NASA Technical Memorandum 106208

A Distributed Version of the NASA Engine Performance Program

Jeffrey T. Cours and Brian P. Curlett
Lewis Research Center
Cleveland, Ohio

July 1993

NASA

A Distributed Version of the NASA Engine Performance Program

Jeffrey T. Cours and Brian P. Curlett

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Abstract

Distributed NEPP, a version of the NASA Engine Performance Program, uses the original NEPP code but executes it in a distributed computer environment. Multiple workstations connected by a network increase the program's speed and, more importantly, the complexity of the cases it can handle in a reasonable time. Distributed NEPP uses the public domain software package, called *Parallel Virtual Machine*, allowing it to execute on clusters of machines containing many different architectures. It includes the capability to link with other computers, allowing them to process NEPP jobs in parallel. This paper discusses the design issues and granularity considerations that entered into programming Distributed NEPP and presents the results of timing runs.

1 Introduction

Distributed computing techniques, in which many computers simultaneously execute parts of a single, large program, could significantly increase performance and decrease computing time for many of the programs NASA engineers use. A distributed program executes in parallel on a collection of normal workstations connected by a network rather than on a dedicated parallel computer, meaning that the program gets many of the benefits of parallel computation but the user need not purchase and install a dedicated parallel machine.

The NASA Engine Performance Program, or NEPP, is one program that could benefit considerably from distributed computation [1]. NEPP is a one-dimensional, steady-state engine performance prediction program. The code is computationally intensive due to its flexible control schemes, its optimization of engine parameters and its chemical equilibrium calculation for thermodynamic properties. The code has a fairly simple overall structure and processes data in a relatively uniform, predictable manner, making it a good candidate for parallel conversion.

This paper describes the process and results to date of converting NEPP to execute in parallel on a collection of Unix workstations connected by an Ethernet network. Section 2 provides an introduction to the problem of parallelizing a program and the software tools this project used. Sections 3, 4, 5, and 6 describe Distributed NEPP's organization and inner workings and mention the specific issues that arose when converting NEPP to execute in a distributed environment. Section 7 provides some details on the provisions Distributed NEPP has that allow it to interface with other programs; specifically, the Integrated Propulsion/Airframe Analysis System, or IPAS [2]. Section 8 mentions some of the limitations of the current version of Distributed NEPP, while section 9 describes the results of the research, including a speed comparison between Sequential NEPP and Distributed NEPP. Finally, section 10 presents some concluding remarks.

2 Machine Architecture and Granularity Considerations

Different parallel computer systems require different approaches to parallelizing programs. A parallel computer may have a *shared memory*, in which all the processors can access the same memory, or it may have a *partitioned memory*, also known as a *distributed memory*, where each processor has its own memory and the processors communicate by explicitly sending messages back and forth. Furthermore, the amount of time it takes to start a *thread of execution*, a sequence of instructions that the computer executes in parallel with other threads, can vary significantly from one machine to another.

For many applications, shared memory architectures tend to simplify parallel programming. With a partitioned memory, the programmer has to divide the data the parallel program will use among the machines' processors and explicitly move information from the memory space of one parallel processor to others that require it. In effect, the programmer must write larger, more complex programs capable of managing data movement. In contrast, shared memory schemes allow all the processors to access all the pro-

gram information whenever they need it, leading to simpler parallel programs. However, for architectural reasons, shared memory CPUs often run more slowly than the CPUs in partitioned memory machines, and there are a number of open research questions concerning how to build shared memory parallel computers with large numbers of processing units.

The *cost* in terms of the amount of time it takes to start and stop threads of execution also has a tremendous impact on the parallel program's design. Systems that can cheaply start and stop threads tend to use *fine-grain parallelism*, in which small portions of the code, like the bodies of loops, execute in parallel. Systems in which the cost of starting and stopping threads is higher tend to use *coarse-grain parallelism*, increasing the size of the portions of code that execute in parallel, often to the point that each thread actually comprises an entire sequential program in itself, and the different programs execute in parallel.

Distributed NEPP uses a free, public domain package called Parallel Virtual Machine, or PVM, to send messages back and forth [1]. PVM makes it easy for programs executing on different Unix computers to communicate with one-another using the low-level network protocols that already exist in Unix. It takes care of translating data from one machine's native format to another, provides the ability to start processes remotely, and allows multiple processes on the same machine to interact.

Anyone can install PVM on a Unix machine: the installation process does not require root authority. PVM compiles on a broad variety of different architectures ranging from workstations to supercomputers. Additionally, some of the distributed queueing systems should support PVM in the near future.

To the programmer, distributed computing under PVM looks like using a partitioned-memory machine with extremely high thread initiation costs. Starting a new thread costs as much as starting a program in Unix: it could take as much as half a second. Parallel programs that execute under PVM have to use parallelism that is as coarse-grained as possible and they have to keep their threads, which are actually entirely separate programs, alive throughout almost the full parallel execution to show speed gains over their sequential counterparts.

The fact that the workstations communicate with each other over Ethernet tends to increase communications costs. Ethernet excels at sending long messages, such as transmitting entire files. Unfortunately, since the underlying software and hardware focuses on long messages, PVM appears to be relatively inefficient at sending shorter messages in the 10 kilobyte or smaller range. PVM requires about 3 milliseconds to set up the message for transmission and then a very low cost per byte to send it, so that the actual cost for a message tends to grow fairly slowly with the message length. On an Ethernet system, then, the program should pay most attention to minimizing the total number of messages, relegating the task of reducing message length to lower priority. However, other networking systems, as well as future revisions of the PVM software, should make shorter messages more efficient. Accordingly, Distributed NEPP tries to minimize both the number of messages and the message length wherever possible.

The concepts of efficiency and speedup prove useful when discussing the performance of a parallel program. Usually, researchers in parallel computing define the *speedup* of a parallel program as the execution time (wall clock time) of the fastest sequential version of the program divided by the execution time of the parallel version for the same task. According to this definition, a speedup of 2 means that the parallel program can finish the job twice as quickly as its sequential counterpart. *Efficiency*, then, is the speedup a parallel program achieves divided by the number of processors it required to achieve that speedup.

Because of the overhead involved in parallel processing, few parallel programs achieve 100% efficiency. A theoretical model developed by Gene Amdahl provides a good feel for what happens [3]. First of all, a portion of the execution time of every program has to happen sequentially. For example, in Distributed NEPP's case only one part of the parallel program, the host, reads the input files and writes information to the output file: none of the other parts of the program can begin execution until the host has read at least enough of the input file to send out the first parallel job. The time a program spends operating sequentially is called T_s. The remainder of the program can benefit from parallel execution, so it is called T_p. Now, if n processors work on the parallel job, the total execution time will be:

$$T = T_s + (T_p/n)$$

Where the sequential execution time always remains constant, but the parallel time varies with the number of processors.

Notice what happens in the equation above when n, the number of processors, goes to infinity: the time approaches the minimum possible time T_s. What this model claims is that, for a constant problem size, as the number of processors increases, execution time will approach a lower bound. Speedup, in turn, approaches an upper limit. Finally, the model indicates that as the number of processors increases, efficiency decreases, which is exactly what tends to happen in real parallel programs.

Another interesting fact to note about the model is that an excellent way to increase efficiency is to increase T_p, the amount of work the program can

[1] For more information on pvm send e-mail to Internet address pvm@msr.epm.ornl.gov.

do in parallel. As the problem size increases, the parallel program becomes more useful because it can process a larger problem faster than a sequential program could execute the same job. Not only does a parallel program speed up current jobs but it allows engineers to pursue larger, more complicated design that could not have been previously investigated due to time constraints.

3 Organization of Distributed NEPP

The distributed version of NEPP consists of several programs. One, the host, has the responsibility of assigning tasks to the many nodes. The nodes perform the actual computation. The host and one or more node programs can execute on the same or different machines, though a more typical arrangement would have the host and one node on one machine and one node on each of the other machines in the cluster. While PVM allows a wide variety of parallel program configurations to exist, the host/node configuration seemed to be the most natural for this application since, in a cluster without file sharing facilities, only one part of the parallel program would be likely to have access to the input files.

The host program reads the normal NEPP namelist input, organizes the information, and sends it to the nodes. The nodes, in turn, perform computation and return the results to the host which writes them to the output file. Distributed NEPP makes very few assumptions about the file structure of its environment. If the host and nodes are executing on a network that uses the Network File System (or some similar system that allows them to access the same files) Distributed NEPP will take advantage of that fact and the nodes will read some of the input data files themselves. However, if a node cannot access the file, it automatically requests a copy of the file's contents from the host and creates a local, temporary version of the file.

4 Job Allocation and Dynamic Load Balancing

A typical NEPP job consists of a design point calculation followed by a sequence of off-design cases. NEPP will size engine components to satisfy the design constraints. Once the program has sized the engine it executes a series of off-design cases to determine how the engine will perform under different flight conditions.

NEPP's grouping of its work into design and off-design cases provides a natural place to split the program into parallel execution streams. In Distributed NEPP, the host sends off-design parameters to the nodes, and the nodes calculate the off-design cases in parallel. The host distributes the cases on a first-come, first-served basis: as soon as a node finishes its current off-design case, it sends the results back to the host along with a request for another job to do at which point the host responds with the next off-design case. Slower nodes will take longer to finish their current jobs, so they will not ask for new jobs as often; as a result, the host gives more work to the faster nodes, while the slower nodes receive less work to perform.

Distributed NEPP's nodes can execute at low priority. In this case, node programs consume only machine cycles in which the machine would otherwise be idle; the dynamic load balancing scheme compensates for nodes on busy machines by sending work to other nodes on idle machines. As a result, Distributed NEPP tends to use CPU cycles on only idle machines without slowing down the busy ones.

5 Communications Protocol

Distributed NEPP uses a fairly sophisticated communications protocol. The communications module tries to keep the messages as short as possible, at the expense of using complex code where necessary. To start Distributed NEPP working, the user executes the host program **nepphost**. **nepphost**, in turn, figures out how many machines are available in the current cluster, and it starts that number of copies of the node program **neppnode**, one copy per machine. Once all the node programs have started, **nepphost** reads all the engine design parameters, packs them into a message, and broadcasts the message to all the nodes; the host then executes the design point locally. Meanwhile, the nodes unpack the design point information and check to see if they need a thermodynamic data file. (NEPP requires a thermodynamic data file for any engine description that uses the chemical equilibrium model.) If they need the thermodynamic data, they first try to open the file using a filename the host supplied when it sent the design parameters. Any nodes that cannot open the file send a message back to the host requesting that it send the contents of the file via a PVM connection.

As soon as the nodes have all the information they need, they also execute the NEPP design point. Having both the host and the nodes execute the design point proved to be the simplest way to initialize many of the internal variables within the original NEPP code.

Once each node has finished executing the design point, it sends a message to the host requesting an off-design case to execute. The host responds to the node's request by packing an off-design case into

a message and sending it to the requesting node. To save communications bandwidth, **nepphost** maintains a record of the most recent off-design case each node has executed, and when it sends the next job it sends only those parameters that have changed since the last one. This message compacting scheme reduces the size of the messages from potentially tens of kilobytes down to around 100 to 200 bytes.

When each node finishes executing its off-design case, it sends the results back to the host along with a request for a new off-design case to execute. **nepphost** continues to distribute off-design cases to the nodes until it reaches the end of the input file. Once it runs out of off-design data, the host starts sending out "die" messages to the nodes, telling them to clean up their temporary files and shut themselves down. The host program then closes its files and exits, leaving the nodes to exit at their earliest possible opportunity.

6 Event-Driven Approach to Scheduling

Since many of Distributed NEPP's operations take place simultaneously, **nepphost** must use a fairly sophisticated approach to decide what to do next. The decision becomes even more complicated because the results from the nodes will probably come back in an order different from the one in which **nepphost** sent them out. However, Distributed NEPP guarantees that it will write the results to the output file in the same order that they appear in the input file; therefore, **nepphost** often has to sort and store results until it has the next one it needs to write to the output file.

To simplify the program's organization and to make it easier to modify, **nepphost** uses an event-driven paradigm in its design. **nepphost** divides everything that can happen into a series of events; it then spends most of its time executing an event loop. At each pass through the event loop, **nepphost** considers what has occurred and, based on a list of priorities, it decides on an event handler to trigger. It makes its decision based on a list of things to do in order of priority (from highest to lowest):

1. If a fatal error has occurred, abort this program.
2. If necessary, reset the host and the nodes.
3. If there is no job (i.e. no off-design case) ready to go to a node, get one ready.
4. If there is a node available to handle a job, send the job to the node.
5. If a node has sent a message saying it is done initializing itself, receive the message.
6. If a node has sent a message saying it has finished the job it was working on, receive that message.
7. If a node has requested a copy of the thermodynamic data file, send the copy.
8. If results are available to write to the output file, write them.
9. If there is no more work to do, quit.
10. If there is nothing to do right now, but there are more off-design cases to execute, enter an idle state that consumes no CPU cycles and wait for something to happen.

The event-driven paradigm makes it easy to modify the priority **nepphost** assigns to the various situations; in fact, changing priority levels requires modifying only a single function. Also, adding event handlers to handle new situations presents few difficulties. The interface to the IPAS code demonstrates how easy this paradigm makes modification: the IPAS interface uses a different set of event priorities but most of the same event handlers that the stand-alone program uses.

7 How to Interface Distributed NEPP With Other Programs

Other programs such as the IPAS system may need to use NEPP to perform some of their computations. Distributed NEPP provides facilities that other codes can use to execute its NEPP jobs either in parallel or sequentially.

The Distributed NEPP source code includes two different interface modules. Linking with the first, **neppint.c**, creates a stand-alone version of Distributed NEPP that reads NEPP input files and generates output files. The second, **ipasint.c**, provides an interface through which other programs can use Distributed NEPP's parallel execution facilities.

To use Distributed NEPP, IPAS must first call the routine **setup_nepp**. **setup_nepp** and its compliment, **takedown_nepp**, are one-time calls that turn Distributed NEPP on and off, respectively.

When IPAS wants to process a design case followed by a series of off-design cases, it calls the subroutine **parnep** to start processing in parallel. The code can also process sequentially, if necessary, by using **seqnep** instead. Because of the overheads involved in parallel processing, for small numbers of off-design cases it is generally more efficient to execute sequentially rather than in parallel. The critical number of off-design cases varies tremendously with the complexity of the engine IPAS processes, but a good rule

of thumb is to use **seqnep** for fewer than 50 off-design points and **parnep** for 50 or more. As far as the IPAS code is concerned, the two subroutines look and behave exactly alike, and either one can do the work of the other (though perhaps more slowly).

The subroutines **seqnep** and **parnep** need a way to get design and off-design cases from IPAS as they execute, and they must be able to return results back to IPAS. The programmer developing the code that connects to Distributed NEPP is responsible for providing two subroutines, **nextpt** and **result**. Distributed NEPP calls **nextpt** whenever it needs a new design or off-design case to process. Whenever either **parnep** or **seqnep** calls **nextpt**, **nextpt** either loads the NEPP common blocks with the next case to process or, if there are no more points, it returns a code indicating that fact. Distributed NEPP uses the routine **result** to return its results back to IPAS; every time **parnep** or **seqnep** has the results of a point, it will make sure those results are in the NEPP common blocks and then call **result** to signal IPAS that it can extract the information. Both **parnep** and **seqnep** guarantee that they will return results in the same order that **nextpt** provides them, since Distributed NEPP stores the results that come back from the nodes and sorts them into the proper order before calling **result**.

This clean interface, involving just 6 subroutines, should make it easy to add Distributed NEPP to other programs that need to quickly calculate off-design engine cycle performance.

8 Limitations of Distributed NEPP

Since this research project focussed mostly on demonstrating that executing NEPP in parallel would show significant improvements over Sequential NEPP, the current version of Distributed NEPP has some limitations that later projects should address.

First, Distributed NEPP is not yet smart enough to take advantage of any special features that may be present on the various nodes. In particular, if a single node program executes on a parallel processing machine such as a Hypercube, the node will use only one processor on the machine. This problem arises because currently there is only one type of node program, and that one type of program supports the lowest common denominator in processors, the single-processor machine. It would be possible in the future to add new variants of the node program that can take advantage of additional features present on some of the more exotic parallel processing machines.

The first-come, first-served load balancing approach Distributed NEPP now uses leads to some inefficiency in the program because it distributes the jobs to the nodes in a somewhat random fashion. The NEPP code relies on the results from the previous off-design case to help it converge quickly to a solution for the next. In ordinary Sequential NEPP, the engineer could arrange the input file so that similar off-design cases occur one after the other in the file. Unfortunately, Distributed NEPP's scheduling algorithm tends to send successive off-design cases to different nodes, so that the nodes take longer on average to converge to a solution. Current research is exploring more sophisticated schedulers that attempt to keep consecutive jobs on the same node and show more resistance to the saturation problem section 9 describes.

Finally, Distributed NEPP's current job allocation scheme assumes that all engine input files consist of one design case followed by a series of off-design cases. The program is not yet capable of handling multiple designs in one file or multi-mode engines (i.e., engines with more than one design condition).

9 Results

Research shows that it is possible to obtain significant increases in speed by executing the NASA Engine Performance Program in parallel on a collection of workstations. Testing was performed on a workstation cluster consisting of 32 IBM RS/6000 Model 560 computers connected by two Ethernet networks; one for file sharing and the other for message passing. Each node of the cluster is rated at 40 DP MFLOPS has at least 64 MB RAM and local disk for paging. One IBM RS/6000 Model 970 is used for file serving.

The test input file is for a turbine bypass engine for a high speed civil transport. The engine cycle is optimized for net thrust at each off-design case. The input file contains 300 off-design performance points, a typical size required for mission analysis. The tests were run using a simple table-lookup thermodynamic routine and a more complex chemical equilibrium thermodynamic model. The more complex model increases the execution time of each off-design case by approximately 10 times. Timing runs were made using 1 to 24 processors for both simple and complex thermodynamic models.

Results show the actual amount of speed gain depends upon the amount of time to execute each off-design case. Figure 1 shows typical speedup versus the number of node programs for both the simple (case 1 – without CEC) and complex (case 2 – with CEC) thermodynamic calculations. Figure 2 shows the corresponding efficiency (speedup/number of nodes).

For the simple case, the speedup increases almost linearly until the number of nodes reaches 7. At this point the host *saturates*. That is, the host cannot prepare and send out work fast enough to keep any

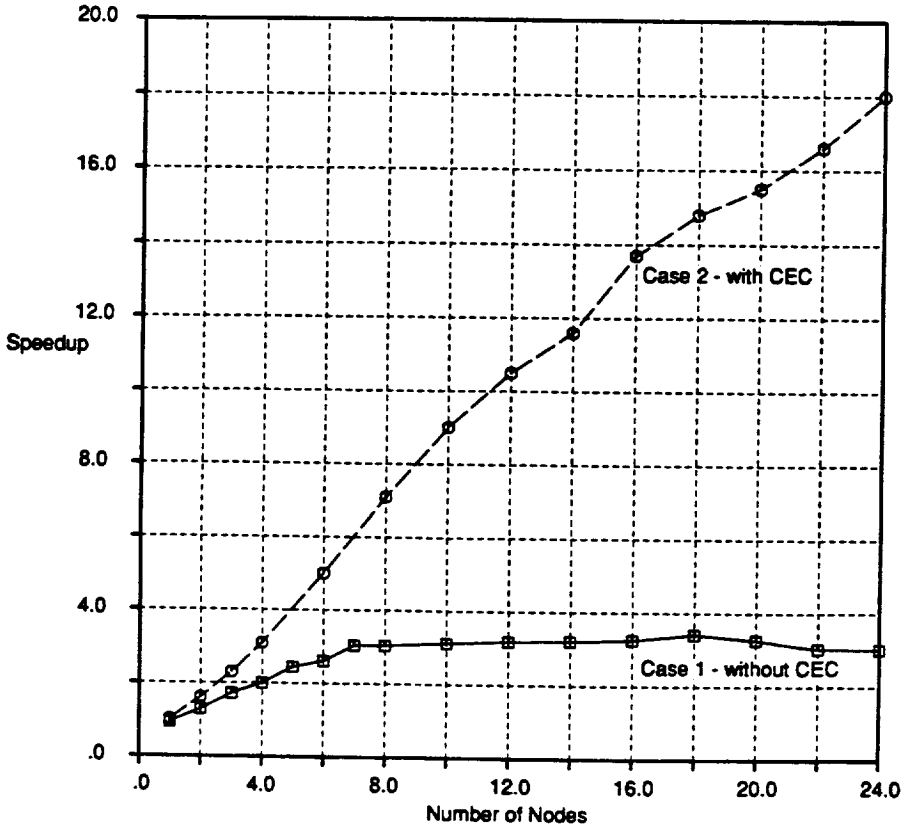

Figure 1: Speedup versus number of nodes

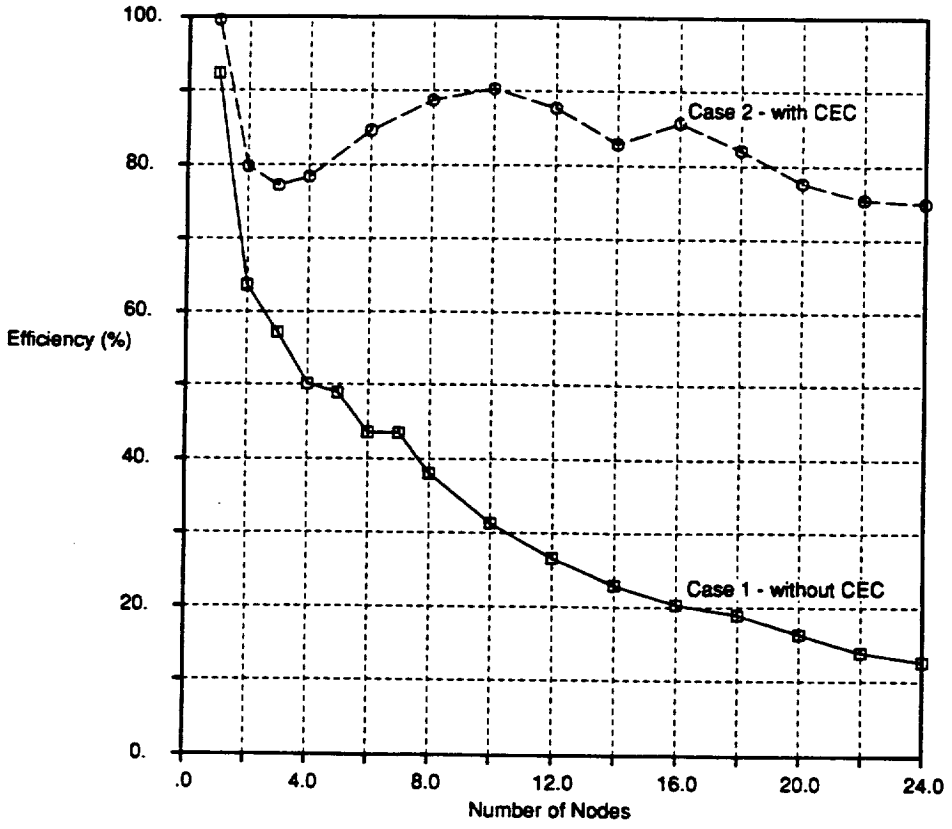

Figure 2: Efficiency versus number of nodes

more nodes busy. This saturation point will vary with changes in network speed as well as processor speed. Each off-design case for this engine model took approximately 0.75 seconds to execute.

The results for the complex case give a better idea of the performance level Distributed NEPP can attain. Each off-design calculation takes approximately 7 seconds to execute allowing the host ample time to complete its work while the node programs remain busy. These results show speedup continuing to increase almost linearly with the number of processors. On 24 processors a speedup of 18 times was achieved reducing a 35 minute sequential job to under 2 minutes in parallel. The fluctuations in the efficiency curve (figure 2, case 2) are due to changes in network and/or cpu loading.

These results indicate that a cluster of relatively inexpensive workstations can achieve supercomputer performance, though no actual measure of floating point performance was made. Furthermore, a dedicated cluster of computers is not necessary to achieve this level of performance: similar results were found while running this code on a group of general use workstations.

10 Concluding Remarks

While programs like Sequential NEPP do not currently take a long time to execute, new programs like the Integrated Propulsion/Airframe Analysis System make more extensive use of the NEPP code, executing it thousands of times in the course of analyzing and designing an engine and airframe. Each call to NEPP may require only a short time, but a few thousand consecutive calls could result in programs that take many hours to execute. Executing in parallel can decrease this time significantly.

Unfortunately, reworking a program to execute in a distributed fashion can require a large effort. For example, converting NEPP to a distributed program required adding at least 15,000 lines of code. Furthermore, it is not always clear how to divide a program so as to make it execute as efficiently as possible. Finally, the maximum performance gains that could come from executing a program in parallel can vary significantly depending on the program and the approach the programmer takes to parallelizing it.

In conclusion, the authors would like to emphasize a point that Section 2 of this paper makes: the real strength of parallel programs lies not only in their speed gains but also in the fact that they become more useful as the problem size increases. Programs like Distributed NEPP, while providing some advantage with the problems Sequential NEPP typically handles, will really prove their worth by allowing engineers to explore far more complex problems than they can with only sequential analysis tools: these new parallel processing techniques will increase the range, rather than just the speed, of what engineers can accomplish.

References

[1] Plencner, R.M.; and Snyder, C.A.: *The Navy/NASA Engine Program (NNEP89) — A User's Manual.* NASA TM-105186, 1991.

[2] Lavelle, T.M.; Plencner, R.M.; and Seidel, J.A.: *Concurrent Optimization of Airframe and Engine Design Parameters.* NASA TM-105908, 1992.

[3] Amdahl, G.M.,Blaauw, G.A.; and Brooks, Jr., F.P.: *Architecture of the IBM System/360.* IBM Journal of Research and Development, 8, no. 2, 87-101, April 1964.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	July 1993	Technical Memorandum

4. TITLE AND SUBTITLE

A Distributed Version of the NASA Engine Performance Program

5. FUNDING NUMBERS

WU-505-69-50

6. AUTHOR(S)

Jeffrey T. Cours and Brian P. Curlett

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135-3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E-7918

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM-106208

11. SUPPLEMENTARY NOTES

Responsible person, Jeffrey T. Cours, (216) 977-7041.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

Distributed NEPP, a version of the NASA Engine Performance Program, uses the original NEPP code but executes it in a distributed computer environment. Multiple workstations connected by a network increase the program's speed and, more importantly, the complexity of the cases it can handle in a reasonable time. Distributed NEPP uses the public domain software package, called *Parallel Virtual Machine*, allowing it to execute on clusters of machines containing many different architectures. It includes the capability to link with other computers, allowing them to process NEPP jobs in parallel. This paper discusses the design issues and granularity considerations that entered into programming Distributed NEPP and presents the results of timing runs.

14. SUBJECT TERMS

Parallel programming; Engine performance; Cycle analysis

15. NUMBER OF PAGES

8

16. PRICE CODE

A02

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	