SOFTWARE THEORY FOR PARALLEL PROCESSORS

Part I

W. Cave & R. Wassmer† - June 27, 2017

THE CHALLENGE

Driven by ever pressing requirements for computer systems that run faster, developers have tried and failed many times to create new computer architectures, [3], [4], [10], [11], and [12]. Now, because of the limitations of today's chip technology, developers are challenged to use parallel processors to meet the ever-increasing speed requirements. This is forcing software developers to break the barriers of complexity incurred when attempting to build applications that are not embarrassingly parallel. This paper describes a theoretical foundation for achieving the desired speed increases when the application is endowed with sufficient inherent parallelism. Using this approach the time and cost to build, support, and run applications have been shown to improve by an order of magnitude - on both single and parallel processors, [9].

DEFINING THE PROBLEM

We start by considering the definitions of parallel processor utilization in a ΔT time interval, ΔT, as illustrated in Figure 1, refer to [8]. This breakout has been selected to simplify analysis of the major factors that can be adjusted to affect parallel processor speed multipliers:

Useful Time - Time spent running instructions required on a single processor.

Overhead Time - Time spent running excess management or communications instructions on a parallel processor.

Idle Time - Time spent in an idle state - waiting to be invoked or waiting for communications.

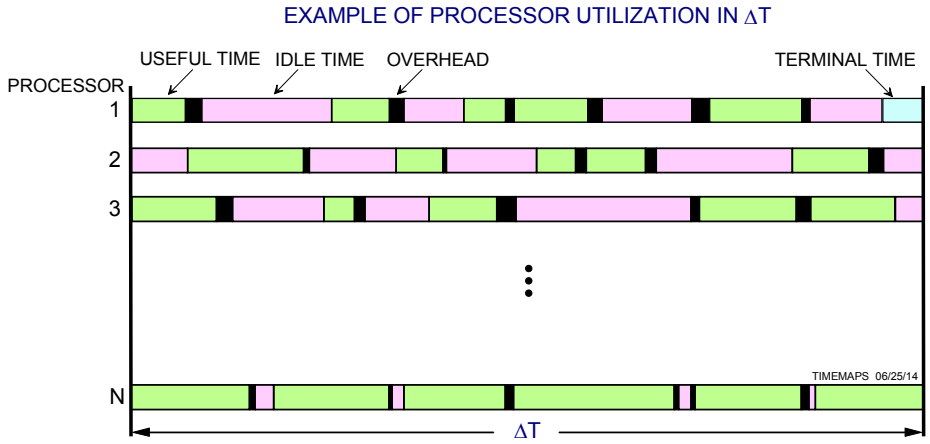

Figure 1. Example of utilization of N processors (rows) in a ΔT time interval.

† The authors are with Visual Software International - www.VisiSoft.com

Figure 1 shows N processors used by a software system running on a parallel processor during a sample interval (ΔT). The green area is processor time doing useful work on a single processor (includes single processor overhead). The black area is processor time spent in excess overhead functions on a parallel processor. The pink area represents time when the task was idle. The blue area is time after which that processor terminates its use (processor 1 happens to terminate its participation in this sample period).

If there is no significant green area overlap, then little useful work is done concurrently and using parallel processors will only yield small improvements. In most cases, with significant inherent parallelism in the system, the lack of overlap is due to poor software design and poor run-time synchronization facilities. This is why most High Performance Computer (HPC) applications tend to produce inefficient results.

Figure 2 shows the relative amounts of processor utilization, overhead, idle time, and termination time grouped together for each processor and ordered by utilization from top to bottom. Clearly the amount of useful work at the bottom of this chart implies significant overlap, whereas at the top of the chart, there may be scarce if any overlap.

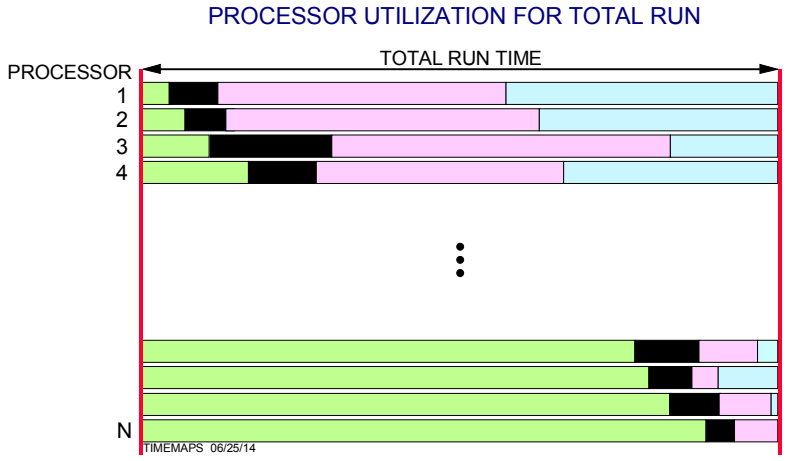

Figure 2. Grouped and ordered processor utilization for entire run.

Estimating Processor Utilization Efficiency

If one plots a curve of the ordered *overlapping* useful time spent on each processor (green bars shown in Figure 2), one finds different outcomes for different software architectures of the same application. Figure 3 illustrates different possible outcomes for 3 different architectures using different shades of green (light, medium, and dark) for 3 of the N processors (Na, Nb, Nc). The lightest shade of green has the least useful overlap and the darkest has the most.

The areas under the curves in Figure 3 represent the corresponding Processor Utilization Efficiency (PUE) when normalized to the time range [0, 1]. The lowest curve Cl has the smallest PUE while the highest curve Ch has the largest. PUE may be increased by shifting all work from one processor to an under-utilized processor, decreasing the number of processors used. But this may increase overall run time if there was useful overlap between the processors. Alternatively, one may shift work from an over utilized processor to an unused processor, decreasing PUE but also decreasing run-time if there is useful overlap between these processors.

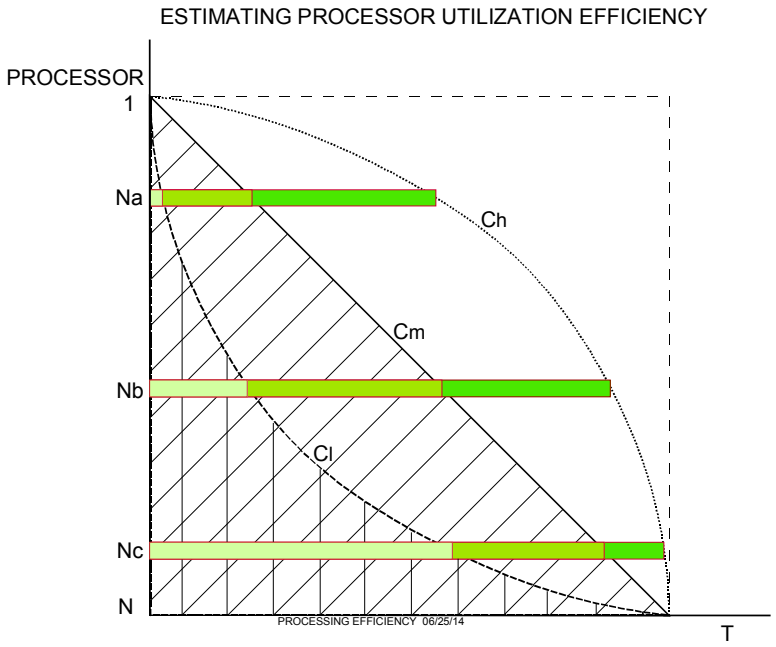

Figure 3. Grouped and ordered useful processor time for an entire run.

Our interest is in developing software architectures that take maximum advantage of the inherent parallelism of a system. As shown in the charts above, without useful time overlap, there is no gain in speed. Experiments show idle time as the major cause of reduction in overlap of useful time on parallel processors. It is caused by the lack of inherent parallelism in a system or, most often, by the lack of a software architecture that takes advantage of the inherent parallelism. There are also significant opportunities to reduce both idle time and overhead when designing an OS to manage parallel processor resources. But such OS designs require special development environment properties described below.

Comparing Parallel Processor Speed Multipliers

The parallel processor Speed Multiplier (SM) is a measure of the time it takes to run an application on a single processor divided by the time taken on a parallel processor.

$$SM = \frac{Ts}{Tp}$$

This is the most important measure when determining the value of running an application on a parallel processor. It must be analyzed carefully to ensure that it has been determined fairly, see Bailey, [2]. When comparing different hardware or software approaches, one must use the same frame of reference, namely the *fastest single processor speed* that has been achieved for that application - independent of the hardware. Otherwise, if we compare software environment A to software environment B, and the single processor time in environment A is longer than that in B, then A may have a higher SM than B only because the fastest single processor time is not used.

Based upon substantial experimentation (see Chapter 18 in [9]), the important conclusions to be drawn are:

- Software architectures must be designed to take maximum advantage of the inherent parallelism in a system.
- High multipliers cannot be achieved without a run-time environment that receives this architectural information from the development environment and uses it to optimize the allocation of processor resources.

Parallel Processor Utilization Efficiency (PUE)

Comparisons of running times on parallel processors must include the cost factor, principally the number of processors. This leads naturally to the Processor Utilization Efficiency (PUE), also defined as the Speed Multiplier (SM) achieved on a parallel processor divided by the number of processors used.

$$PUE = \frac{SM}{Np}$$

Alternatively, the SM is equal to the product of the PUE times the number of processors used. From a software design standpoint, PUE is the most critical factor determining the speed with which an application runs on a parallel processor. Again, for fair comparison, PUE must be calculated using the same (fastest) single processor speed to compute the SM for the application.

PUE depends upon the average amount of useful work done on each processor. When the amount of work done on each processor is highly varied, many processors will have large idle times, and the resulting PUE will be low. When the useful times spent on each processor are all close in size, the average idle time is typically much smaller rendering the PUE much higher.

Achieving Order Of Magnitude Economic Benefits

There are two ways to measure the improvement that one can achieve using parallel processors. The first is to use as many processors as possible to increase the SM of an application. In the second case, a constraint is placed on the time to run the application, and the number of processors required to meet the constraint is minimized using the techniques described below. We note that the number of processors may be reduced by as much as an order of magnitude or more. In this case, a nonlinear reduction typically occurs since the spatial hardware footprint may be reduced considerably (e.g., from 20 square feet to 3 square feet), reducing transmission delays as well as memory boundary crossing delays.

We will use the reduction of number of processors to understand the factors affecting the potential speed improvements. This is a critical point that helps one to take advantage of multiple factors that cumulatively provide large returns, including significant reductions in power consumption and floor space as well as hardware operational costs.

Learning From The Past

Creating software to take maximum advantage of a parallel processor requires that the software be decomposed into independent modules that can run concurrently. This architectural decomposition generally depends the knowledge of a subject area expert to define independent data spaces that reflect the inherent parallelism of the system being developed.

A major drawback of current software languages is the use of abstractions and data hiding. These properties obscure how data is shared (the use of inheritance exacerbates this obscurity). Such approaches inhibit the understanding and determination of the property of independence - the key to software architecture for parallel processors. More importantly, these languages inhibit the creation and use of large hierarchical data structures. This reduces the ability to create data spaces that simplify the design of independent modules. Coupled with the terse and cryptic nature of C-based languages, it is difficult for subject area experts to understand the algorithms, making it hard to control the growing complexity of large software systems - independent of using parallel processors. These problems are described, by Don Anselmo, Director of Computer Development at Bell Labs when C and UNIX were developed, see [1].

KEY CONCEPTS FROM PROVEN THEORIES

Software Spaces For Parallel Processing

Building a dog house does not require the tools needed to build a skyscraper. As software applications grow in size and complexity, designing and building reliable systems requires proven tools and techniques as well as teamwork - on single as well as parallel processors. As stated by Brooks, [6], adding more people to a software project can slow it down. Examples in architectural, chemical, mechanical, and electrical engineering make this obvious. The environment that people share is critical, especially the tools they use to create and communicate their parts of the design. The best examples of such software tools are Computer-Aided Design (CAD) systems. This is stated emphatically by Poore, [13], and Broy, [7], both describing the need for an engineering approach supported by a CAD environment for developing software. The companion to this paper, [8], describes the contribution of key concepts from prior proven theories, and introduces *VisiSoft®*, a CAD environment that makes it easy to implement the desired concepts, see also [9].

To simplify software development on parallel processors, one must be able to map the inherent parallelism in an application into a software architecture such that processes can run concurrently. This implies creating processes that are independent, i.e., they share no data directly. To determine the independence of processes, designers must be able to easily see which processes share what data (memory resources). This can only be done when the following *Critical Software Architecture Requirements* are met:

- Data is organized into a minimum number of structures shared between processes;
- Data structures can be organized into the deep hierarchies required to represent the best spaces to implement problem solutions;
- Designers can easily determine which processes share what data so they can assure their independence properties.

The above requirements are best met when data is separated from instructions at the language level. This separation supports the design of a data language for organizing large data structures using deep hierarchies. It also supports design of an instruction language for building hierarchies of rule structures. Both looping and complex IF ... THEN ... ELSE statements are then flattened. What is known as *Waterfall* or *Fall through* code is gone (without GOTOs). These properties dramatically simplify design of the best data spaces, and concurrently, the design of complex algorithms. Both lead to substantial increases in both understanding and run time speed - on single as well as parallel processors, see [8] and [9].

Software Architecture - Modularity & Independence

In engineering, breaking complex systems into independent modules is embodied in the architecture, a concept that has been misunderstood in software. This is because *architecture describes connectivity*, i.e., how a module is connected to other modules. *Engineering architectures represent the time-invariant properties of a system - not flow of control* (they are not flow charts).

Descriptions of architecture are not convenient using algebraic or linguistic representations. Like other engineering fields, software architecture is best described with drawings, depicting how modules are connected. Only then can one visually observe independence - the key property supporting concurrency. Flow charts, or graphical variations on flow charts, are of little use when describing the property of independence.

SOFTWARE ARCHITECTURE

As illustrated in Figure 4, software architects can decompose a system into modules by grouping resources and processes into an *elementary module*. *Hierarchical modules* are created by grouping modules into higher level modules. Figure 4 shows a library module that is sufficiently complex to warrant its own drawing. In general, modules are independent if they share no resources (i.e., they are not connected). Having designed an architecture, developers can implement the data structures and rules using the *resource* and *process* languages. Using this CAD system, resources and processes may be edited directly on the drawing as illustrated in Figure 6. The languages do not permit the declaration of scope rules. It is the architecture that determines how data is shared, and the corresponding independence of modules. Most important, the languages are designed to provide for deep hierarchies in both data structures and rule structures to support the Critical Software Architecture Requirements defined above. Without these language properties, understandability of complex software is difficult.

Parallelism, Architecture, and Decomposition

When striving to take advantage of the inherent parallelism in a system, one must determine the architecture of software modules that maximizes concurrency on a parallel processor. Picking the best set of state vectors is key to solving this problem. Again, best translates to simplicity of transformations and run-time speed.

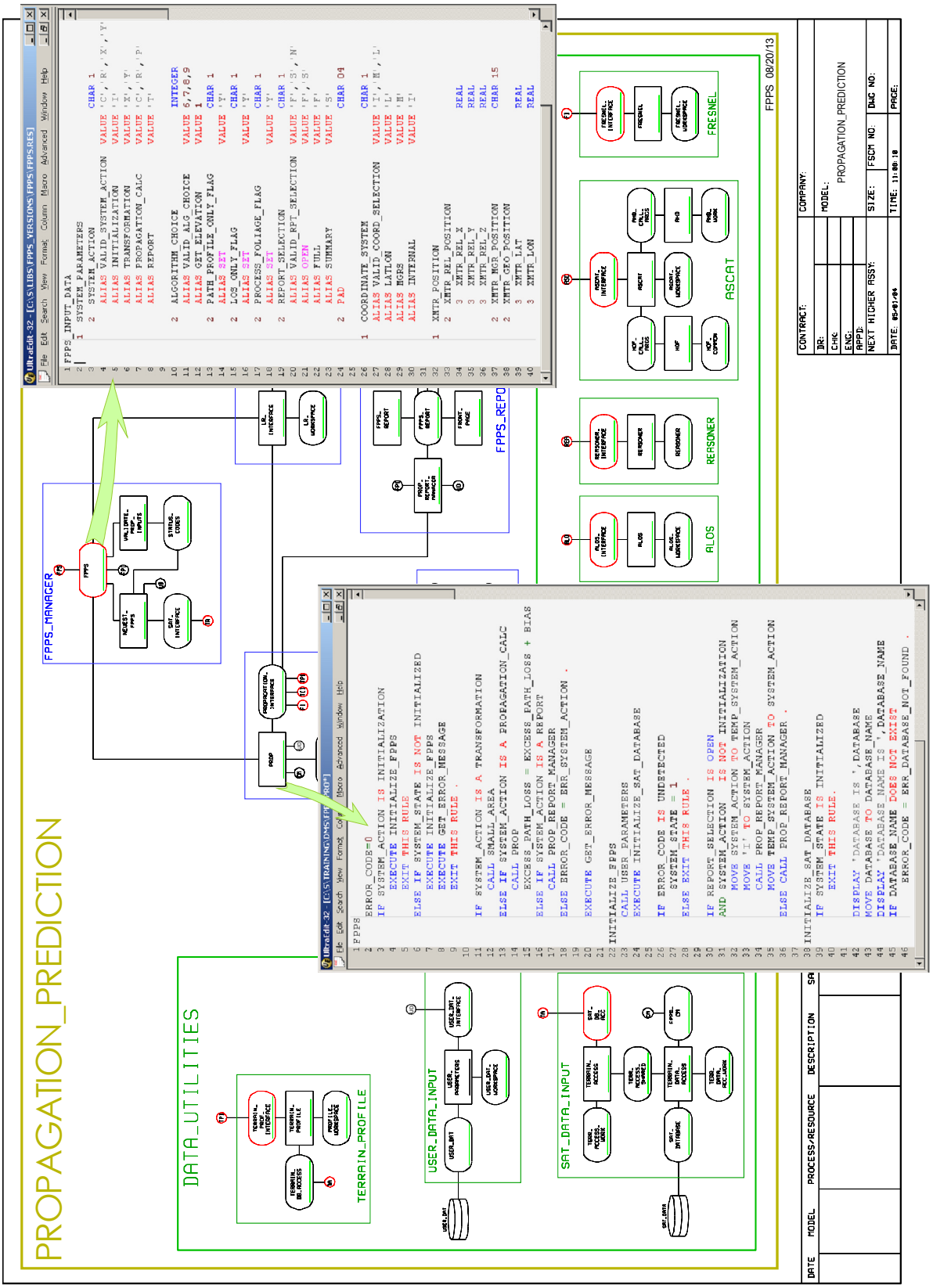

Figure 4. Illustration of editing resources and processes on the drawing.

Having selected Generalized State Space as the framework, the mathematical analogy becomes one of selecting the best set of information vectors (Resources) to represent the system attributes. Depending upon how the resources are designed and structured, the rules (Processes) may be much more simple to understand, build, and modify. This is also determined by the *independence properties of the architecture*, i.e. the interconnection of resources and processes.

Unless one has witnessed directly the development of such architectures, the above discussion may take time to comprehend. Having used it, it is apparent that architecture, as defined here, is as critical to software design as it is to any other engineering discipline, with or without parallel processing. But the ability to design good architectures depends directly on the language. It is why productivity multipliers are very high when using this CAD environment, especially in the support mode when a new person has to understand what another has built. We now turn to the critical importance of language in taking advantage of parallel processors.

SOFTWARE LANGUAGE

The requirements for the resource and process languages were driven in part by factors somewhat akin to those motivating the use of *tiling* in parallel versions of FORTRAN. These are to minimize memory management overhead due to swapping processes and paging data. This is accomplished by maximizing the work done on each processor while running concurrently with work on the other processors, thus maximizing the PUE.

To do this, the language must support design of software spaces that simplify the human translation of inherently parallel physical entities into an organization of independent workloads. As understood by Grace Hopper, likely the most knowledgeable software language designer, [5], such organizations are best supported by deep hierarchies of both data and instructions. A simple example of such a data structure (RESOURCE) is shown in Figure 5.

```
MESSAGE_TABLE       QUANTITY(3)
    1  MESSAGE_INDEX                                INTEGER
    1  MESSAGE_ELEMENT  QUANTITY(13)
       2  UNIT_ID                                   INTEGER
       2  SLOT_ID                                   INTEGER
       2  MESSAGE_INFORMATION
          3  MESSAGE_TYPE                           STATUS  DATA_OUTPUT
                                                            USER_REQUEST
          3  STATE_S
             4  NUMBER_TO_BE_SENT                   INTEGER
             4  SEQUENCE_NUMBER                     INTEGER
             4  MESSAGE_ACTION                      STATUS  SEND, HOLD
             4  AGGREGATE_STATE
                5  MESSAGE_STATE    QUANTITY(7)
                                                    STATUS  EMPTY, FULL
             4  INDIVIDUAL_STATE  REDEFINES AGGREGATE_STATE
                5  SEQUENCED_MESSAGE
                   6  GROUP_MESSAGE                 STATUS  EMPTY, FULL
                   6  BUDDY_MESSAGE                 STATUS  EMPTY, FULL
                   6  QUEUED_MESSAGE                STATUS  EMPTY, FULL
                   6  RESERVED_MESSAGE              STATUS  EMPTY, FULL
                   6  INTERCOM_MESSAGE              STATUS  EMPTY, FULL
                5  NON_SEQUENCED_COMMAND
                   6  DATA_INPUT                    STATUS  EMPTY, FULL
                   6  USER_COMMAND                  STATUS  EMPTY, FULL
```

Figure 5. Example of a hierarchically structured state vector (RESOURCE).

Deep hierarchies allow large complex data structures to be moved in a single instruction fetch, with all of the individual fields directly available to instruction hierarchies as illustrated in Figures 6, 7, and 8. This provides order of magnitude improvements in single processor speeds as well as understanding, see the experimental results in Chapter 17 in [9].

```
PROCESS_CLASS_MESSAGE
    IF  MESSAGE_ACTION(CONTROL_UNIT, RADIO) IS SEND
    AND MESSAGE_TYPE(CONTROL_UNIT, RADIO) IS USER_REQUEST
        MOVE INDIVIDUAL_COMMAND(CONTROL_UNIT, RADIO)
             TO AGGREGATE_STATE(CONTROL_UNIT, RADIO)
        EXECUTE CHECK_MESSAGE_INDEX .

CHECK_MESSAGE_INDEX
    IF MESSAGE_INDEX(TIME_SLOT) IS GREATER THAN ZERO
        SET MESSAGE_ACTION(CONTROL_UNIT, TIME_SLOT) TO HOLD
        SEQUENCE_NUMBER(CONTROL_UNIT, TIME_SLOT) = MESSAGE_INDEX(TIME_SLOT)
        MOVE AGGREGATE_STATE(CONTROL_UNIT, TIME_SLOT)
             TO INDIVIDUAL_COMMAND(CONTROL_UNIT, TIME_SLOT) .
```

Figure 6. Example of part of a hierarchically structured PROCESS.

```
MESSAGE
    1  SYNC_CODE                    CHAR 6
                ALIAS  VALID     VALUE '101010',
                                       '010101'
    1  TYPE                         STATUS FORMAT_A
                                           FORMAT_B
    1  CONTENT                      CHAR 46

FORMAT_A    REDEFINES MESSAGE
    1  PAD                          CHAR 14
    1  HEADER
       2  PRIORITY                  STATUS FLASH
                                           IMMEDIATE
                                           ROUTINE
       2  ORIGIN                    INDEX
       2  DESTINATION               INDEX
             ALIAS   BROADCAST           VALUE 0
    1  BODY
       2  LENGTH                    INTEGER
    1  TRAILER
       2  MESSAGE_NUMBER            INTEGER
       2  TIME_SENT                 REAL
       2  TIME_RECEIVED             REAL
       2  ACKNOWLEDGEMENT           STATUS RECEIVED
                                           NOT_RECEIVED
       2  LAST_SYMBOL               CHAR 2
            ALIAS  TERMINATOR     VALUE '\\', '//', '<<','>>'

FORMAT_B    REDEFINES MESSAGE
    1  PAD                          CHAR 14
    1  HEADER
       2  SOURCE                    INDEX
       2  SINK                      INDEX
    1  BODY
       2  CONTENTS                  CHAR 42
```

Figure 7. Example of a hierarchically structured state vector (Resource).

```
 PROCESS: RECEPTION

 RESOURCES: TERMINAL_PARAMETERS      INSTANCES: TRANSMITTER
            MESSAGE_FORMATS                     RECEIVER
            TRANSCEIVER
```

```
START_RECEPTION
    IF TRANSCEIVER IS IDLE
        EXECUTE GOOD_RECEPTION
    ELSE IF TRANSCEIVER IS RECEIVING
        EXECUTE CONFLICTING_RECEPTION
    ELSE IF TRANSCEIVER IS TRANSMITTING
        EXECUTE CONFLICTING_BROADCAST .

GOOD_RECEPTION
    IF SIGNAL_TO_NOISE_RATIO IS GREATER THAN

RECEIVER_THRESHOLD
         SET TRANSCEIVER TO RECEIVING
         ADD SIGNAL POWER TO POWER_AT_RECEIVER
         CALL DECODE_MESSAGE
    ELSE EXIT THIS RULE .

    IF SYNC_CODE IS VALID
    AND LAST_SYMBOL IS A TERMINATOR
    AND MESSAGE TYPE IS FORMAT_A
        EXECUTE SEND_ACKNOWLEDGEMENT .

CONFLICTING_RECEPTION
    IF POWER_AT_RECEIVER IS GREATER THAN SIGNAL_POWER
         SCHEDULE ABORT_RECEIVE NOW .

CONFLICTING_BROADCAST
    CANCEL END_RECEIVE NOW
    SCHEDULE START_RECEIVE IN EXPON(0.83) MILLISECONDS
        WITH PRIORITY 80

SEND_ACKNOWLEDGEMENT
    MOVE ACKNOWLEDGEMENT TO TRANSMIT_MESSAGE_BUFFER
    IF DESTINATION IS BROADCAST
         SEARCH RECEIVER_CONNECTIVITY_VECTOR OVER RECEIVER
              EXECUTING TRANSMISSION
                   WHEN LINK IS GOOD
    ELSE EXECUTE TRANSMISSION .

TRANSMISSION
    SCHEDULE RECEPTION
       IN LINK_DELAY MICROSECONDS
           USING TRANSMITTER, RECEIVER
```

Figure 8. Example of a hierarchically structured transformation (Process).

When building complex software, human translation is simplified if a language supports obvious representation of physical behavior. The examples in Figures 7 and 8 are taken directly from large detailed simulations of Packet Radio networks. With hierarchical data structures like those shown, one can represent the complex algorithms associated with physical systems with ease. This is illustrated in the above figures. Actual systems may entail more complex resources and processes than those shown, but are easily understood by subject area experts.

Not shown in Figure 7 are the QUANTITY clauses used in Figure 5. Likewise, similar corresponding subscripts in Figure 5 are not used in Figure 8. This is because the resource and process pair are part of an instanced module, where instances are automatically handled at the module level, being set when a process within an instanced module is CALLed or SCHEDULEd. Moving instance implementation to the module level substantially enhances understanding of the code.

The first GSS simulation was built for the U.S. Army to support design of a Packet Radio system in the early 1980s. Prior to that, Army engineers used a Program Design Language (PDL) to define the algorithms. When they saw the GSS language they said "If that language compiles directly, you have a contract." As the simulation neared completion, it was determined that the resources and processes could be represented by icons on engineering drawings with lines connecting them - indicating which processes had access to what resources. That has evolved into engineering drawings of software as shown in Figure 4.

Taking Advantage Of Architectural Information At Run-Time

To take advantage of a parallel processor at run-time, the OS must map threads onto processors to maximize the speed multiplier. A designer faced with generating complex algorithms should not be concerned with this problem. Similarly, a traditional compiler will have little success trying to interpret an architect's decomposition of modules from the code. Finally, the operating system will not be very successful in determining where to map threads based upon current run-time statistics, especially if they are nonstationary - as they are in many discrete event simulations due to nonlinearities.

Referring to Figure 9, if sufficient inherent parallelism exists in the system, architects can decompose software into large Independent (IND) modules. As described in [9], threads are contained within IND modules. Because threads in one IND module are independent of those in another, they can run concurrently on separate processors without concern for synchronization.

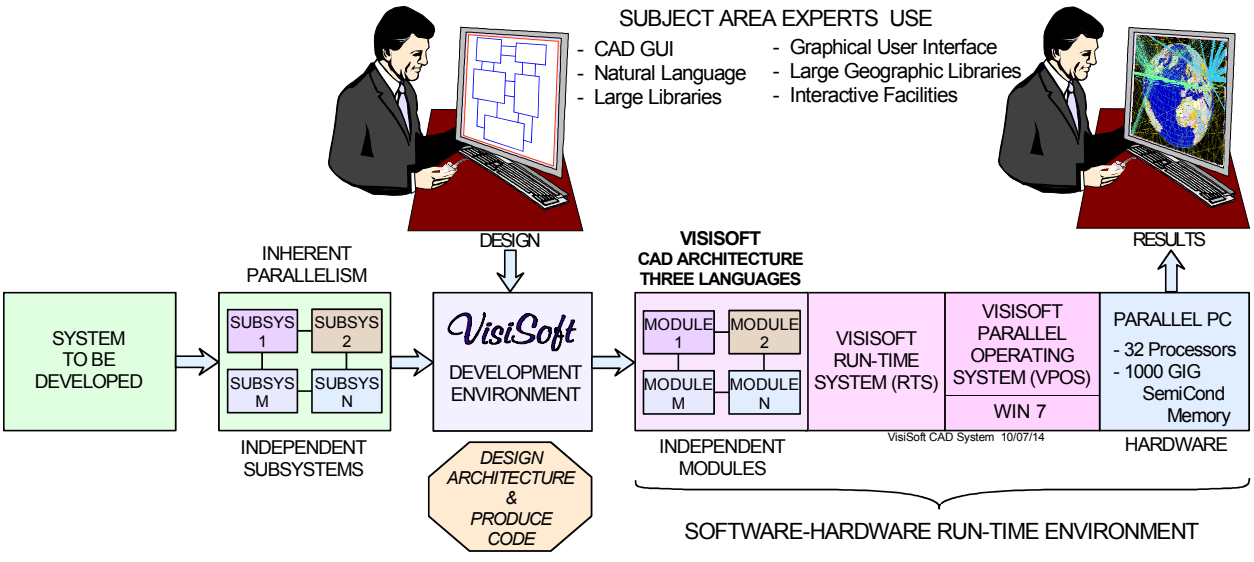

Figure 9. The parallel processor software-hardware environment.

Temporal Independence And Synchronization

IND modules communicate using Inter-Processor (IP) resources. IP resources may be read or written by any process within an IND module, but processes outside that module may only read them. Full duplex channels are implemented using two IP resources - one in each IND module. All IND modules are synchronized in time within a ΔT interval by VPOS (Figure 9) to ensure that the results are complete and consistent.

ΔT is determined by the designer based upon the accuracy requirements on the application results. By *synchronizing the release of, and access to IP resources*, the temporal independence of IND modules that communicate with each other is automatically ensured by VPOS.

The VisiSoft Run-Time System (RTS)

The architectural information that characterizes inherent parallelism is contained in databases that support the CAD development environment. A Run-Time System is generated from that information to control OS calls that assign modules to processors. It also ensures that the resources reside with the processors that use them.

VisiSoft IND modules are typically large and remain on a specified processor, minimizing if not eliminating swapping and paging, and increasing PUE. However, as processor loads become unbalanced, processor utilization efficiency will fall and one must carefully consider the application level design constraints and function to be optimized.

This problem is illustrated in Figure 10 where processor loading is shown in the green area with unused processor time shown in pink and blue. Figure 11 illustrates a reduction in time (about half) using an improved architecture where large IND modules are split into smaller ones. By grouping lightly loaded modules, the number of processors used, M, may also be less than N in Figure 10. We note that VisiSoft takes direct measurements of the loading shown in the figures, and contains a graphical facility to provide a detailed visual depiction of the results, by module - by processor - within interval, along with time stamps. This facility, shown below, enables users to easily balance the processor loadings - by IND module, by processor - based on sufficient statistics from the synchronization time intervals.

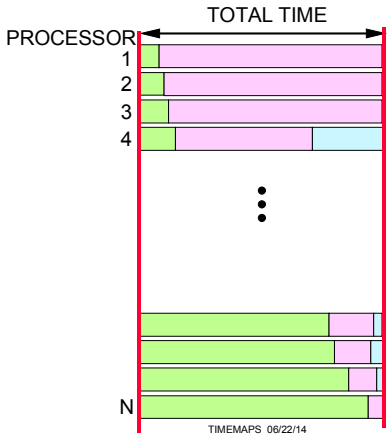

Figure 10. Unbalanced processor loading.

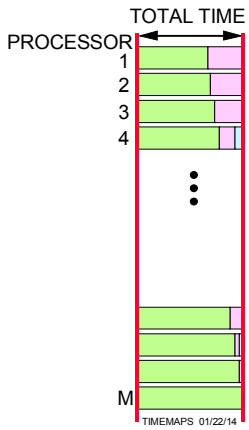

Figure 11. A more balanced processor loading.

Achieving Optimal Resource Solutions

Looking at Figures 10 and 11, design of parallel processor software becomes a relatively simple problem when the IND module statistics are relatively stationary. One can vary the processor loading and assess the results to determine what is best for a given set of application requirements. These criteria must include the cost of resources used (number of processors, electric power, air conditioning, floor space, etc.) as well as time to complete a run.

EXPERIMENTAL RESULTS

The experiments described here use a large simulation of the GPS satellite constellation interacting with 8 other types of air and sea platforms, for a total of 153 moving platforms. When run using a single processor, a complete satellite orbit (12 hours real-time) took 24.15 seconds.

GLOBAL_PLANNER_21 - The 10 & 9 Processor Cases

The parallel processor experiments illustrated here started with 10 processors to house the 10 IND modules on separate processors for "maximum" speed. This resulted in an orbit time of 4.19 seconds. Since the first module, IND_MAIN, was used only for initialization, and since that time was negligible, the second IND module shared the same processor with the first, yielding the same results with 9 processors as with 10.

Compared to the single processor time of 24.15 seconds, the 9 processors produced an SM of 5.764. Using the measures of useful times - within a ΔT time interval - from the test data portrayed in Figure 12, the corresponding PUE ranged from about 55% to about 60%.

Figure 12. Snapshot of IND Module times per processor in ΔT time intervals for 9 processors.

GLOBAL_PLANNER_21 - 7 Processor Case

In the next test, the three shortest IND modules shown in Figure 12 were grouped with the on the first processor with IND_MAIN, yielding the results shown in Figure 13. The orbit time remained at 4.19 seconds, matching the 9 & 10 processor case. This is because the UAV platform (blue bar) still took the longest time with the occasional exception when the F18 ran longer (orange bar). Compared to a single processor time of 24.15 seconds, this produced the same SM of 5.764, but the corresponding PUE was about 92%.

GLOBAL_PLANNER_28 - 14 Processor Case

To cut the orbit time of 4.19 seconds approximately in half, one must double the speed over the 7 processor case shown in Figure 18-16. Because of the high PUE achieved in the 7 processor case, this requires cutting all of the color bars in half - using at least twice as many processors and corresponding IND modules. Assuming the desired constraint can be met using 14 processors, the architecture for this approach is shown in Figure 14. In order to split the 6 large running time platforms (Figure 13) onto two separate processors, a total of 16 IND modules is required. Although this appears to be a substantial increase in the architecture, the new modules remained very similar to the prior ones with minimal new code. With the two small modules still running on the same processor as IND_MAIN, a total of 14 processors is required.

Figure 15 illustrates the results of this experiment, where the simulated orbit time was reduced to 2.31 seconds, about 54% of the 10 and 7 processor cases. The SM was 10.9, yielding a PUE of about 78%.

ANALYSIS OF RESULTS

Increasing the number of processors by 40% over the original 10 processor case increased the speed by 81%. Using the graphical facilities provided by VisiSoft, splitting the modules can be tested easily and results plotted to determine the best fit. We note that, when a number of modules are split, the statistical measure of their variations generally becomes more stationary.

In the special case where IND module loading is not statistically stationary, balancing can be done automatically if the time constants are sufficiently long compared to the time spent balancing. This is the case for most physical systems. In these cases, VPOS can be used to assess the statistics as they are being taken and reassign IND modules based upon knowledge of the loading. We note that the optimization algorithms used to place modules to account for changes in communication delays between modules (who talks to whom when) to minimize memory boundary crossing delays.

Finally, in typical applications, the run time constraint is most important, determining whether application speed requirements are met. When problems are posed with this constraint, one looks to minimize the number of processors, which simplifies the load balancing problem. We note that, when using a large number of processors, speed can rise just due to the smaller footprint, reducing power requirements as well as floor space.

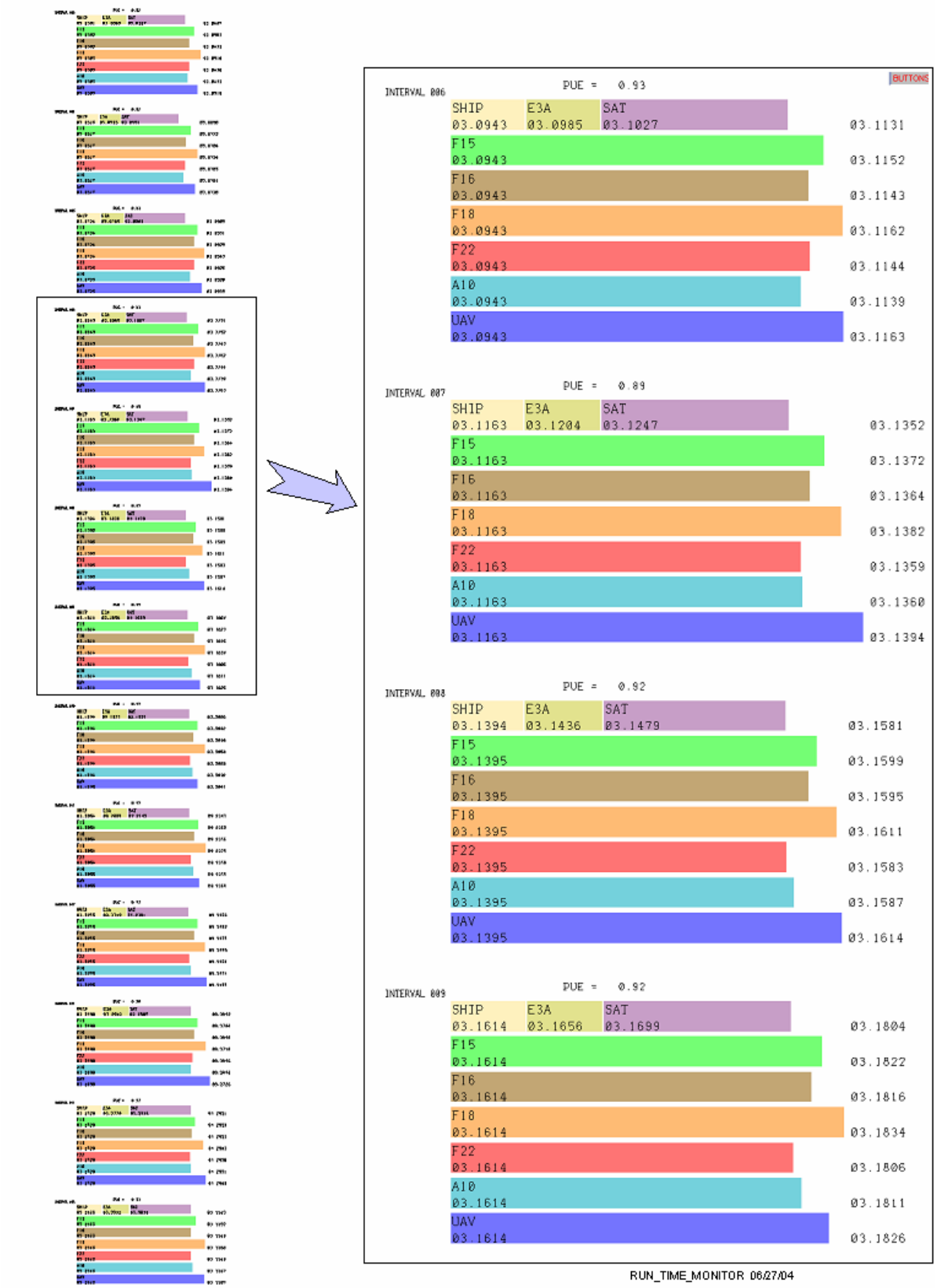

Figure 13. Snapshot of PUE within ΔT time intervals for 7 processor case.

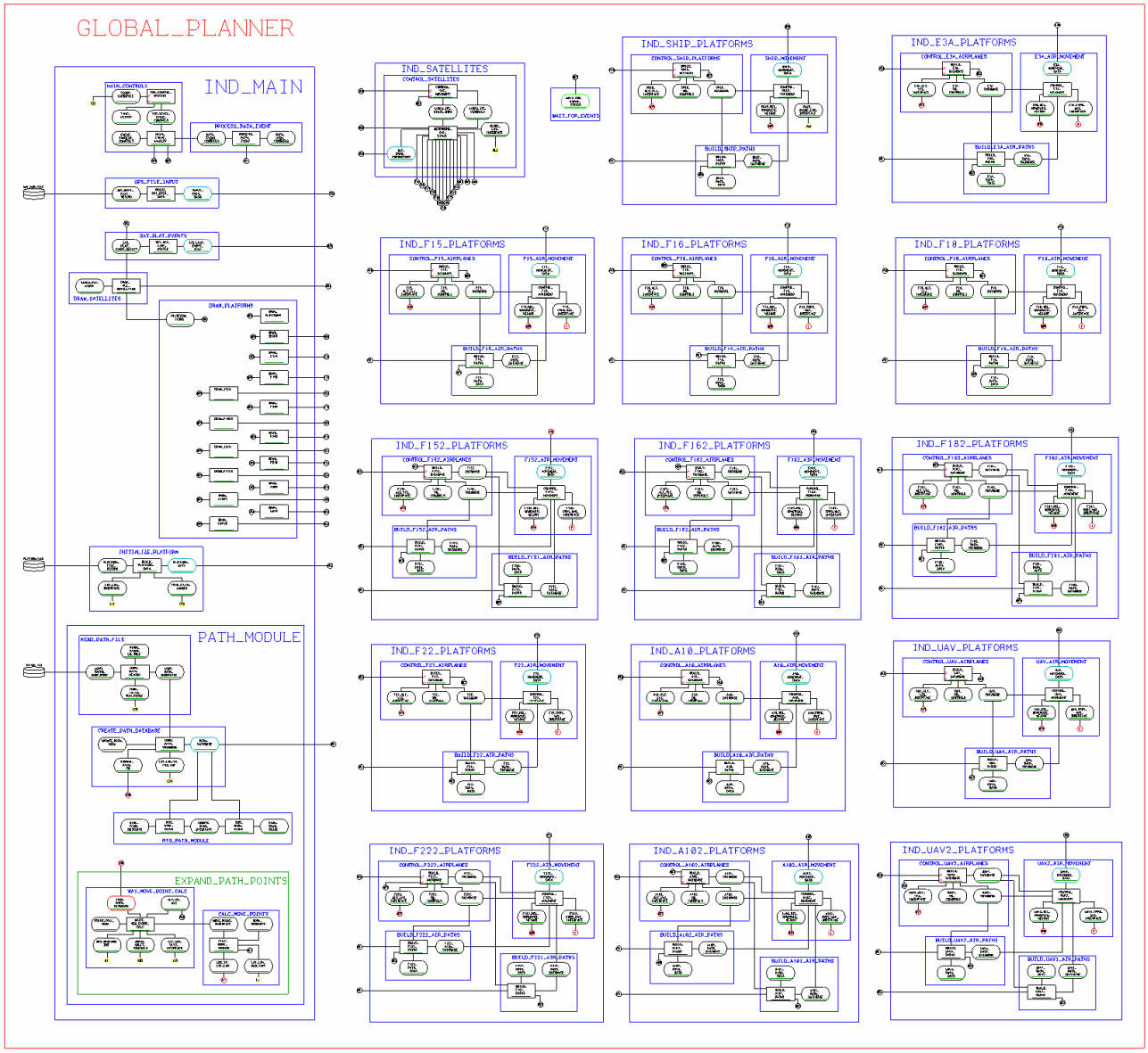

Figure 14. GLOBAL_PLANNER 28 - 16 IND modules running on 14 processors.

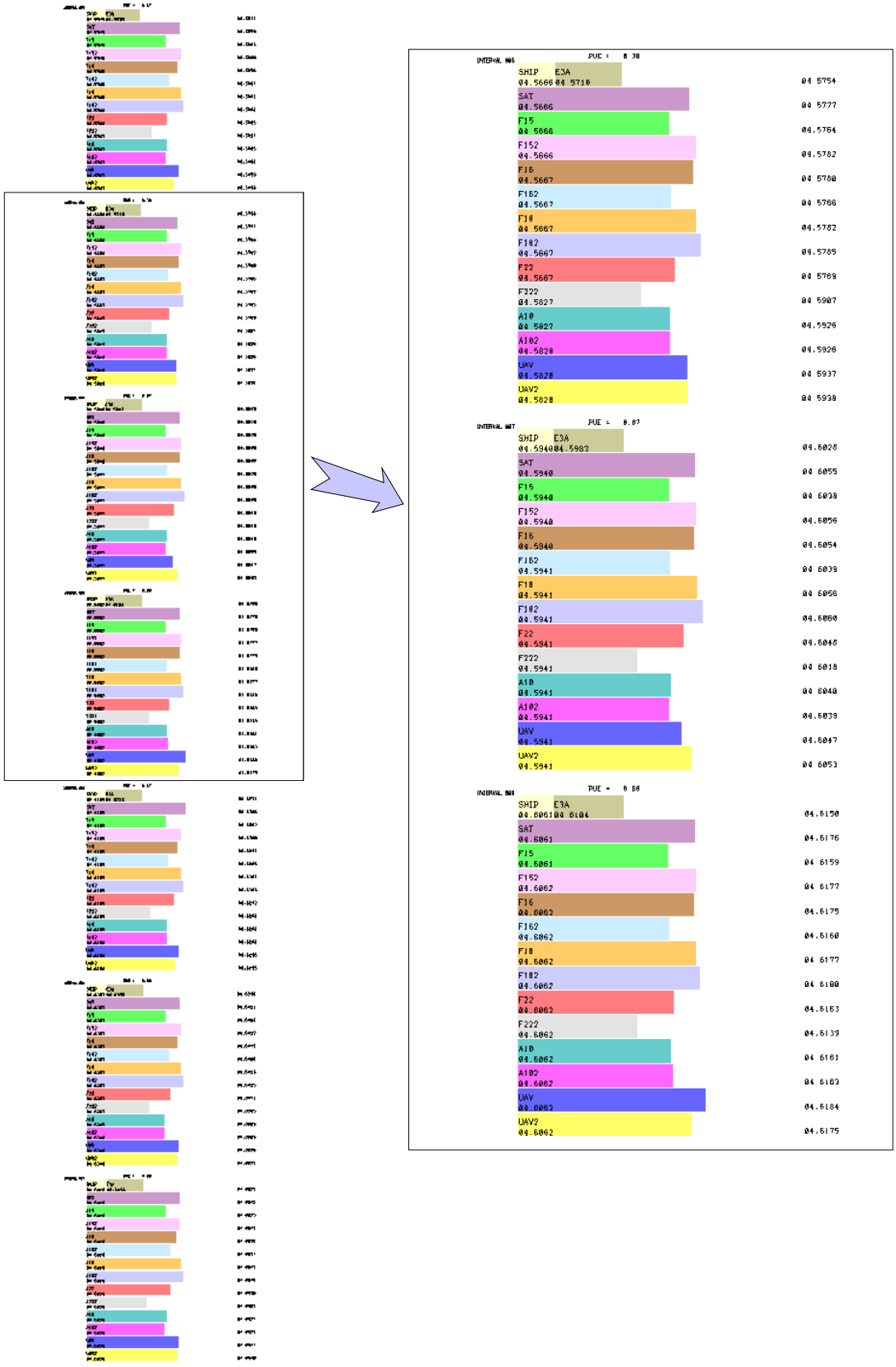

Figure 15. Snapshot of IND Module times per processor per ΔT interval for 14 processor case.

Graphical Facilities For Placement Of Instanced Modules

The figures above illustrate a graphical approach to organizing and placing IND Modules on processors to take advantage of the heterogeneous running times of different modules covering different module types and corresponding running times. By optimizing the approach to packing these modules together to support required processing within a given ΔT, one can minimize the time wasted waiting for different processors to finish the work required within that ΔT.

Figure 16 below illustrates the use of VisiSoft graphical outputs that measure the PUE of Instanced IND Modules within a ΔT. In this example, IND Modules are broken into many instances to provide for improving the PUE. The cases shown in this figure depict the running times on each of 14 processors for three ΔT periods. The different colors represent different IND Modules, each with multiple instances being run on different processors.

The facility illustrated in Figure 16 provides a graphical as well as numerical representation of the times taken by the different instances of IND Modules on a given processor. The sequence of time slots represents the actual running of an instance of each module. The starting times and running times are provided for each module. This figure shows that there is virtually no idle time between the time a process ends and the next process starts to run. In the case that the space required to print the times exceeds the colored block size, the block is extended with no color to indicate that the running time was very short. The actual numbers are used to determine if any time was lost between end times and start times.

Although not shown in this figure, the final times for each process are displayed further to the right of the longest row for a given ΔT. Since the times for each of the ΔTs are all close in this set of samples, these numbers do not appear in the figure. However, these numbers identify that time when the last processor completed running the last IND Module, the time just before the next ΔT could start.

Based on many experiments, the time between sequential ΔTs is always insignificant compared to the time within a ΔT. Thus, the only time wasted is that between the longest and shortest utilization times for those processors being used. One can look at the area of the rectangle covering the start of a ΔT and the end of that longest ΔT as the total time taken to produce the results within that ΔT.

Then consider the blank (unused) area within the end of that rectangle as the wasted time. If the useful time shown in the figure is the same as that on a single processor, then the ratio of the useful time (total time - blank area) to the total time is representative of the PUE. In the tests illustrated here (a nonlinear - nonstationary military application), that PUE measure averaged approximately 95%. This matched closely with the single processor versus parallel processor measures of PUE for that application.

Note that the IND Modules with the smaller running times are generally placed at the end of the ΔT time period. This done purposely to simplify the decision process for moving different IND Modules to different processors to maximize PUE.

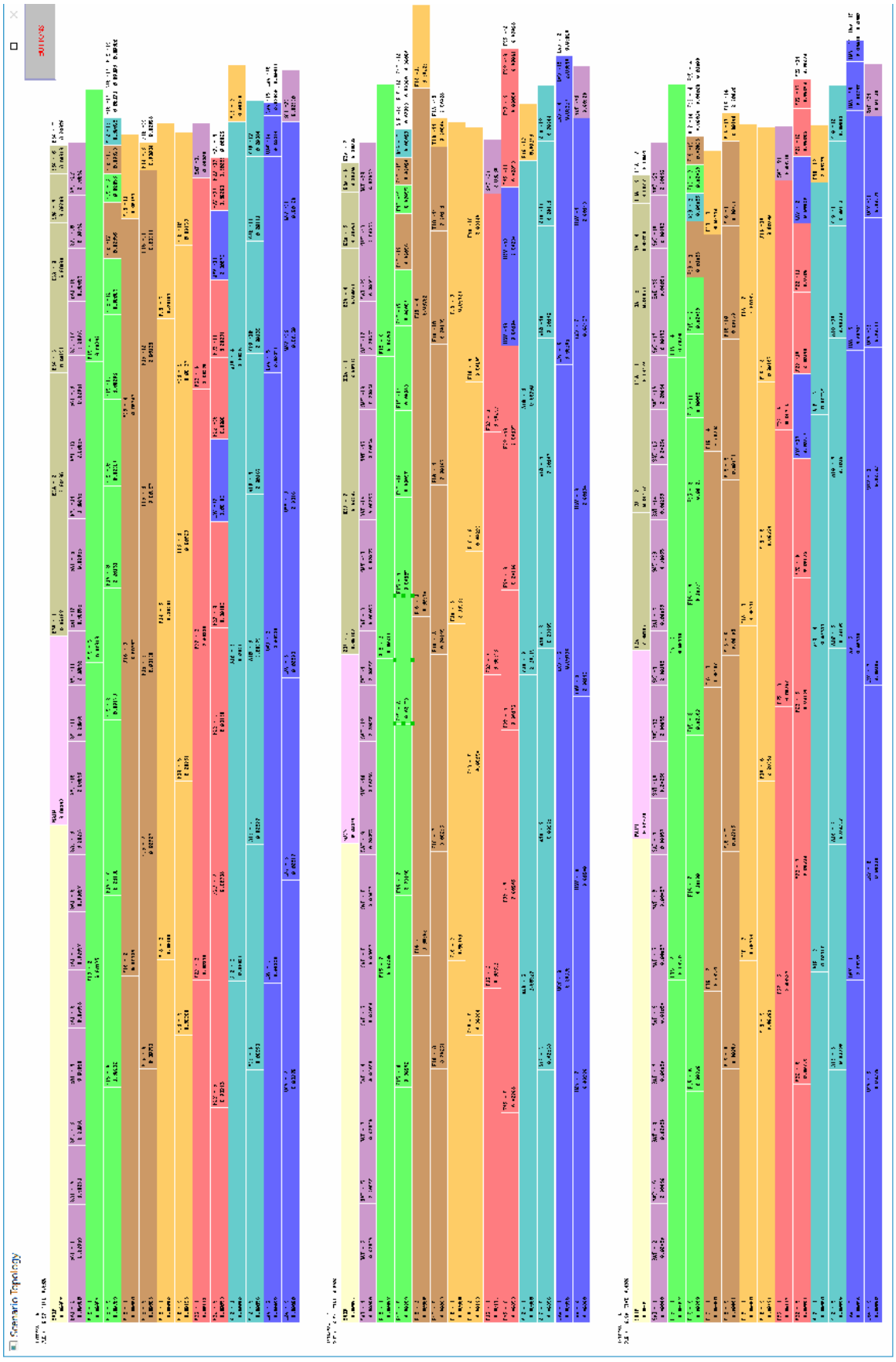

Figure 16. Snapshot of PUE for 3 Delta_Ts, using 14 processors.

REFERENCES

[1] Anselmo, Donald, *Why Software Productivity Has Not Improved*, Software Summit, Washington, D.C., May 2004.

[2] Bailey, D.H., Twelve Ways To Fool The Masses When Giving Performance Results On Parallel Computers, Supercomputing Review, Aug. 1991.

[3] Bell, C.G., "Multis: A New Class of Multiprocessor Computers," Science, Vol. 228, pp 462-467, April 1985.

[4] Bell, C. G., "Ultracomputers A Teraflop Before Its Time," Communications of the ACM, August 1992, Vol.35, No 8.

[5] Beyer, Kurt W., *Grace Hopper and the Invention of the Information Age*. Cambridge, MA: The MIT Press, (2009). ISBN 978-0-262-01310-9.

[6] Brooks, Fred, **The Mythical Man-Month**, Addison-Wesley, Reading, MA, 1975.

[7] Broy, Manfred, *The "Grand Challenge" in Informatics: Engineering Software-Intensive Systems*, Computer, IEEE Computer Society, 2006.

[8] Cave, W.C., et.al, *A CAD System For Developing Complex Software - An Engineering Approach*, Visual Software International, Spring Lake, NJ, September, 2014. http://www.VisiSoft.com/PDF_Files/ACADSystemForSoftware.pdf .

[9] Cave, W.C., et.al, **Software Theory For Parallel Processors**, Visual Software International, Spring Lake, NJ, August, 2014. http://www.VisiSoft.com/PDF_Files/SoftwareTheoryBook.pdf .

[10] Daly, W., "The J-Machine: A Fine-Grain Concurrent Computer; MIT VLSI Memo 89-532, May, 1989.

[11] Holland, J.H., "A Universal Computer Capable Of Executing An Arbitrary Number Of Sub-Programs Simultaneously," Proc EJCC, pps 108-113, 1959.

[12] Patterson, D., Bell, G., et al, "Massively Parallel Uproar," Upside, pps 88-97, March, 1992.

[13] Poore, Jesse H., *A Tale of Three Disciplines and a Revolution*, IEEE Computer Society, Jan 2004.