUTH: A/JONES, W. H.
CORP: National Aeronautics and Space Administration. Lewis Research Center, Cleveland, Ohio. AVAIL.NTIS
SAP: HC A02/MF A01
CIO: UNITED STATES Presented at the 1986 International Conference on Parallel Processing, St. Charles, Ill., 19-22 Aug. 1986; sponsored by IEEE
MAJS: /*COMPUTATION/*PARALLEL PROCESSING (COMPUTERS)/*RUN TIME (COMPUTERS)
MINS: / ALGORITHMS/ NAVIER-STOKES EQUATION/ PROGRAMMING LANGUAGES/ SCHEDULING
ABA: Author
ABS: Some parallel processing environments provide for asynchronous execution and completion of general purpose parallel computations from a single computational phase. When all the computations from such a phase are complete, a new parallel computational phase is begun. Depending upon the granularity of the parallel computations to be performed, there may be a shortage of available work as a particular computational phase draws to a close (computational rundown). This can result in the waste of computing resources and the delay of the overall problem. In many practical instances, strict sequential ordering of phases of parallel computation is not totally required. In such cases, the beginning of one phase can be

INCREASING PROCESSOR UTILIZATION DURING PARALLEL COMPUTATION RUNDOWN

William H. Jones
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

E-3101

SUMMARY

Some parallel processing environments provide for asynchronous execution and completion of general purpose parallel computations from a single computational phase. When all the computations from such a phase are complete, a new parallel computational phase is begun. Depending upon the granularity of the parallel computations to be performed, there may be a shortage of available work as a particular computational phase draws to a close (computational rundown). This can result in the waste of computing resources and the delay of the overall problem.

In many practical instances, strict sequential ordering of phases of parallel computation is not totally required. In such cases, the "beginning" of one phase can be correctly computed before the "end" of a previous phase is completed. This allows additional work to be generated somewhat earlier to keep computing resources busy during each computational rundown. This paper identifies the conditions under which this can occur, reports the frequency of occurrence of such overlapping in an actual parallel Navier-Stokes code, suggests a language construct, and discusses possible control strategies for the management of such computational phase overlapping.

INTRODUCTION

General purpose parallel computations are usually divided into phases that must execute sequentially in order to guarantee algorithmic integrity. For instance, the checkerboard approach to the successive over-relaxation solution of the potential field problem divides into two such phases: the "odd" locations phase and the "even" locations phase. On the parallel phase level, the iterated values of the previous phase must be complete before the new values of the next phase can be correctly computed.

In the checkerboard algorithm, the execution time of each location is definite (nominally, the time for four additions and a divide). Thus, the distribution of work among processors can be accurately planned. Under ideal conditions (involving the number of checkerboard locations in comparison to the number of processors), the distribution of work can be arranged so that each processor shares an exactly even portion of the work and, as a consequence, each processor completes its work at exactly the same time. Perfect computation resource utilization is realized (at least in a practical sense) since the next computational phase can begin immediately.

Unfortunately, ideal conditions are infrequently found in real applications. Continuing with the checkerboard algorithm, consider the situation when the potential grid is 1024 points on a side (2**20 grid points) and 1000 processors are available. Each computational phase will provide 524, 288 individual computations, or 524 computations for each of the 1000 processors;

however, 288 computations will be left over for distribution among the 1000 processors. This will leave 712 processors with nothing to do while the final 288 computations are carried out.

The burden of experience gained by the author suggests that even this example is optimistic. Most computations carried out by the author's parallel Navier-Stokes solver (the Combined Aerodynamic and Structural Dynamic Problem Emulating Routines or CASPER (ref. 1) which was controlled by the Parallel, Asynchronous Executive or PAX (ref. 2)) could not even be ascribed with definite execution times. In some instances, whether or not the computation was even to be carried out in a particular instance was a conditional part of the algorithm. No control over the computation-count-to-processor ratio was attempted -- processors were allocated as they became available on a the-more-the-merrier basis. Also, shared information access times were unpredictable and unrepeatable from instance to instance. As a result, there was no assurance that individual processors could be kept busy as a particular computational phase drew to a close.

The PAX/CASPER project provided the experience base cited later in this paper. PAX/CASPER was focused on a parallel, general purpose, Navier-Stokes solver. Thus, this experience base is presented not as a grand generalization for all of parallel processing, but as a specific example in practical parallel processing.

Certain other situations that might seem of interest in the overlapping of computational phases (for instance, the possibilities for overlapping in a tight iterative loop) are not treated for the simple reason that they did not occur in the PAX/CASPER project. PAX/CASPER was not so much a research project in parallel processing as an exploratory development of a far-term aerodynamic tool. Thus, the motivation was to solve the problems that occurred rather than to solve the problems that one could imagine.

It has been suggested that scheduling and overhead problems will be a particular problem in PAX/CASPER. So far, this has not been the case. Operational experience shows that the ratio of computation to management has been running at something in the neighborhood of 200. This paper is an effort to chart a method of improving upon this situation so as to stave off any backsliding that might occur as the ratio of computational to management resources increases. There are additional strategies which have been identified for development. These include a middle management scheme to parallelize the serial management function, a direct worker-to-worker lateral communication scheme, and a data-proximity work assignment algorithm. These strategies combined with the overlapping of computational phases should enhance the management overhead situation.

Various solutions to the computational rundown problem may be acceptable. Some parallel processing schemes for general purpose computation may choose simply to accept the lower processor utilization as a minor design flaw. Another alternative is to create a multi-parallel-job-stream environment that allows computational work of one job stream to fill in when another job stream enters a computational rundown situation. This will bring processor utilization up; however, it should be recognized that the primary goal of parallel processing is to reduce elapsed wall-clock time for a given job. The introduction of such a "batch" environment will inevitably distribute processor

resources among the several job streams and, thus, reduce the total processing power on any particular job and lengthen its elapsed wall-clock time.

Overlapping Computational Phases

The goal then is to find more ready-to-compute work from the parallel algorithm that is being computed. As mentioned previously, this is not possible at the parallel phase level: each phase must be completed before the next is begun in order to guarantee algorithmic integrity; however, if an examination is made at a deeper (sub-phase or, in the terminology of the author, task) level, it is frequently discovered that the completion of portions (tasks) of one phase will allow the correct computation of portions (tasks) of the succeeding phase.

Consider again the checkerboard algorithm. If all the "odd" locations adjacent to a particular "even" location have been updated with new values from the current computational phase, then the new value for that particular "even" location for the next computational phase can be correctly computed. Additionally, since all the computations requiring as an input the current value of that particular "even" location have been completed, the value for that "even" location can be updated without affecting the results of the current computational phase.

At this point, it is necessary to make certain assumptions (or, alternatively, set certain system design constraints) about the nature of computational phase rundown. Two basic situations arise: one in which task assignments and releases are statically determined and one in which such matters are dynamically determined.

The static situation is much simpler from the standpoint of next-phase task release timing since everything is determined ahead of time. In this case, it can be acceptable for computational rundown to begin almost immediately since the scheduling of the next-phase task has already been statically determined. No completion processing of current-phase tasks is required to schedule the release of the next-phase task. (In fact, work in this area for the purposes of real-time simulation has been conducted for some years at NASA Lewis (refs. 3 and 4)).

The dynamic scheduling situation is substantially more interesting. Some time delay must be available between the completion of the first current-phase tasks and the onset of computational rundown. This delay is needed to provide time to process the completion of the early current-phase tasks and, in so doing, schedule the next-phase tasks that are thus enabled. During this delay, there must be enough current-phase tasks to keep the processing resources busy in order to avoid a computational load dip while the next-phase tasks are scheduled.

In the dynamic scheduling situation, enablement relationships between the current-phase tasks and the next-phase tasks (i.e., the relationship that enables a next-phase task based upon the completion of a current-phase task) may be either static or dynamic. That is, the completion of a particular current-phase task may always enable the same next-phase task (the static enablement case) or it may enable some next-phase task that can only be identified at the time of execution (the dynamic enablement case). The nature of

the enablement relationships is important because it is involved in setting the time delay from the completion of the first current-phase tasks to the availability of the first enabled next-phase tasks.

Considering these characteristics of the dynamic scheduling situation (i.e., the time to process current-phase task completion, the time to recognize enablement relationships, and the time to schedule enabled next-phase tasks), it can be observed that the number of tasks should substantially outnumber the number of processors. Certainly, there should be at the outset of the current-phase work at least two tasks for each processor so that at least one task execution time will be available to process the completion of the first task assigned to the processor and to schedule the enabled next-phase task. This presumes that completion processing and task scheduling time is small with respect to task execution time. In particular, it assumes that one such completion, enablement, and scheduling cycle for each of the processors in the system can be completed in a single task execution time. (The author's experience with PAX suggests that this is reasonable even for dynamic managerial style parallel processing systems. Systems that use hardware-level synchronization primitives presumably would be at even greater advantage in this area.)

The conditions under which this overlapping of computational phases can correctly occur are the same as those that allow parallel computations within a particular phase. Let the logical predicate PARALLEL(x,y) return the condition TRUE when x and y are such that parallel computations are allowed. Clearly, PARALLEL(n,m) must always be TRUE if n and m are distinct computational granules of the same parallel computational phase. Let q be an uncompleted granule of the current phase and r be a granule of the next phase that has been enabled by some completed granule, p, of the current phase. If PARALLEL(q,r) necessarily returns the value TRUE, then the current-phase and next-phase can be correctly overlapped.

The exact nature of the logical predicate PARALLEL(x,y) is, of course, of substantial practical interest; however, it has no direct impact upon the ability to overlap phases as outlined above. Different parallel systems may identify different logical predicates.

Identifying Enabled Granules

The first challenge to be met is to find a way of identifying enabled next-phase granules for overlapping. It is easy to postulate that some mapping function exists either to map from the set of completed granules, p, to the set of enabled granules, r, or to map from the set of uncompleted granules, q, to the set of enabled granules, r. It is very difficult to establish what this mapping function might be in any general way. Fortunately, this mapping function is much more easily identified when each concrete situation is faced.

First, consider the simplest imaginable case as represented by the following Fortran code segment:

```
      ...
      DO 100 I=1,N    ! First computational phase
      B(I)=A(I)       !
100   CONTINUE        !
      DO 200 I=1,N    ! Second computational phase
      D(I)=C(I)       !
200   CONTINUE        !
      ...
```

Assuming that there are not shared output area constraints, it can be observed that these two parallel computational phases can be computed in parallel with each other. This represents what might be called a universal mapping function wherein any granule of the second computational phase is enabled by any granule or set of granules (including the null set) of the first computational phase.

PAX/CASPER experience shows that 6 out of 22 (or 27 percent) of the parallel computational phases allow universal mapping enablement of the succeeding phases. This represents 266 out of 1188 lines (or 22 percent) of the code that is executed in parallel in PAX/CASPER.

This universal mapping usually occurs in PAX/CASPER when the nature of the larger computational process is changing. For instance, the change over from power of compression computations to interpolator matrix generation is one such character change. The two computations do not involve shared information of any kind and, thus, they can be entirely overlapped. Of course, the two phases could be merged into one by a preprocessor of the parallel control stream; however, since the mechanisms necessary to handle this case would be a subset of those needed for the following case, it might well be simpler to support this enablement mapping.

For the next case, consider the following Fortran fragment that is to be computed in parallel as two succeeding computational phases:

```
...
      DO 100 I=1,N    ! First computational phase
      B(I)=A(I)       !
100   CONTINUE        !
      DO 200 I=1,N    ! Second computational phase
      C(I)=B(I)       !
200   CONTINUE        !
      ...
```

Again assuming that there are not shared output area constraints, it can be observed by inspection that the identity mapping function (I = I) maps from completed granules, p, to enabled granules, r. This is also a simple and easily identified mapping.

PAX/CASPER experience indicates that it applies in 9 out of 22 (or 41 percent) of the parallel computational phases (representing 551 of 1188 code lines, or about 46 percent of the parallel code in PAX/CASPER). Combining this direct mapping with the simpler universal mapping above indicates that (at least in PAX/CASPER experience) 68 percent of the parallel computational phases and 68 percent of the code executed in parallel can be easily overlapped

to defeat computational rundown. These two enablement mapping possibilities are the most frequently occurring situations in PAX/CASPER experience.

The next most frequently occurring enablement mapping in PAX/CASPER experience is what could be called null mapping, that is, the situation in which no overlapping is possible. This occurs in 4 out of 22 (or 18 percent) of the computational phases and represents 262 out of 1188 (or 22 percent) of the lines of code executing in parallel. In all cases the cause was not that such an overlapping did not exist between the parallel computations but was, in fact, that serial actions and decisions had to occur between the phases. This is important since it allows one to assess how often the extra effort of supporting overlapping features will be entirely defeated, regardless of the sophistication of the overlapping phase support features.

Another enablement mapping occurring in PAX/CASPER experience is a reverse indirect mapping. Consider the following Fortran fragment:

```
      ...
      DO 10 I=1,N          ! Set up source mapping
      DO 10 J=1,10         !
      IMAP(J,I)=IRAND()    ! IRAND produces an integer
10    CONTINUE             !   in the range 1 to N
      DO 100 I=1,N         ! First computational
      A(I)=FUNC(I)         !  phase generates some
                           !  number in A(x)
100   CONTINUE             !
      DO 200 I=1,N         ! Second computational
      DO 200 J=1,10        !  phase sums subsets of
      B(I)=A(IMAP)(J,I))!  the results of the
                           !  first computational
                           !  phase
200   CONTINUE             !
      . . .                !
```

Clearly, this computation can be overlapped; however, determining the enablement mapping is very difficult. This is because knowing that a particular first phase granule is complete does not directly identify any distinct second phase granule as computable; however, a reverse mapping from desired second phase granule to required first phase granules is possible.

In PAX/CASPER experience, this situation occurs in 2 of 22 (or 9 percent) of the computational phases representing 78 out of 1188 (or 7 percent) of the lines of code executing in parallel. While this is not a frequently occurring situation in PAX/CASPER experience, it cannot be ignored out of hand. Some engineering judgement must be made to weigh the cost (in terms of management overhead, computational resource transferred from workers to management, etc.) of some reverse enablement mapping solution against the cost of computational rundown in 9 percent of the parallel computational phases.

Certainly, a solution exists for the reverse, indirect enablement mapping. Once the values of the information selection map (represented in the code fragment by the array IMAP) have been determined, it is a simple matter to produce a composite map of first phase granules that must be completed in order to enable a particular second phase granule. The executive can then use this map upon each first phase granule completion to determine the computability of

particular second phase granules. This map could also be used to direct a preferred order of first phase granule dispatching so as to enable a known second phase granule as early as possible.

Two important facts about this reverse enablement mapping must be included. First, both occurrences of this situation involved a dynamically generated information selection map. Thus, the composite granule map would have to be generated by the executive at or after first phase initiation but before any second phase enablements. Second, the impact of executive computation must be considered. In the PAX/CASPER UNIVAC 1100 test bed, executive computation was done at the direct expense of worker computation. Thus, extensive composite granule map generation could be self defeating. Some real parallel machines may provide separate executive computing resources, in which case the generation and use of composite granule maps would not be out of the question.

A final enablement form was observed in PAX/CASPER that could be characterized as a forward, indirect mapped situation. Consider the following Fortran fragment:

```
      DO 10 I=1,M              ! Generate forward
      IMAP(I)=IRAND()          !  map
10    CONTINUE                 !
      DO 100 I=1,M             ! Use forward map
                               !  to operate on a
      B(IMAP(I))=A(IMAP(I))    !  subset of the
100   CONTINUE                 !  arrays
      DO 200 I=1,N             ! Perform some further
      C(I)=B(I)                !  further opera-
200   CONTINUE                 !  tion on the
                                   complete arrays
      ...
```

This situation is somewhat easier than the reverse, indirect mapping in that the identification of a particular granule in the first phase can be directly mapped to an enabled granule in the successor phase; however, much of the complication of a mapped enablement remains. This form was the least frequently occurring situation in PAX/CASPER showing up only once (5 percent of the phases) and accounting for only 31 of 1188 lines of code executed in parallel.

No other forms of enablement mapping were observed in PAX/CASPER. Certainly, extensions of the forms already presented can be imagined. Additionally, a seam mapping problem (such as would be appropriate for the checkerboard approach to the successive over-relaxation problem) can be foreseen. These other forms are beyond the scope of the present paper.

Language Construction

The developing PAX/CASPER language is simple and requires the user to make specific statements concerning choices for the management of each parallel computational phase. Statements involving the enablement of a succeeding phase could be made at two times: during the definition of a computational phase to

the management system and during the invocation of the phase for actual computations. The difficulty to be faced is that the statements no longer apply solely to the phase being referenced, but rely also on the characteristics of the succeeding phase.

The simplest approach is to require the user to specify the appropriate enablement mapping method when the phase in invoked. It might appear as in the following PAX parallel language fragment:

```
...
DISPATCH  phase-name                              -
          ...                                     -
          ENABLE/MAPPING=option                   -
          ...
...
```

This is simple and explicit; however, it leaves the door wide open to user mistakes. There is no interlock between this phase and the next that can be verified by the executive. A simple solution to this would be to identify the name of the enabled next phase so that the executive system (or language processor) can verify that, in fact, that phase is following. This might appear as follows:

```
...
DISPATCH  phase-name                              -
          ...                                     -
          ENABLE [phase-name/MAPPING=option]      -
          ...
...
```

This allows the desired verification, but also brings up a new possibility. Occasionally, a conditional branch that is not dependent on the computational phase separates that phase from two or more succeeding phases, each of which may (or may not) be overlappable. If each of these phases were identified in the above construct, the executive could preprocess the branch and overlap the appropriate phase. This could look as follows:

```
    ...
    DISPATCH phase-name                           -
             ... -
             ENABLE/BRANCHINDEPENDENT             -
              [phase-name-1/MAPPING=option        -
               phase-name-2/MAPPING=option]       -
             ...
    IF       (IMOD(LOOPCOUNTER,10).NE.0)          -
    THEN     GO TO branch-target                  -
    DISPATCH phase-name-1                         -
             ...
    GO TO    rejoin
branch-target:
    DISPATCH phase-name-2                         -
rejoin:
    ...
```

Finally, the matching of mapping selections and phases and the invocation of the appropriate overlapping services is something that could be done when the parallel phase is defined to the system; however, it would still be necessary to identify preprocessable branches at the computation invocation site. This could appear as follows:

```
DEFINE   PHASE phase-name                          -
         ... -
         ENABLE [                                  -
                phase-name-1/MAPPING=option        -
                phase-name-2/MAPPING=option        -
                phase-name-3/MAPPING=option        -
                ]                                  -
...
DISPATCH phase-name                                -
         ... -
         ENABLE/BRANCHDEPENDENT                    -
...
```

The ENABLE/BRANCHINDEPENDENT would be deleted when branch preprocessing was either not appropriate or not needed. The executive system could perform the appropriate lookahead to see whether any of the named succeeding phases was actually following and apply, as appropriate, the specified enablement mapping.

Control Strategies

Control strategies for enabling and scheduling overlapped parallel computational phases are, of course, highly dependent upon the overall parallel processing strategies. As alluded to earlier, some approaches to parallel processing may do all of this before any computations are begun. Indeed, the entire process may be done manually by a human being when the pattern of parallel processing is fixed for the life of the system.

Within the PAX system, the opposite is true: the identification and scheduling of computable granules is entirely automatic. A scheduling mechanism for enabled computational granules already exists within the PAX system. It was developed to schedule dynamically created computations that conflicted (usually in terms of shared data access) with pre-existing computational granules.

Within PAX, each internal description of one (or more) computational granules included a queue head for a double circularly-linked list of computable but conflicting computational granules. Upon completion of the described computation, all the queued conflicting computations became unconditionally computable and were placed in the waiting computation queue. The waiting computation queue was kept in a known order and, for the purposes of the conflicting computation problem, it was determined that such conflicting computations would be placed ahead of the normal computations in the queue and, thus, given higher priority.

The scheduling of universally mapped successor phases within this system is very easy indeed. At the time of phase initiation, the successor phase is

also initiated and the resulting computation description placed in the waiting computation queue behind the current phase description.

The scheduling of directly enabled successor phases is similarly easy at first sight. At the time of phase initiation, the successor phase is also initiated and the resulting computation description placed in the conflicted computation queue of the current phase description. Thus, when the current phase computation is completed, the now-enabled successor computation will be placed in the waiting computation queue to be considered for scheduling.

The above approach for directly enabled successor phases is fine if each indivisible granule of computation is described separately. Unfortunately, this is usually not economical (in terms of storage space and task search times, among other things) and was not the choice taken in PAX design. Computations were, instead, described as large, contiguous collections of granules. The descriptions were split apart as necessary to produce conveniently sized tasks for workers and then merged back into single descriptions when the work was completed. This splitting of descriptions requires that queued computation descriptions also be split so that each queued description will accurately reflect the enablement relationship between the computation and its queued successor computation.

While this is certainly possible, it forces a further design decision for the executive software. PAX computation splitting was demanddriven by the presence of an idle worker. It was felt that the delay while splitting a task description was acceptable; however, the additional delays of splitting queued successor computation descriptions may represent an unacceptable situation. Two possible solutions exist. One possibility is to presplit the tasks before idle workers present themselves to the executive. This would allow the executive to work ahead in otherwise idle time. Alternatively, the splitting of a computation could generate a successor-splitting task that could be quickly queued for later attention when the executive would again be idle.

The successor computation description could be removed from the current computation description and included in the successor-splitting task information. When the successor-splitting task is executed the successor computation could be split and requeued to the appropriate current computation descriptions.

Management of indirectly (both forward and reverse) mapped successor computations is a good deal more interesting. The description of the successor computation cannot simply be queued to the description of the current computation since there is no guarantee of the enablement relationship. Additionally, it would seem wise to get the current phase into execution without the delay of constructing the necessary information for enabling successor computations. Both forward and reverse indirection would seem well handled by much the same mechanisms since the only significant difference is the direction of the indirection. Each leads naturally to a list of current phase granules that must be completed to enable a particular successor phase granule.

It would seem appropriate to identify a subset group of successor-phase granules that are to be the subject of the enablement operation so as to avoid solving an unnecessarily large enablement problem. Once this subset has been identified, the current-phase granules that enable the successor subset can be identified. Since these are not necessarily the current phase granules that would be naturally selected by the scheduling mechanism, they should be split

into individual descriptions and placed in the waiting computation queue in such a manner as to elevate their computational priority.

It is important to note that the description of the successor subset cannot simply be queued to any one of the identified current-phase granules since it is enabled not by the completion of any one such granule but by the completion of all the identified granules. This enablement on completion of all identified current-phase granules can be handled by any number of simple mechanisms. For instance, during completion processing, a status bit (set when the current-phase granules were identified and split into individual descriptions) can be checked and, if it is set, an enablement counter decremented. When the enablement counter reaches zero, it can be taken as a signal that the successor-phase granules are computable.

CONCLUDING REMARKS

This paper has discussed the possibilities for overlapping parallel computations in a general purpose parallel-computation environment so as to minimize loss of computational resources. Practical experience with PAX/CASPER, a parallel Navier-Stokes solver, suggests that simple and plausible steps could provide such overlapping in 68 percent of the computational phases and that, with extended effort, more than 90 percent of the computational phases are amenable to some form of phase overlapping.

REFERENCES

1. W.H. Jones, "Combined Aerodynamic and Structural Dynamic Problem Emulating Routines (CASPER): Theory and Implementation, NASA TP-2418, 1985.

2. W.J. Jones, "Parallel, Asynchronous Executive (PAX): System Concepts, Facilities, and Architecture," NASA TP-2179, 1983.

3. D.J. Arpasi, "Real-Time Multiprocessor Pro- gramming Language, (RTMPL) User's Manual," NASA TP-2422, 1985.

4. D.J. Arpasi and E.J. Milner, "Partitioning and Packing Mathematical Simulation Models for Calculation on Parallel Computers," NASA TM-87170, 1986.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-87349		

4. Title and Subtitle	5. Report Date
Increasing Processor Utilization During Parallel Computation Rundown	6. Performing Organization Code: 505-62-21
7. Author(s): William H. Jones	8. Performing Organization Report No.: E-3101
	10. Work Unit No.
9. Performing Organization Name and Address: National Aeronautics and Space Administration, Lewis Research Center, Cleveland, Ohio 44135	11. Contract or Grant No.
	13. Type of Report and Period Covered: Technical Memorandum
12. Sponsoring Agency Name and Address: National Aeronautics and Space Administration, Washington, D.C. 20546	14. Sponsoring Agency Code

15. Supplementary Notes

Prepared for the 1986 International Conference on Parallel Processing, sponsored by the Institute of Electrical and Electronics Engineers, Inc., St. Charles, Illinois, August 19-22, 1986.

16. Abstract

Some parallel processing environments provide for asynchronous execution and completion of general purpose parallel computations from a single computational phase. When all the computations from such a phase are complete, a new parallel computational phase is begun. Depending upon the granularity of the parallel computations to be performed, there may be a shortage of available work as a particular computational phase draws to a close (computational rundown). This can result in the waste of computing resources and the delay of the overall problem. In many practical instances, strict sequential ordering of phases of parallel computation is not totally required. In such cases, the "beginning" of one phase can be correctly computed before the "end" of a previous phase is completed. This allows additional work to be generated somewhat earlier to keep computing resources busy during each computational rundown. This paper identifies the conditions under which this can occur, reports the frequency of occurrence of such overlapping in an actual parallel Navier-Stokes code, suggests a language construct, and discusses possible control strategies for the management of such computational phase overlapping.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing; Parallel process rundown; Parallel process overlapping	Unclassified - unlimited STAR Category 59

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price*
Unclassified	Unclassified		

*For sale by the National Technical Information Service, Springfield, Virginia 22161

End of Document