ARCHITECTURE OF TICC-PPDE, A NEW PARADIGM FOR PARALLEL PROGRAMMING

Ticc (Technology for Integrated Computation and Communication) provides a high-speed message-passing interface for parallel processes. A patent for this has been already applied for (Patent Application Number 102,655/75, Dated Oct. 7, 2003). Ticc does high-speed asynchronous message passing with latencies in the nanoseconds scale in shared memory multiprocessors and latencies in microseconds scale in distributed shared memory supercomputers. Ticc-Ppde (Ticc based Parallel Program Development and Execution Environment) coupled with Ticc-Gui (Graphical User Interface) provides a component based parallel program development environment, and provides infrastructure for dynamic debugging and updating of Ticc-based parallel programs, self-monitoring, self-diagnosis and self-repair. Ticc based parallel programs may be arbitrarily scaled to run in any number of processors without loss of efficiency. Their structure, innovations underlying their principles of operation, details on developing parallel programs using Ticc-Ppde and preliminary results that support the claims are presented in this application.
UNITED STATES PATENT APPLICATION

INVENTOR: Chittoor V. Srinivasan
Citizen of United States of America

TITLE OF PATENT: Architecture of Ticc-Ppde, A New Paradigm for Parallel Programming."

DATE SUBMITTED: December 28, 2005

2031 SE Kilmallie Court, Port Saint Lucie, FL 34952
Tel: 772 285 6463 Fax: 772 337 1388
Email: srinivas(S).cs.rutgers.edu
CUSTOMER NUMBER: 000033641

CROSS REFERENCE TO RELATED APPLICATIONS

1. This is the non-provisional of the provisional application, entitled "TICC-PP: A Ticc-based Parallel Programming System ". Application Number 60/576,152 filed on 09/06/2005;
Confirmation Number 6935, dated 09/15/2005.

2. This is being filed as continuation-in-part of patent application number 10/265,575, Examiner Mr. Lewis Bullock, Art Unit 2195. Patent application 10/265,575 was filed on Oct. 7 2002. Its was entitled, "TICC: Technology for Integrated Computation and Communication," and was published by USPTO on 03/04/2004, Publn. Number US-2004-OO44794-A1.

3. Foreign Filing License Granted, 08/04/2004, under Title 35, United States Code, Section 184; & Title 37, Code of Federal Regulations 5.15.
REFERENCES FROM PUBLISHED LITERATURE


14. M. Ehtesham Hayder, *et al*, "Three Parallel Programming Paradigms: Comparisons on an archetypal PDE computation", Center for Research on High Performance Software, Rice University, Houston, TX 77005, USA, hayder@cs.rice.edu, Parallel Processing Research Group, University of Greenwich, London SE18 6PF, UK, c.ierotheou@gre.ac.uk, Computer Science Department, Old Dominion University and ICASE, Norfolk, VA 23529-0162, USA, keyes@icase.edu.


18. SHMEM: http://www.csar.cfs.ac.uk/user_information/tools/comms_shmem.shtml


29. Chitoor V. Srinivasan, "Writing Parallel Programs in Ticc-Ppde" (in preparation).


32. Souripriya Das, "RESTCLK: A Communication Paradigm for Observation and Control of Object Interactions", Ph.D. Dissertation, Department of Computer Science, Rutgers University, New


44. Robin Milner Communication and Concurrency_ Prentice Hall, 1989

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH

0001 The work that led to this patent application was supported by federally sponsored research, NSF SBIR grants DMI 0232073 and DMI-0349414. However, the inventor enjoys all rights to ownership and use of the patent claimed in this application.

FIELD OF THE INVENTION

0002 These inventions relate generally to the following methodologies: (i) writing Ticc based parallel programs that may be run at near 100% efficiency and may be arbitrarily scaled without loss of efficiency, (ii) component-based parallel program development (iii) defining parallel programming networks using a Graphical User Interface (GUI), (iv) using GUI to dynamically debug and update parallel programs, (v) automatic security enforcement, (vi) infrastructure to develop automatic diagnosis and repair.

COPYRIGHT NOTICE

0003 NONE
BACKGROUND OF INVENTION

0001 Ticc-Ppde is a Parallel Program Development and Execution platform that is based on Ticc. Ticc provides a high-speed message-passing interface with nanosecond latencies. A utility patent application was filed for Ticc on Oct. 7, 2002 (see 0003 below). Parallel programs developed and executed in Ticc-Ppde fully exploit the claimed properties of Ticc, and in addition provides some new capabilities, a Graphical User Interface that simplifies development and maintenance of parallel programs. Inventions in Ticc-Ppde relate generally to the following: (i) Introducing a new model of parallel process execution, (ii) Introducing new programming abstractions that simplify writing of parallel programs, (iii) memory organization that improve efficiency of execution by minimizing memory blocking, (iv) component-based parallel program development methodology (v) a Graphical User Interface (GUI) for developing parallel programming networks, and for dynamic debugging and updating of parallel programs, (vi) specific implementations of Ticc security and privilege enforcement facilities, (vii) infrastructure for self-monitoring, self-diagnosis and self-repair based on principles introduced in Ticc. Items (i) through (v) above constitute the essential ingredients provided by Ticc-Ppde that make it possible to use Ticc in parallel programming environments.

0002 Development and testing of Ticc-Ppde was supported by NSF SBIR grant, DMI-0349414 during 2004 Jan. through 2005 December. A provisional patent application for Ticc-Ppde was filed on Sept. 06, 2005, Provisional Patent Application number, 60/576,152.

DRAWINGS:

Figure 1: ATicc Cell.
Figure 2: Two models of Parallel Processes.
Figure 3: A simple pathway.
Figure 4: A Compound Pathway.
Figure 5: Model of Parallel Computations.
Figure 6: Illustrating Probe Attachment Schemes
Figure 7: In Situ Testing arrangements.
Figure 8: Latency Test Network
Figure 9: Non-Scalable FFT Network
Figure 10: Scalable FFT Network

TABLES:

Table I: A generic poiiPorts ()
Table II: Latency Comparisons
Table III: poiiPorts () for Latency Test
Table IV: poiiPorts () for non-Scalable Version of FFT
Table V: poiiPorts () for Scalable Version of FFT
Table VI: Timing Statistics for FFT
SUMMARY

1. MOTIVATION

There have been several discussions of different paradigms for parallel programming, based on languages and libraries used [14,40], based on different message passing interfaces [15,16,17,18,19], based on data flow models [20,22], based on different hardware architectures [21,23], and based on network models [23,24]. All of these accept the inevitability of two fundamental incompatibilities: (i) Incompatibility between communication and computation speeds; this is usually compensated for by increasing grain size\(^1\) of computations, and (ii) Incompatibility between speed of data access from memories and CPU speeds; this is usually compensated for by using cache memories, pipe-lined instruction execution and look ahead instruction scheduling.

These approaches, however, do not facilitate arbitrary scaling at high efficiencies. To write, debug and run parallel programs with large scaling factors and high efficiencies, and to maintain and modify them easily we need new methodologies.

Parallel programs are costly to develop and maintain, requiring special expertise not commonly available. Parallel programming community has learnt to live with this reality. One seeks higher productivity [12,13] by increasing peak flops/sec yields of parallel processors, while maintaining compatibility to run existing parallel programs. Commodity based massively parallel cluster computing will find its limits in realizable flops/sec efficiency (which is currently about 30%), realizable flops/unit-of-power efficiency and flops/unit-of-space efficiency measures. These efficiencies are likely to decrease dramatically at scaling factors of 10^4 or 10^6.\(^2\)

With nano-scale computing [33] fast approaching (in the coming decade) and quantum computing (in the next two decades), we may confront a need to perform massively parallel computations at 10^4 and 10^6 scales. To do that we will need (i) new models of parallel computations, and new methods to (ii) develop parallel programs, (iii) debug and maintain them, (iv) run them efficiently with very small grain sizes, (v) manage message passing with nanosecond latencies, and (vi) organize memories and CPUs. The paradigm introduced here will provide means to address these issues. Even without pressing need to scale by factors of 10^4 or more, the new paradigm has immediate benefits to offer: It can easily

---

\(^1\) Grain size as the average amount of time spent on computations between successive message passing events in a parallel processing system.

\(^2\) It is claimed, Blue Gene has been scaled up to 10^5 processors. Papers indicate 30% efficiency with 9000 processors. It is not clear whether applications using all the 10^5 processors have been written and tested. Our objective here is to write and run parallel programs at near 100% efficiency, independent of the scaling factor.
double the performance of any existing shared-memory supercomputer\(^0\), at low grain sizes of the order of 50 to 100 microseconds with scaling with in limits of hardware technology.

0008 **Ten Requirements:** We posit the following as being essential to fully realize large scale, easily manageable parallel computations in any technology: (i) very low communication latencies, (ii) efficient running of parallel programs with message-flow driven self-synchronized scheduling, (iii) automatic asynchronous execution and coordination, (iv) processes using only local data and data received through message passing, (v) computations supported by local memories, shared with processes in given neighborhoods, (vi) messages communicated using only local memories, (vii) methods for dynamically debugging parallel programs, and infrastructures for deploying (viii) security and protection, (ix) dynamic modifications and updating, and (x) self monitoring and self repair.

0009 Scalable parallel processing hardware networks appear in cellular automata [25,26], systolic machines [20], and asynchronous data-flow machines [22]. We present here a new paradigm for developing and running scalable parallel software systems, which have the potential to satisfy all the above characteristics.

0010 We have built prototypes of Ticc\(^4\) (new Technology for Integrated Computation and Communication), and Ticc-Ppde\(^5\) (Ticc-based Parallel Program Development and Execution Environment) with Ticc-Gui (Graphical User Interface). At present, the prototype Ticc-Ppde can be used only in shared-memory environments. Test results are shown in (Section 8).

0011 Message passing latencies were of the order of 350 nanoseconds to 3.5 microseconds for messages of length 0 through 1000 bytes. We built two versions of complex double precision one-dimensional FFT (Fast Fourier Transform) [36], one not scalable and the other potentially scalable. Both ran at 100% to 200% relative efficiencies\(^6\) at grain sizes of 50 to 100 microseconds (Section 8.2) [29,30].

0012 In the following, we introduce the two fundamental concepts that gave rise to this new paradigm: (i) a new model of parallel processes and (ii) integrated computations and communications. We show how abstractions introduced by the new model simplify parallel program development and, together with integrated computation and communication, provide a rich collection of benefits which have the potential to satisfy all the ten requirements mentioned above.

---

\(^3\) We use the terms shared-memory “supercomputer” and “multiprocessor” interchangeably. In this paper, these terms always refer to shared-memory multiprocessors or tightly coupled distributed-memory supercomputers, where each memory unit is shared by a group of processors in a neighborhood and adjacent neighborhood groups may write in each others memory.


\(^6\) Two hundred percent relative efficiency is obtained here because of cache memory limitations. Cache could not hold all needed data in sequential computation with one processor. Since data were split among several processors in parallel computation, each processor could hold all data it needed in its cache.
1.1 OVERVIEW OF DISCLOSURE

We begin Section 2 below with a brief description of top-level structure of Ticc. This sets the basis for describing in Section 3 the two innovations that gave rise to Ticc-Ppde, consequent properties, benefits they confer and some operational details. We then present in Section 4 (first section of detailed description) a brief historical background and point out bottlenecks we have inherited. Elements of Ticc are introduced in Section 5. We begin by comparing Ticc with MPI [15], CSP [34] and D-Calculus [42, 43, 44, 45] and then describe the structure and operation of Ticc. Parts of this Section were presented already in our patent application on Ticc [28c]. They are presented again here for convenience. Paragraphs that review features of Ticc are marked "(Ticc)" and paragraphs that define or comment on features of Ticc-Ppde are marked "(Ticc-Ppde)". Section 6 introduces Ticc models of sequential and parallel computations [28c] and points out the change in the Ticc models that Ticc-Ppde introduced in order to simplify parallel programming. Section 7 gives a brief overview of the structure of implementation of Ticc and Ticc-Ppde. Section 8 summarizes three Ticc-Ppde parallel programs and presents test results. This is followed in section 9 by concluding remarks. Ticc and Ticc-Ppde are closely intertwined, each adding to the other to create this new parallel programming and execution environment.

The tests provide a proof of concept demonstration of the new paradigm that shows concepts in the new paradigm are sound practical and can be generalized to distributed shared memory supercomputers.

2. TOP LEVEL STRUCTURE OF TICC-PPDE

Ticc and Ticc-Ppde are both written in C++ and run with LINUX operating system. Ticc-Ppde provides an API (Application Programmer Interface) to develop and run parallel programs in LINUX C++ development environment. Ticc-Gui may be used to set up, debug, run and update Ticc-based parallel processing networks.

2.1. STRUCTURE OF CELLS AND COMPUTATIONS

Parallel computations in the new paradigm are organized around active computational units called, cells. Cells contain ports. The cell to which a port is attached is called the parent cell of the port, which is always unique. Ports of different cells in a Ticc-network will be interconnected by pathways. A port may have at most only one pathway connected to it. Cells will use their ports to exchange messages with other cells via pathways connected to them. Message might be a service request sent by one cell to another or it might be a response sent back to the cell that requested service.

Computations performed by a cell in a Ticc-network will consist of (i) receiving service requests, performing requested services and sending back results, or (ii) preparing service

---

7 We use *italics* to mark undefined terms, terms that are defined when they are first introduced or defined later in the text.
requests, sending them to other cells and receiving responses. Each cell in a network will run in parallel with other cells, each in its own dedicated CPU. Cells, ports and pathways are C++ classes with their own data structures and methods defined for them. They are not hardware devices.

0018 (Jjcc) Cells have different kinds of ports. GeneralPorts are used to send service requests and receive replies. FunctionPorts are used to receive service requests and send back responses. A cell may have an arbitrary number of general and function ports. Each cell will also have a set of four designated ports: interruptPort, statePort, diagnosisPort, and csmPort. Details on use of designated ports are not important at this time, except to note that interruptPort is used by a cell to receive interrupt messages from other cells, which may start, stop, suspend and resume computations performed by the cell.

0019 (Ticc) Active constituent of a cell that drives computations performed by the cell is its process, called poiiPorts. The schematic diagram of a cell is shown in Figure 1. Its poiiPorts method is represented in the schematic by its polling arm.

![Schematic Diagram of a Cell](image)

**Legend:**
- GeneralPorts: SEND OUT SERVICE REQUESTS AND RECEIVE REPLIES.
- FunctionPorts: RECEIVE SERVICE REQUESTS AND SEND BACK REPLIES.
- InterruptPort: START, END AND INTERRUPT COMPUTATIONS.
- StatePort: GET AND SET THE STATE OF A CELL.
- DiagnosisPort: DIAGNOSE MALFUNCTIONS.
- CsmPort: REQUEST SERVICES FROM Communications, System Manager, Csm, TO DYNAMICALLY INSTALL/REMOVE CELLS AND PATHWAYS IN A NETWORK.

2.2. **POLLING AND MESSAGE DRIVEN ACTIVATION**

0020 Polling Process and Threads: (Ticc-Ppde) Once a cell is activated it will begin running its poiiPorts process in its assigned CPU. Each poiiPorts process will consist of a collection of threads, at least one for each port of the cell. Cell will use its poiiPorts to poll its ports in some order, in a cyclic fashion, to receive and respond to messages or to send service requests. Message received at a port will determine the thread used to respond to that message. Two threads in the same cell are
said to be dependent on each other if data produced by one are used by the other. They are independent if neither use data produced by the other. Two ports of a cell are mutually independent if all threads at one port are independent of all threads at the other port. Cells in Ticc-Ppde may have mutually independent ports. Port independence is an important property introduced by Ticc-Ppde.

0021 (Ticc-Ppde) We will use \( \text{Th}(P) \) to refer to a thread at port \( P \), \( \text{R}(P, m_l) \) to refer to the part of \( \text{Th}(P) \) that is used by port \( P \) to respond to message \( m_l \), and \( \text{S}(P, m_2) \) to refer to the part of \( \text{Th}(P) \) that is used by \( P \) to send out message \( m_2 \). Task performed at a functionPort, \( fP \), will have the form\(^8\)

\[
\text{Th}(fP) : [\text{R}(fP, m_l), \text{S}(fP, m_2)],
\]

(1)

where \( m_l \) is the received message and \( m_2 \) is the message sent out in reply. For every service request there will be a reply. It is possible that \( \text{R}(...) \) may have some embedded \( \text{S}(...) \) for service requests it might send to other cells in the middle of responding to message \( m_l \). Task performed at a generalPort, \( gP \), will have the form

\[
\text{Th}(gP) : \text{S}(gP, C(gP)),
\]

(2a)

where \( C \) is the computation performed to construct a required service request message. \( \text{S}(fP, C(fP)) \) constructs a service request message and sends it off. When a reply message is received at \( gP \) one may have

\[
\text{Th}(gP) : \text{R}(gP),
\]

(2b)

where \( \text{R}(fP) \) may simply save a pointer to the reply locally or do any other operation depending on application. Reply will be received only after a certain delay. A cell need not wait to receive the reply. It may instead immediately proceed to service another independent port after sending the service request and return later to \( gP \) to receive the reply. This is, of course, possible only if the cell had mutually independent ports.

0022 Message Driven Activation: (Ticc-Ppde) A cell not running its poiiPorts \( () \) will be activated automatically by the first message delivered to it via any one of its ports. After activation, operating system cannot interfere with its computations. Only other cells in the network may influence its computations, by sending messages to the cell. Messages will be exchanged only when data needed to respond to them are ready. Ticc [28c] pointed out this possibility for message driven activation of cells, but it is Ticc-Ppde that actually implemented it and used it to run parallel programs.

0023 (Ticc-Ppde) Activation of a cell in LINUX takes about 2.5 microseconds, more than 6 times the average latency. However, cell activation is done only once for each cell. Once activated, the cell will start running its poiiPorts \( () \) method. Thereafter, every time a new message is sensed at a port the appropriate thread at that port will be automatically activated.

\(^8\) We use ":" to indicate definition of item to its left. Definition appears to its right.
0024 **Process Scheduling**: Ticc-Ppde clones certain parts of LINUX operating system that are involved in process scheduling. Ports use these clones, which are a part of Ticc-Ppde, to make the operating system do their bidding in scheduling and activating processes, and prevent the operating system from interfering with their scheduling decisions. LINUX itself is not changed in any manner.

0025 The novel concepts in Ticc and Ticc-Ppde that makes this new paradigm work are introduced in the next section.

### 3. CONCEPTS IN THE NEW PARADIGM

#### 3.1. NEW MODEL OF PARALLEL PROCESSES

![Diagram of Ticc and Conventional Parallel Processing Models](image)

**Figure 2**: Two Models of Parallel Processes.

0026 **Conventional Model**: A parallel process is usually viewed as a collection of sequential processes communicating with each other by sending messages. This is shown in the top diagram of Figure 2. P₁, P₂ and P₃ are processes of an application. They are running in parallel. Control flows along each process horizontally from left to right. Arrows jumping off these processes represent messages sent by one process to another. For simplicity, we show here only point-to-point message exchange. Facilities like MPI [15] provide mechanisms for exchanging such messages. Processes of MPI that
transmit and deliver messages are distinct from the processes $P_1$, $P_2$ and $P_3$ of the application. MPI may invoke assistance of an operating system to perform its tasks.

0027 New Ticc Model: (Ticc-Ppde) The bottom diagram in Figure 2 shows the model of parallel processes in the Ticc paradigm. $C_1$, $C_2$ and $C_3$ are cells. The ellipses represent the pollPorts of processes of the cells. Small rectangles on the ellipses are the ports. Pathways connect these ports. Cells exchange messages between ports using the pathways. Each pathway contains its own memory (dark disks in Figure 2). This memory will hold the message that is delivered to a port. In the current implementation, this message is defined by a C++ Message class, with its own associated data structures and methods.

0028 Threads: (Ticc-Ppde) Parallel processing computations are performed not by the pollPorts method, but by the little threads that hang down orthogonal to the ellipses. At any time only one thread in each cell will be running. Thus in Figure 2, three threads will be running at any time in the bottom diagram corresponding to the three processes in the top diagram. As mentioned earlier, since threads at different ports of a cell, may perform computations that are independent of each other, threads of any given cell will not together constitute a sequential computation in the conventional sense. However, the three cells together will ultimately perform the same computation that is performed by the conventional model. Ticc model of parallel computation, discussed in Section 6, explains how this is accomplished.

0029 Before discussing the benefits conferred by the Ticc-Ppde model, it is instructive to first explore the structure of pollPorts method, as it would appear in C++. In the discussion below we assume, pathway memory will have only one unique message in it. As we shall later see, models of Ticc parallel computations make this hold true. Hereafter, whenever we say a cell is performing a computation, it should be understood that one of its threads is doing that computation.

3.2. GENERIC POLLPORTS

0030 Integration of Computation & Communication: (Ticc-Ppde) We present here fragments of code in Ticc-Ppde that illustrate the advantages of abstractions introduced in Ticc-Ppde, and a top level view of how computation and communication are integrated in Ticc-Ppde. Whereas in Ticc [28c] cells delegated message transmission to one or more dedicated communication processors, in Ticc-Ppde each cell by itself may directly and immediately transmit messages. No communication processor is necessary. In the following, we will assume familiarity with C++. One may write in C++ the function, $S(P, m)$ used in (1), as the method, $P->s(m)$, where $P$ is the pointer to port $P$ and $m$ is the pointer to message $m$; $P->s(m)$ may be decomposed to, 9

$$P->S(m) := [P->W(m); P->S()] ;$$

(3)

9 We use " := " to indicate code decomposition.
where $p\rightarrow w(m)$ writes $m$ into the memory of the pathway attached to $P$ and $p\rightarrow s()$ sends it off to its intended recipients. $P\rightarrow S()$ will not invoke any assistance from any other process to transmit and deliver $m$. The process that transmits and delivers message will be entirely embedded in the thread $Th(P)$ and thus will be fully executed by the thread itself. The manner in which a cell uses its CPU to send a message is no different from the manner in which in may use its CPU to do an arithmetic operation. It is in this sense computation is integrated with communication in Ticc-Ppde.

0031 **Thread at a Function Port:** (Ticc-Ppde) When a cell senses receipt of a message $m_1$ at a functionPort $fP$, it will automatically immediately activate the thread $Th(fP)$ shown in (1). $Th(fP)$ may be defined as,

$$Th(fP); [R(fP,m1), S(fP,m2)]; fP\rightarrow R(); fP\rightarrow s(); ]$$  \hspace{1cm} (4)

Here $fp\rightarrow R()$ has no reference to received message $m_1$, since it will be the message in the pathway attached to $fP$. One may here think of $R()$ as the process that responds to $m_1$. Suppose, $fp\rightarrow read()=m_1$ is the pointer to the message $m_1$ in $fP$'s pathway memory. Then one may decompose $fp\rightarrow RO$ as,

$$fp\rightarrow R(); = fp\rightarrow W(fp\rightarrow read(); )\rightarrow processMsg (fp);$$  \hspace{1cm} (5)

where processMsgO is the method defined in the message subclass of $m_1$. It processes message $m_1$ and returns a pointer $m_2$ to the reply message $m_2$. $fp\rightarrow w(m_2)$ writes $m_2$ into pathway memory. One may use the *polymorphism* feature of C++ to automatically invoke the right processMsgO associated with the message subclass of $m_1$, no matter what subclass of message it is. Since $fp\rightarrow R()$ would have already written the reply message into the pathway memory no reference to this message is needed when it is sent out. Thus we use $f p\rightarrow s()$ in (4).

0032 (Ticc-Ppde) The pathway memory here will itself provide the execution environment for $mi\rightarrow processMsg (fP)$. Thus, message $m_1$ need not be copied. If the message subclass of $m_2$ that is returned by $mi\rightarrow processMsg(fP)$ always remains fixed then one may write a properly initialized instance of $m_2$ into the memory of pathway at $fP$ at the time the pathway was installed, and simply update it every time $mi\rightarrow processMsg(fP)$ is evaluated. This will simplify (5) above to,

$$fP\rightarrow R(); = fP\rightarrow r e a d(); \rightarrow processMsg (fP);$$  \hspace{1cm} (6)

thereby eliminating one write operation. We refer to messages installed in pathway memories in this manner as *containers*. In all cases, responding to a received message is mandatory in this model.

0033 **Thread at a General Port:** (Ticc-Ppde) In the case of a generalPort, $gP$, one may have

$$Th(gP); [gP\rightarrow C (...) ; ];$$  \hspace{1cm} (7)

where $gP\rightarrow c (...) $ is the computation performed at a general port, defined as

$$gP\rightarrow C (...) := [gP\rightarrow W(gP\rightarrow \chi (...) ); gP\rightarrow S(); ]$$  \hspace{1cm} (8)
where \( gP->x (...) \) is a method, which constructs a message and returns a pointer to it. This might be the message that defines a service request based on its arguments. This will be an application dependent method. In (8) this message is written into memory and sent off. Again, if the message class written into memory is always the same one may use containers and simplify (8) to

\[
gP->C := [gP->X (...) ; gP->S ()],
\]

eliminating one write operation. Later, when a reply message is sensed at \( gP \) one may perform \( gP->R() \), which may simply locally save a pointer to the reply message or do anything else that might be appropriate for an application.

**Grain Size:** \((\text{Tick-Ppde})\) The time spent by a thread at a port to complete its computations will be the **grain size** of parallel computations, which may range from 50 to 100 microseconds in the
current implementation of Ticc. Sending a message will consume about 4uu nanoseconds of this grain size.

0035 **Generic Codes:** (Ticc-Ppde) It may be noted, fragments of code shown above are all generic. Indeed, one may write a generic poiiPorts O as shown in Table i, using these fragments. Implementation of Ticc-Ppde uses generic poiiPorts O like these. For different applications, the message subclasses will be different. Each application will have some variations on the generic code shown in Table i. We present Table i to illustrate simplicity of code generation in the new paradigm.

3.3. **BENEFITS CONFERRED**

0036 Many of the benefits enjoyed by Ticc-Ppde follow directly from this new view of parallel processes.

0037 **New Abstraction Layer:** (Ticc-Ppde) When a cell sends a message via one of its ports, unlike MPI [15], it does not have to specify source, destination, length, data-type, or communicator in the send/receive statements. This information is built into the pathways. No tags or contexts are needed in Ticc since each thread is obligated to respond to a message as soon as it is sensed, and no buffers holding message queues are used (Section 6). One may simply use P->R O and p->s o ; message in memory of a pathway will then be responded to and sent.

0038 (Ticc-Ppde) Pathways thus provide a level of abstraction that decouples source, destination and message characteristics from send/receive operations and local computations. This simplifies programming considerably and makes it possible to dynamically change the structure of parallel processing networks, independent of send/receive operations and computations used in them. One may add/remove cells, ports and pathways without interfering with ongoing computations (Section 7). One may even run the same parallel program on two different networks. Only the initialization methods and poiiPorts () might be different. We will see an example of this in Section 8. This facilitates dynamic reconfiguration. Ticc pathways also play important roles in dynamic debugging, dynamic monitoring and updating of Ticc-based parallel programs, as we shall later see. (Section 7)

0039 (Ticc-Ppde) Pathway abstraction in Ticc-Ppde is analogous to the data type abstraction in programming languages. Pathways introduce a new level of flexibility and generality to specifications of communications in parallel programs, just as data types introduced a new level of flexibility and generality to specifications of operations in conventional programs. There are several other unexpected benefits as we shall see below.

0040 **Security Enforcement:** (Ticc) In Ticc, one may define for each port a security profile and use the pathway connected to the port to enforce defined security at the time of message delivery. Security enforcement at a port may even depend on the number of times message was sent or received at that port; a mode of security enforcement unique to Ticc. Agents attached to pathway memory, small green
discs in Figure 2, perform this function (Section 5.5). Ticc-Ppde implements this security enforcement facility.

Minimizing Memory Blocking: (Ticc-Ppde) In tightly coupled distributed shared memory systems by allocating pathway memories judiciously and defining message classes appropriately, one may avoid both memory blocking and memory contention (Section 6.4). This facilitates arbitrary scaling.

Send & Delivery Synchronization: (Ticc) Only control signals will travel along pathways. Signals traveling along a pathway will establish the context at which message in the pathway memory may be received and responded to by a thread at the port to which it was delivered. When a message is sent from a group of sending ports to another group of recipient ports (we will refer to these groups as port-groups), agents on pathways will be responsible for the following (Section 5.5): (a) receive, gather and forward signals traveling on pathway; (b) enforce message security and protection; (c) synchronize broadcast of message in pathway memory to all recipient ports in a port-group; (d) synchronize message dispatch by ports in the sending port-group, and (e) clock computational and communication events that occur around pathway memory.

Synchronization Levels: (Ticc-Ppde) In the current implementation of Ticc-Ppde, both send and delivery synchronization has two levels of synchronization with increasing precision and cost. Messages are delivered to a recipient port-group of size $g$ in level-1 synchronization with in $2g$ nanoseconds, and $g$ nanoseconds in level-2, where $g$ is the size of receiving port-group. In send synchronization, timings in level-1 and level-3 will be application dependent (Section 5.5).

To get good efficiencies, we believe, $g$ should be $\leq 16$ in the current implementation. In Ticc-Ppde, both send and delivery synchronizations are automatic. They are built in features of Ticc-Ppde with user controls only for specifying the level.

We could not find analogs to these in MPI.

Low Latency Communications: (Ticc) Agents and ports on a pathway that receive and send signals are tuned to each other to guarantee that no agent or port will ever fail to promptly receive and respond to a signal that is sent to it. Tuned ports and agents will thus be always listening to each other at right times (Section 5.2). Thus, no signal will ever be missed and no agent or port need ever wait for synchronization. This contributes to high-speed message exchange with guaranteed message delivery.

Scalability: (Ticc-Ppde) Since threads themselves execute all protocol functions necessary to cause messages to be delivered, and since each cell in a network runs in its own dedicated CPU, all messages will be exchanged in parallel. Number of messages that may be exchanged at any time will be limited only by the number of active cells at that time. Since each port may be connected to only one
pathway, Ticc guarantees message delivery without message interference. These features, coupled with
ability to control memory blocking, facilitate arbitrary scalability, limited only by the available hardware
technology.

0048 The structure of pathways and use of a special Causal Communication Primitive (Ccp) in
programming language that make this kind of communication possible are explained in Section 5.

0049 The engine that drives Ticc-Ppde is the Ticc communication system. In the new paradigm,
Ticc takes over the role that MPI plays in conventional parallel processing. The difference is, Ticc
together with Ticc-Ppde provides practically unlimited number of parallel simultaneous asynchronous
buffer free message transfers, with guaranteed high-speed communications without message
interference, and with automatic asynchronous message driven execution of parallel processes, all
without assistance from application programmer.

3.4. POLLING MOPES

0050 (Ticc-Ppde) We use a weak definition for synchronous and a strong one for asynchronous:
An event in a system is synchronous if its time of occurrence has to be coordinated with the time of
occurrence of another event in the same system. They need not necessarily occur at the same time. An
event in a system is asynchronous if its time of occurrence does not have to be coordinated with the
occurrence of any other event in the system. We will soon see why these notions of synchrony and
asynchrony are unique to Ticc and are different from the way they are used in other systems, including
MPI [15].

0051 Asynchronous Receiving: (Ticc-Ppde) In asynchronous receiving, while polling a port, P, a
cell will not wait for a message to arrive. It will simply check for a message at port P by evaluating, "p->
messageReady ()", and respond to it if one existed, else proceed immediately to poll its next port. This is
asynchronous in the sense, the time at which this happens is not coordinated with any other event. A
cell may check for a received message at any time it chooses. Clearly, threads at a port P and its next
port should be independent if asynchronous receiving is used on P. The generic pollPorts () shown in
Table I uses only asynchronous receiving. We will refer to computations performed with asynchronous
message receipt as asynchronous computations.

0052 Checking and ignoring messages, as is done in MPI, based on tag or context is different from
asynchronous receiving. In Ticc, every thread is obligated to respond to a message as soon as it is
sensed. No tag or context is used in Ticc-Ppde.

0053 Synchronous Receiving: (Ticc-Ppde) In synchronous receiving, cell will use "P->
receive ()" to wait at port P for a message. It will respond to the message when it arrives, and then only
poll its next port. We call this synchronous because starting of the thread that responds to a received
message is in this case coordinated with sending of that message by another thread. This is similar to
blocking receive in MPI. There are differences though, since messages do not have to be copied in Ticc. 
PortPoll.

PollPorts of FFT described in Tables IV and V of Section 8 uses synchronous receiving. We will refer to 
computations performed with synchronous message receipt as synchronous computations. It is always 
harder to write code for synchronous computations than it is for asynchronous ones. In synchronous 
computations, one has to be careful to avoid dead locks.

It should be noted, when synchronous receiving is used at a port, P, it is possible that 
computations performed at P and its next port in a cell may be dependent on each other. This happens, 
for example, in the FFT code shown in Tables IV and V (see Detailed Description, Section 8).

Asynchronous Sending: (Ticc-Ppde) In asynchronous sending of messages cell will use 
"p-> pathwayReady ()" to check whether pathway at port P is ready to send a message. If it is, then it 
will send its message, else proceed immediately to poll its next port. This is asynchronous because the 
time at which a cell chooses to do this is not coordinated with any other event. Again, threads at port P 
and its next port should be independent.

(Ticc-Ppde) It may be noted asynchronous receiving and sending are feasible in Ticc-Ppde 
only because it is possible for adjacent ports in a cell to be independent. No analogs to these exist in 
MPI [15] or CSP [34]. In CSP, all communications are synchronous in the sense of Ticc.

Synchronous Sending: (Ticc-Ppde) In synchronous sending, cell will use "P-> 
sendImmediateif Ready()" to wait for pathway at a port to become ready and then send message. It will 
poll its next port only after sending the message. This is synchronous because readiness of a pathway 
here requires coordination with another thread. In certain ways, synchronous sending in Ticc-Ppde is 
similar to non-blocking MPI-send where a process waits for a buffer to be cleared. Again, there are 
differences; Ticc has no buffers.

There is no need in Ticc-Ppde for synchronous sending in the sense of MPI, where a sender 
waits for its intended recipient to become ready to receive a message. This is because cells will always 
send messages at any time they please, if pathway is ready. Recipient need not be ready to receive the 
message.

Suspend/Resume mode of Polling: (Ticc-Ppde) In the middle of responding to a service 
request received via one of its functionPorts, fP, if a cell had to send a service request to another cell, 
then after sending the service request via one of its generalPorts, gP, the cell will not wait to receive a 
response. It will suspend its current operations at fP and proceed to poll its next independent port. It will 
resume the suspended operation at fP later when it polls fP again, if response to its request had been 
received by then by gP. Such suspend/resume operations will be done automatically without need for 
operating system intervention. Thus, no cell will wait at a port to receive a message, unless it was 
specifically designed to do so in carefully coordinated synchronous computations. Together with low
latency communications, this contributes to high efficiency. Again, it may be noted, this mode of operation is feasible only if threads in a cell have mutually independent pairs.

0060 (Ticc-Ppde) Two cells, X, Y, will be in a deadlock if they are blocking each other from proceeding further with their computation. This may happen if X is waiting for a response from Y to proceed further, and similarly Y is waiting for a response from X. Since no cell waits for response from another cell in Ticc-Ppde, except for purpose of coordinated synchronous computations, no deadlocks will occur in Ticc-Ppde.

3.5. PATHWAY COMPONENTS

0061 Virtual Memories: (Ticc) These are the memories associated with pathways. They could be memory areas allocated from different physical memories in a tightly coupled distributed shared memory system or an allocated memory area in a shared memory system. Allocation of virtualMemories in shared memory systems is straightforward.

0062 (Ticc) Each virtual memory will have three components: A read-memory, R, write-memory, W, and a scratchpad memory, SP. R will contain the message to be delivered. The message in R will usually be delivered to a port-group, say G1. Parent cells of ports in the port-group will write their response messages into W. They will use SP for exchanging data among themselves while responding to the message. SP may also provide execution environments for threads used by ports in a port-group. When response message is delivered to another port-group, say G2, R and W will be switched. This will enable ports in G2 to read from their read-memory message written by ports in G1 into their write-memory.

0063 (Ticc-Ppde) In tightly coupled distributed memory environments one has to make sure that processes would always process received messages only in their local memories (read-memories R), and write messages into local memories of recipient cells (write-memories W). To make this possible, we will assume each memory module may be shared by g CPUs (cells), where g is the maximum size of a port-group. We will also assume, each CPU could directly write into designated memory areas of a limited number of other CPUs. If a cell has at most n ports, and is running on cpu_1, then it is possible that ng other CPUs might simultaneously attempt to write into the local memory of cpu_1. However, this is highly unlikely. Experimentation is necessary to determine bounds that actually occur in practice.

0064 Ticc-Ppde provides a way of interrupting parallel computations at specified parallel breakpoints. After such a break, one may examine data held in various virtual memories. This makes it possible to develop dynamic debugging facilities for parallel programs in Ticc-Ppde (Section 7.4).

0065 Agents: (Ticc) For each virtualMemory, M, agents of M are organized in a ring data-structure. By convention, signals flowing along the pathway of M will flow from one agent to its next agent on the ring in clockwise direction. We refer to this ring as clockRing, since agents on this ring clock
computation and communication events occurring around M. In the schematic representation of a pathway, we enclose M inside the clockRing of agents that surround it (see Figure 3).

Ports: (Ticc) We think of ports as belonging to cells, even though each port may have a pathway connected to it.

3.6. OPERATIONAL DETAILS

Starting and Stopping Computations: (Ticc-Ppde) Ticc-Ppde has a distinguished cell called Configurator. It is used by Ticc-Gui to set up Ticc-network, initialize virtual memories and pathways, and start parallel computations by broadcasting a message to interruptPorts of a selected subset of cells in the network. This will activate the selected cells. From then on computations will spread asynchronously over the network in a self-synchronized manner modulated by messages exchanged among cells. When parallel computations are completed each cell in the network either may itself terminate, based on some locally defined conditions, or may terminate based on an interrupt message received via its interruptPort from another cell. As a cell terminates it may send an interrupt message to the Configurator. When Configurator had received, interrupt messages from all cells that sent them, it will terminate polling it ports, transfer control to C++ main or Gui, print outputs and cause the network to be deleted, including itself.

Partitioning Resources of a Supercomputer: Ticc-Ppde could run in a shared memory supercomputer together with any other message-passing platform. Thus, one need not discard one's parallel software resources. If a supercomputer had, say N processors, then any portion of it may be assigned to running Ticc-based parallel programs, and the rest assigned to run on any other message passing platform. Ticc will have no knowledge of the processors assigned to other systems and vice versa. They will have independent resources assigned to them and could run at the same time without interference.

Developing Parallel Programs: (Ticc-Ppde) Programming a parallel processing application will consist of defining the following in C++: (i) Cell subclasses in an application, (ii) poiiPorts () method and all other methods called by poiiPorts () for each cell subclass, (iii) message subclasses used in the application, and (iv) Ticc-network. The only new task is setting up Ticc-network. This is easily done using Ticc-Gui.

(Ticc-Ppde) Efficiency with which a parallel application runs in Ticc-Ppde is crucially dependent on the Ticc-network set up for that application. Once the necessary initial network is set up, Ticc-Gui may be used to start computations in the network, and debug parallel programs dynamically using parallel breakpoints in a manner similar to using sequential breakpoints in ordinary sequential programs. During computations, the network may grow or shrink dynamically. One may also use Ticc-
Gui to dynamically update a parallel program and monitor its performance (Section 7). These simplify parallel program development and maintenance in Ticc-Ppde.

3.7. CONCLUDING REMARKS

Ticc message passing facility and Ticc-Ppde models of parallel computation provide a framework to design and implement parallel programs using cells in Ticc-networks. It has the following features: (i) pathway abstraction with built-in synchronization features that simplify writing of parallel programs; (ii) self-synchronized self-scheduled message-driven asynchronous thread execution with no user participation; (iii) parallel execution control structure that is isomorphic to message flow structure in a network of cells and pathways; (iv) low latency communications, (v) capability to simultaneously transfer practically unlimited number of messages in parallel at any time without message interference, (vi) mutual independence of threads in asynchronous polling, (vii) virtual Memory allocation to minimize memory blocking, and (viii) facilities for dynamic security enforcement, debugging and updating. These features together simplify parallel program development and maintenance, yield high execution efficiencies even at low grain sizes, and scalability. With this preamble, we will now introduce the structure and operation of Ticc and Ticc-Ppde, and illustrate their use through three simple examples. A user manual for developing parallel programs using Ticc-Ppde and Ticc-Gui is now in preparation [29]. It is not pertinent to the subject matter of this patent application, because its details are incidental to the current implementation. Implementation may change in the future while preserving all the fundamental features of Ticc-Ppde, claimed in this patent application.
4. HISTORICAL BACKGROUND (SECTION NUMBERS CONTINUE WITH SUMMARY)

0072 Dichotomy between Computation and Communication: We carry a historical burden. There are no integrated conceptualizations of communication and computation: Communication is not a part of program specification in our theoretical models of programming, which are based on three primitives assignments, if-then-else statements, and while statements, and conventions for program control \([1,2,3,4,5,6,7]\). They do not provide input/output or message/passing primitives. It is common to view communication, as a necessary evil one has to suffer in order to do computations.

0073 Turing machines [8, 10] provide a theoretical model of sequential computations. It provides a definitive definition of what a sequential computation is. It is possible to write a universal Turing machine simulator and use it to run compiled Turing machine programs. PRAM [35] models are good for analysis of parallel programs, as also multi-tape Turing machines [10]. They do not provide a complete model of parallel computations since they ignore synchronization and coordination by assuming a single universal clock, \(\pi\)-calculus \([42, 43, 44, 45]\) provides a comprehensive model of concurrent computations, where interactions among independent units are the basis for all computations. It is, however weak on synchronization and abstractions needed for easy programming. We will say more on this in Section 5.1.

0074 Lack of Synchrony in Software: Reasons for this dichotomy are quite simple: For communication to occur, receivers should listen to senders at right times and fully absorb messages. This requires a certain synchrony. Such synchrony does not naturally manifest among parallel processes or among interrupt driven concurrent processes. Consequent gap is bridged by using protocols, synchronization sessions, buffers holding message queues, and by programmed punctuated data exchange sessions among parallel processes. These add to latencies.

0075 Synchrony in Hardware: There is no such dichotomy in hardware. Communication, synchronization and coordination are all a part of every connected pair of hardware components. Clock pulses enforce synchrony in synchronous hardware circuits. Start and completion signals between connected hardware units enforce synchrony in asynchronous hardware circuits. Thus, programs rely on hardware circuits to perform communications, invoking operating system intervention within programming systems to use hardware at right times in the right manner. This requires synchronization sessions and use of buffers with message queues. Consequent software complexities that add to latency are hidden from users.

0076 Bottlenecks: This gives rise to the first two of three bottlenecks we face in parallel programming technology: (i) Communication Bottleneck: This is caused by high communication
latencies and inability to cater to communication needs of parallel processes in a timely manner.  

(ii) Debugging Bottleneck: Caused by lack of tools to dynamically debug parallel and concurrent processes.  

(iii) Memory Bottleneck: Data cannot be fetched from memory at rates adequate to feed the parallel processes. This is caused by memory bandwidth limitations and memory blocking.  

0077 Ticc eliminates the first bottleneck above (Sections 5) and Ticc-Ppde eliminates the second one (Section 5). The two together can help eliminate the third bottleneck through appropriate allocation of virtual memories and organization of messages (Section 3.5).  

5.0. MESSAGE PASSING IN TICC  

5.1. Ticc, MPI, CSP, π-calculus  

0078 MPI: Unlike MPI, Ticc is a connection oriented communication system. A message can be sent only if there is a pathway connecting senders and receivers. A cell may establish a pathway between two ports only if it had the appropriate privilege to do so. Privileges are used in Ticc-Ppde to enforce application dependent security. We have already discussed differences between MPI and Ticc. Let us now briefly consider how Ticc differs from CSP.  

0079 CSP: CSP [34] is also a connection oriented communication system. All communications in CSP are synchronous in the sense of Ticc. User may-skip waiting for a message by using guard statements. CSP has its own pathways for exchanging messages. However, pathways in CSP are implicit. They do not have an explicitly defined structure. They are built into the processes that exchange messages. They do not provide a level of abstraction that decouples data exchange details from network connectivity or computations performed by processes. Thus, they cannot be dynamically changed or updated. Introducing or removing a pathway would require program rewriting. Most importantly, pathways do not carry with them execution environments to process received data. Methods used to process data are built into the sending and receiving processes. CSP is not used in parallel programming, although there are parallel programming languages based on CSP [38]. It is used mostly in operating systems.  

0080 π-calculus: This specifies the mathematical foundations of a framework [42, 43, 44, 45] for describing many types parallel and concurrent process interactions, and indeed defines parallel computations definitively. As mentioned earlier, it is weak on issues of synchronization, coordination and abstractions. It does not provide explicit controls for synchronization. Applications of the ideas in π-calculus to practical parallel programming methodologies have not emerged yet. Some structural and operational components of Ticc-Ppde, such as (i) dynamically changeable connection oriented communication, (ii) automatic process activation based on message exchange events and (iii) local and
remote pathways and memory environments of Ticc-Ppde overlap with those used in fl-calculus. Property (iii) in Ticc-Ppde follows from use of virtual memories and component encapsulation in Ticc-Ppde (Section 7.6). Pathways and memories of encapsulated components will not be accessible to parts of network that are outside the encapsulation. This is similar to use of restricted names in Pi-calculus.

0081 We will now proceed to describe the Ticc communication system and network models of parallel computation that they naturally give rise to.

0082 A Note of Caution: The infrastructure described below might seem quite formidable at first reading. It should, however be noted, the concepts are very easy to implement. Ticc and Ticc-Ppde prototypes were implemented in C++ by one person (this author) in two and a half years.

5.2. CAUSAL COMMUNICATION PRIMITIVES (CCP’S) AND PATHWAYS

0083 Ccp: (Ticc & Ticc-Ppde) We add a new kind of programming primitive to programming language, besides assignment, if-then-else and wft/fe-statements. It is called Causal Communication Primitive, Ccp. It has the form "X: x → Y," where X is the context (signal sender) of the Ccp, x is a one or two bit control signal and Y is the signal recipient. X can be a Cell, a Port, or an Agent. The same holds for Y. There are six versions of Ccp:

\[\begin{align*}
\text{i)} & \quad \text{cell: } c \rightarrow \text{port;} && \text{port should be tuned to cell.} \\
\text{ii)} & \quad \text{port: } c \rightarrow \text{agent;} && \text{agent should be tuned to port.} \\
\text{iii)} & \quad \text{agenti: } s \rightarrow \text{agent2;} && \text{may send } s \text{ to itself,} \\
\text{iv)} & \quad \text{agent: } s \rightarrow \text{port;} && \text{port should be tuned to agent,} \\
\text{v)} & \quad \text{agent: } s \rightarrow [P_1, ..., P_k]; && \text{agent to a group of ports.} \\
\text{vi)} & \quad \text{port: } s \rightarrow \text{cell;} && \text{cell should be tuned to port,}
\end{align*}\]

where \( c \) is a completion signal and \( s \) is a start signal. Ccp is similar to assignment in that it sets values of signals associated with cells, ports and agents. Whereas the effect of an assignment action is immediate, the effect of Ccp is not immediate. It causes certain things to happen. A sequence of Ccp’s when evaluated, will cause signals to travel along a pathway (Figure 3) and this will eventually cause a message to be delivered to recipient cells.

0084 Structure of Pathways: (Ticc) Pathways have a rather complex structure. Figure 3 illustrates a simple pathway connecting two ports \( P_1 \) and \( P_2 \) of cells \( C_1 \) and \( C_2 \), respectively, and containing two agents \( A_1 \) and \( A_2 \) on the clockRing that surrounds a virtual memory \( M \). \( A_1 \) and \( A_2 \) are connected to \( P_1 \) and \( P_2 \), respectively, by watchRings. Cells, ports, agents, clockRings,

\(^{1}\)We have changed the format of Ccp in Ticc-Ppde from the one used in Ticc.
virtualMemories and watchRings are all C++ classes with data and methods defined for them. Each Ccp is compiled and executed over these C++ classes, in the same manner as any other programming statement is compiled and executed over a priori defined data structures and methods, without invoking the assistance of an operating system.

0085 (Ticc) The Ccp-sequence [1] in Figure 3 is associated with generalPort P1 through which the message will be sent, and the sequence [2] is similarly associated with functionPort P2 through which the reply will be sent. We will use CcpSeq(P1) and CcpSeq(P2), respectively, to refer to them. Every time a message is sent from generalPort P1 to functionPort P2, signals will flow along the simple pathway in Figure 3 from P1 to P2 (dotted blue arrows) When the reply message is sent from P2 to P1, signals will flow from P2 to P1 (dotted orange arrows). A second message may be sent from the generalPort only after receiving the reply.

![Diagram](image)

**Figure 3: A Simple Pathway**

0086 (Ticc) Ticc evolved from earlier works on Harmonic Clocks [31] RESTCLK [32]. Pathway structures introduced here are similar to those introduced in [32], but signal transmission protocols used by Ccp are different from the protocols used in RESTCLK and Harmonic Clocks. Ccp protocols guarantee high-speed message delivery without message interference, and led to successful applications to parallel programming, while Harmonic Clocks and RESTCLK did not do so.

0087 (Ticc) **Control Signals:** In a Ccp of the form "X:x->Y," X and Y will have states. A signal x can be one of two types: a start or a completion signal, where each may have upto four subtypes The
three subtypes of completion signal will each specify one of three possible alternatives: (i) send: switch R and W (ii) forward: don't switch R and W or (iii) halt computations. Each subtype of start signal will specify one of four possible choices: (i) broadcast signals to ports, or post one of the following three notifications on a port, (ii) waiting-for-message, (iii) message-ready or (iv) pathway-ready.

0088 Tuning: (Ticc) in any Ccp, "X: x → Y;" Y will receive x and respond to it only if Y is in a state, in which it is expecting to receive a signal of the type of x. X and Y are said to be tuned to each other if X or Y will never fail to receive and respond to a signal sent by the other. Tuned pairs (X, Y) will thus be always listening to each other at right times. The next state to which Y transfers itself will always be such that it will be the appropriate state to respond correctly to the next signal that Y will receive.

0089 (Ticc) Tuning of successive agents around a virtual memory is enforced by the clockRing. Tuning of an agent to ports connected to it is enforced by watchRings. Proper tuning is facilitated by the fact that in successive instances of message-flow along a pathway the direction of signal flow would alternate only between two possible choices. The clockRing and watchRings on a pathway will force the state of each entity in the pathway to switch in synchrony with the expected direction of next message-flow along that pathway. This guarantees that it would be always possible to pass signals along any pathway with no need for dynamic state or type checking, or synchronization sessions. This contributes to low latency message exchanges.

0090 Semantics of Ccp: (Ticc) When a Ccp, "X: x → Y;" in a Ccp-sequence is evaluated it will cause Y to sense x, and perform the following, (a) Some book keeping logical operations (details not important here), (b) change its state, and (c) cause Y to either send an appropriate signal to the next object Z that follows Y in the pathway and then return SUCCESS, or (d) return FAILURE. If "X: x → Y;" is immediately followed by "Y: y → Z;" in a Ccp-sequence, then the second statement will be executed only if the first one returned SUCCESS. Otherwise, evaluation of all subsequent causal statements in the Ccp-sequence, after "X: x → Y;" will be abandoned.

5.3. EVALUATING CCP-SEQUENCES

0091 (Ticc-Ppde) A Ccp-sequence, CcpSeq may be evaluated by the parent cell of P, or a (Ticc) Ticc-virtualProcessor (not shown in the figure) associated with the parent cell, or a (Ticc) communications processor implemented in hardware together with CPU. As mentioned earlier, evaluation of CcpSeq(P,) will cause signals to travel along the pathway attached to P, (see Figure 3) and cause the message in the virtualMemory of the pathway to be delivered to its intended recipients. The three modes of evaluations and their characteristics are described below.
By Cell: (Ticc-Ppde) If the thread of Pi evaluates CcpSeq(P-i), then message delivery will be immediate. The thread will use "p_i->sendimmediate ()" ("forwardimmediate O") to send (forward) the message immediately\(^2\). All tests reported in Section 8 used sendimmediate (). This is the normal mode of Ccp-evaluation in most parallel programs. Terms Th(IP, m2) and Th(gP) in equations (1) and (2a) in the SUMMARY embed these operations.

By VirtualProcessor: (Jjcc) VirtualProcessor is a C++ object that is used both to execute Ccp-sequences, when necessary, and to keep data related to CPU assignments and dynamic process scheduling. Every cell will have a unique VirtualProcessor associated with it, but each VirtualProcessor may service more than one cell. A cell may delegate evaluation of a Ccp-sequence to its associated VirtualProcessor at any time, if a CPU is available to run it. Cell will use "p_i->send() O" (or "Pi->forward ()") to do this, where P_i is the port of the cell though which message is being sent. VirtualProcessor will maintain a queue of pending Ccp-sequences and evaluate them in the order they were received, in parallel with computations performed by cells. Advantage is, it will cut grain sizes of cells by 400 nanoseconds. Disadvantages are, message delivery may not be immediate and CPU overhead will increase since each VirtualProcessor will require a dedicated CPU to run it. Each VirtualProcessor may send more than 2 million messages per second.

By Communication Processor: (Jjcc) VirtualProcessor may be implemented in hardware as the communications processor of a CPU. Since each cell has a unique CPU, each cell will then have a unique communication processor as well. In this case, when a thread calls "P->send ()" (or "p->forward ()") the corresponding Ccp-sequence, CcpSeq(P), will be executed immediately by the communication processor of the cell's CPU, in parallel with computations being performed by the cell. Thus, the grain size of the cell will not increase. The number of messages that may be sent at any time will be limited only by the number of available CPUs. The communication processor hardware will require capabilities to perform logical operations on bits of a 32-bit register, simple small integer additions, and at most 128 such registers.

Using VirtualProcessor or communications processor allows cells to devote all their time only to computations. This is useful when it is necessary for cells to distribute data being received from an external source at very high speeds. Cells may distribute received data at high speeds to their destinations without having to spend time to send messages.

\(^2\) It will use "Pi->halt ()" in all cases to halt computations.
Pending-Flags: (Ticc) Each Ccp in a CcpSeq(P) is associated with a pending-flag. This flag will always be set to true, before evaluation of CcpSeq(P) begins. It would be reset to false only after the message associated with CcpSeq(P) had been delivered, or evaluation of CcpSeq(P) was abandoned. Pathways or cells will be dynamically changed only if all of its associated pending-flags are false. We will later see how these are used to facilitate dynamic updating (Section 7.3).

5.4. COMPOUND PATHWAYS

Structure: (Ticc) In a compound pathway, Figure 4, there may be several agents around the virtual memory of the pathway (see Section 8.2 for an example of use of a compound pathway with just one agent). Each such agent may be tuned to several ports, each port belonging to a distinct cell. Cells whose ports are thus tuned to the same agent are said to form an ordered group. Thus, cells \([C_1, C_2]\) and \([D_1, D_2]\) in Figure 4 form ordered groups. Each cell in such a group will run in parallel with other cells in the group, each in its own assigned CPU. In Figure 4, a message sent by \([C_1, C_2]\) will be delivered to \([D_1, D_2]\) and a message sent by \([D_1, D_2]\) will be delivered to cell \(C_5\). Messages will thus travel around the clockRing from one group to another in clockwise direction. CcpSeq(P) for \(i=1,2\), in Figure 4 will be,

\[
\text{CcpSeq(P)} = [C_i; x_i \rightarrow P \cup P; x \rightarrow A_1; A_1; s \rightarrow A_2; A_2; s \rightarrow [Q_i; Q_2]; [Q_1; s \rightarrow D_1; Q_2; s \rightarrow D_2]]. \tag{Eq1}
\]
Agent $A_2$ broadcasts start signal $s$ to all ports in $[Q_1, Q_2]$. In general, when a group $G_i=[C_i|1\leq i \leq m]$ with ports $[P_i|1\leq i \leq m]$ tuned to agent $A_1$ sends a message to group $G_2=[D_j|1\leq j \leq k]$ with ports $[Q_j|1\leq j \leq k]$ tuned to agent $A_2$, CcpSeq[$P_i$] for $1\leq i \leq m$ will be,

$$\text{CcpSeq}(P_0 = [C_i \times P_i \iff x_i \rightarrow A_i; A_i \rightarrow A_2; s \rightarrow \{Q_1, ..., Q_k\}; [Q_i \rightarrow D_j; ..., Q_k \rightarrow D_k]. \text{ (Eq2)}$$

Thus it may be noted, when a group with $m$ cells sends a message to a group with $k$ cells, for each port $P_i$ of the sending group, $1\leq i \leq m$, its CcpSeq($P_i$) will contain $(4+k)$ Ccp’s.

0098 **Tuning Conventions: (Ticc-Pdde)** We will say a port is tuned to a virtual memory if it is tuned to an agent on that memory, and a cell is tuned to an agent if one of its ports is tuned to the agent. No two ports of the same cell may ever be tuned to the same virtual memory. Ports tuned to the same agent should be either all generalPorts or all functionPorts, or same kind of designated ports (see Figure 1). All cells in a group will have the same message broadcast to them with in a few nanoseconds of each other. Each cell in the group may however, use different components of that message, thereby eliminating memory contention.

5.5. **TASKS PERFORMED BY AGENTS AND PORTS**

0099 **Security Checks: (Tjcc)** When a Ccp of the form "$A_i \text{ : } s \rightarrow A_2$" (third Ccp in Eq1 and Eq2 above) is evaluated in a Ccp-sequence, where $A_1$ and $A_2$ are agents, $A_2$ will begin broadcasting start signals to ports tuned to it. $A_2$ will send start signal only if the port satisfied certain a priori specified security conditions. Application system message security may thus be enforced at the lowest message passing level. Latency measurements we made ([28a,b] and Section 8.1) included security checks. If security checks are not needed, they may be turned off. This kind of security check infrastructure can play a significant role in database, business and intelligence processing parallel applications. We will not enter into details here.

00100 **Message Delivery: (Tjcc)** The port $Q_j$ for $j=1, 2$ in Figure 3 will perform message driven cell activation when "$A_2 \text{ : } s \rightarrow [Q_i, Q_2]$:" is evaluated (see Eq1), i.e. when start signal broadcast by $A_2$ is received by ports $Q_j$ for $j=1, 2$. When $Q_j$ receives a start signal, $Q_j$ will post a message-ready signal on itself. We will refer to this posting as message delivery.

00101 **Enforcing Agreement Protocol: (Ticc)** Suppose $m$ cells, $G_i=[C_i|1\leq i \leq m]$, in an ordered group with ports $[P_i|1\leq i \leq m]$ tuned to agent $A_1$ send a message to $k$ cells in a receiving group, $G_2$. Since cells in such groups operate in parallel, each cell $C_i$ in $G_i$ will evaluate its CcpSeq($P_i$) in parallel with other cells in $G_1$, when it sends out its message. In each CcpSeq($P_i$) the second Ccp has the form "$P_i: X_i \rightarrow A_i$;" (see Eq2) where $X_i$ is a subtype of completion signal sent by port $P_i$ to agent $A_i$. Each cell in
G₁ will check completion signals received by agent A₁. This check is called agreement protocol check. It will perform this check in parallel with other cells in G₁, while it evaluates \( \text{PJ: } x_i \rightarrow A_i \text{; i.e., when } A_i \text{ receives completion signal from } P_i \).

00102 (Ticc) Agreement protocol check has two parts to it: we will refer to them as AP1 and AP2. AP1: For all \( i, 1 \leq i \leq m, (X_i > 0) \), where \( X_i \) is the completion signal sent by port \( P_i \) to \( A_i, x_i > 0 \). This will hold true only if \( A_i \) had received a completion signal from \( P_i \). While evaluating \( \text{PJ: } x_i \rightarrow A_i \text{; in CcpSeq(P)} \), each \( C_j \) will first check for satisfaction of AP1, namely whether \( A_i \) has received completion signals from all cells in the group. It would return FAILURE if AP1 was false at the time it was evaluated. Once FAILURE was returned, of course, \( Q \) would abandon evaluation of all subsequent Ccp's in CcpSeq(P) as per semantics of Ccp, and proceed to poll its next port.

00103 (Ticc-Ppde) Thread-lock associated with AP1 checking will make sure that only one cell, say cell \( C_j \) for some \( j, C_j \) in \([C_j | 1 \leq i \leq m] \), will succeed in AP1 testing. Let \( P_j \) be the port of \( C_j, P_j \) in \([P] | 1 \leq i \leq m \). Let us now suppose that it takes on the average \( t \) nanoseconds of time to evaluate a Ccp. Each \( C_j \) would have to evaluate almost two Ccp's in CcpSeq(P j) in order to check AP1 and return FAILURE. The \((m-1)\) cells in \( G_1 \) that failed in AP1 testing would have thus together spent less than at most \((m-1)2t \) nanoseconds, since they worked in parallel. The winner, \( C_j \), will then do AP2 checking. All \( C_i \neq C_j \) may immediately proceed to their respective next ports.

00104 (Ticc & Ticc-Ppde) AP2 is defined by, \( \text{AP2 = } B[X_i | 1 \leq i \leq m] \), where \( B \) is a Boolean condition on subtypes of completion signals, \( x_i \), received by \( A_i \). Condition \( B \) checks for a priori defined compatibility conditions on completion signals. Details are not important here. If AP2 test succeeded, then \( C_j \) will continue with evaluation of all \((k+4)\) Ccp's in CcpSeq(P j) (see Eq2), where \( k \) is the number of cells in the receiving group \( G_2 \), and cause a new message to be sent, or old message to be forwarded, or computations to be halted, as the case may be, depending on subtypes of received completion signals. It will spend a total time of \([ (k+4) ] \) nanoseconds to evaluate CcpSeq(P j). In all cases message will be delivered or forwarded exactly once, if computations are not halted. Message in the read-memory \( R \) will always be protected until all cells that received the message had fully responded to it. If AP2 test failed then an error condition will be generated and no message will be delivered. It may be noted, cells in a sending group, like group \( G_1 \), may always use their scratchpad memory to coordinate completion signals they send to agent, like agent \( A_j \), and thus avoid AP2 test failure. Total time spent to deliver a message from \( m \) sending cells to \( k \) recipient cells will be less than

\[^{3}\text{There are differences in the way AP2 is used in Ticc and Ticc-Ppde.} \]
at most \([(m-1)2f +(k+4)r]+kcf]\) nanoseconds, where \(d\) nanoseconds is the time taken by each receiving port to deliver the message to its parent.

In a 1.5 Gigahertz/sec CPU, \(t\) is of the order of 78.8 nanoseconds, and \(d\) is of the order of 2 nanoseconds when there are no cell activations involved. Thus the time for message delivery, while the cells are already running, will be at most \([(2/77+2+k)78.8 +2k]\) nanoseconds. In the latency test described in Section 8.1 \(m = k = 1\) was true, and the latency for 0-byte message was 396 nanoseconds. The above figures are based on this.

**Delivery Self-Synchronization:** (Ticc-Ppde) This is done when a Ccp of the form "\(A_i: s \rightarrow A_2\)" is evaluated, where \(A_1\) and \(A_2\) are agents (third Ccp in Eq2). It will cause start signals to be broadcast to ports tuned to agent \(A_2\). There are two levels of self-synchronization in Ticc during message delivery, with increasing costs in time. In the first level, when an agent broadcasts start signals to ports in a receiving group, the ports in the group will post *message-ready* postings on themselves with in \(kcf\) nanoseconds of each other, where \(k\) is the number of cells in the receiving group. We refer to this as Level-1 synchronization. Each cell in the receiving group will receive and process the message at the time it polls the port to which the message was delivered. Here it is quite possible that a cell in the receiving group started to process the delivered message before *message-ready* notifications had been posted on all ports in the group.

In Level-2 synchronization, cells in a group may begin processing delivered message only after *message-ready* notifications had been posted on all ports in the receiving group. In this case, ports in the receiving group would all get their respective *message-ready* notifications with in \(n\) nanoseconds of each other. In normal mode of operations, only level-1 synchronization is used.

**Send Self-Synchronization:** (Ticc-Ppde) Level-3 synchronization pertains to messages sent out by cells in a group. When cell \(C_i\) in a group uses "\(Pi->sendi \_mediate()\)" or "\(Pi->\_forwarding()\)" \(C_i\) will execute CcpSeq(\(Pi\)) using CPU assigned to it, in parallel with other cells in the group. However, execution of the CcpSeq(\(Pi\)) will succeed only if AP1 described above is satisfied. Otherwise, in Level-1 synchronization, \(C_j\) will abandon CcpSeq(\(Pi\)) execution and may proceed immediately to poll its next port. Level-3 synchronization will guarantee that no cell in a group would proceed to poll its next port until exactly one of them had succeeded in AP1 testing and has delivered message to the receiving group. This mode of synchronization is useful while running a Ticc-network in the *debug mode* (Section 7.2).
These facilities make it possible to run parallel programs with automatic self-synchronized asynchronous execution with high efficiencies, fully exploiting the available high-speed communications with guaranteed message delivery.

6. Ticc MODELS OF COMPUTATION

**Single Group Restriction:** Please refer to Figures 3 and 4. At any given time, only one group of cells around the virtual memory of a pathway may be active responding to a message received from that virtual memory. This is a very important restriction; we will refer to this as the single group restriction. Since each cell around a compound pathway runs in its own distinct CPU, in parallel with other cells, while cells in one group are responding to a message received from the virtual memory, other cells in the compound pathway outside this group may service their other ports not tuned to the same virtual memory. Since only cells in one group will be accessing and updating virtualMemory at any given time, one may suitably organize data in virtualMemory, and allocate virtualMemories themselves in a manner that minimizes memory contention and memory blocking.

6.1. MODEL OF SEQUENTIAL COMPUTATIONS

**Ticc-Sequential Computation:** Sequential computation in Ticc will migrate from one group to its next around a virtualMemory in clockwise direction, synchronized by message-receipts. Computations will continue indefinitely until they are stopped by one of the groups around the virtualMemory. Even though all cells around the memory run in parallel independently, each in its own CPU, computations migrating around the virtual memory will be sequential. This migration is clocked by the clockRing as one group completes its computations and sends message to its next group; hence, the name clockRing. This is the model of Ticc-sequential computations. Configurator may be used to start such sequential computations by initializing the read-memory R of a compound pathway and injecting a start signal into one of the agents on the virtual memory that is tuned to functionPorts. This will activate all cells tuned to that agent and begin computations around the virtualMemory.

**Buffer-free Communication:** In such a sequential computation each group will receive its next message only after computations had migrated through all groups. Thus, no group will receive a second message while it is still working on its first one. Hence, there is no need for the virtual memory to hold more than one message at a time. This is a consequence of the single group restriction. We call this buffer-free communication because not only are there no message queues, but also virtual memories play a role in providing execution environments for methods used to respond to messages. Messages are never copied, unless copying was forced by computations performed on messages.
Structure of Parallel Computations: (Ticc) Figure 5 illustrates the model of Ticc-parallel computations. It consists of a collection of compound pathways each running its own Ticc-sequential computation. Each compound pathway in the network will communicate with another by using specialized cells, called collator cells. It is the job of collator cells to receive data from different compound pathways, collate them, format them and send them to groups of cells in one or more of the pathways that are connected to it. Collator cells will do this at each step only when all needed data are ready and are properly collated. Collator cells will not contain any memory. They will instead use the virtual memories of pathways connected to them.

6.2. MODEL OF PARALLEL COMPUTATIONS

Collator Cell

Figure 5: Model of Parallel Computation

(Ticc-Ppde) In Ticc, poiiports do not have threads associated with them. Ticc-Ppde associates threads with poiiPorts and redefines parallel computations in terms of these threads.

Buffer-free Parallel Processing: (Ticc) Since parallel computations are defined by (i) a collection of inter-communicating compound pathways, (ii) computations in every compound pathway are buffer-free and (iii) collator cells do not contain any memory, one may conclude that all Ticc based parallel computations will always be buffer-free in the sense defined above.
6.3. INHERENTLY PARALLEL (CONCURRENT) INTERACTIONS IN TICC-PPDE

Robin Milner's Turing commemorative lecture [41] eloquently articulates the need for transition from sequential computations to concurrent interactions. fl-calculus provides the basis for such a transition. This is what formalisms like OCCAM [38], Petri Nets [39], CCS [44] have successfully done to varying degrees of elegance, generality and practicality. Fl-calculus unifies them. It is instructive to examine the role abstractions played in this evolution.

In assembly language programs control structure is explicit. This is true also in high-level parallel languages like Petri Nets [39] and Parallel Fortran [40]. Descriptions of interactions in Fl-calculus are similar in many ways to assembly language descriptions of computations. Control structure of parallel computations is explicit and could be non-deterministic. Layers of abstractions might be necessary before π-calculus is reduced to a practical parallel programming framework. It is possible that one could define useful fl-calculus abstractions in the bigraph [45] model.

In high-level sequential programming languages, control structure is implicit, driven by the semantics of programming language statements, like if-then-else, for, while statements and function invocation statements. As Milner points out, [41] object oriented languages took this abstraction-one level higher and began to shift focus to interactions, instead of operations. In high-level sequential programming languages, user focuses only on the semantics of activities to be specified, not on the control structure of how they interact. This makes sequential programs easier to write, more readable and understandable.

OCCAM [38] provides abstractions that help make some concurrent control structures implicit and dynamically dependent on actions performed by objects. However, computation, message passing and pathways are inextricably intertwined with each other. No abstraction decouples pathway and message details from message transfer and computations. In addition, operators are needed for dynamic activation and termination of parallel (concurrent) processes.

In Ticc-Ppde, control structure of parallel program interactions is implicit, just as in high-level sequential programming languages. Ticc-Ppde naturally extends the sequential object oriented paradigm to parallel computations. The construct used in Ticc-Ppde for implicit specification of process interaction is "sendimmediate ()". But, sendimmediate O just sends a message. This naturally merges with the semantics of activities performed by a cell. It does not look like a construct intended for process activation and process control.
As mentioned earlier, dynamic control structure of process activations and process interactions in Ticc-Ppde networks are isomorphic to dynamic message flow structure. All parallel process activations and interactions are driven by message exchange events. User who writes a parallel program in Ticc-Ppde has to focus only on the semantics of activities performed by a cell, not on the control structure of how they interact with other cells. This makes Ticc-Ppde parallel programs easier to write, and easier to read and understand.

In all cases, receipt of messages will automatically coordinate computations, synchronize them when necessary and activate them. No user intervention is necessary. This is similar to data-flow activated asynchronous hardware systems.

6.4. ADAPTING TO DISTRIBUTED MEMORY SUPERCOMPUTERS

Degree of Sharing: (Ticc-Ppde) We have assumed that $g$ is the maximum port-group size. We will refer to $g$ as the degree of memory sharing, because ports belonging to a port-group should be able to read messages delivered to them from a shared read-memory. Let the maximum number of ports a cell may have be $n$. We will refer to this as the degree of cross memory writing, because $n$ together with $g$ will determine an upper bound on the number of distinct groups that should have the capability to write into a shared memory not belonging to those groups.

Cross Memory Writing: (Ticc-Ppde) A cell $C$ with $n$ ports may have $n$ different pathways connected to it. Each one of these pathways may have a port-group of $g$ ports connected to it at its other end. Parent cells of these $ng$ ports would each run in its own distinct dedicated CPU. Thus, at most $ng$ different CPUs could potentially attempt to write into the local shared memory of $C$. This is an extremely large upper bound not likely to be ever reached in any parallel computation. One has to experiment with systems and programs to get representative values.

Complexity of memory interconnects for a distributed shared-memory Ticc supercomputer will depend on values of $g$ and $n$. We think, practical systems could be built with $ng = 100$. The point to note here is, memory organization for supercomputer designs for Ticc are likely to be different from the ones that are currently used in supercomputers. This problem needs further study.
7. IMPLEMENTATION AND TICC-GUI IN TICC-PPDE

7.1. CLASSES

Ticc-Ppde provides a Ticc-Gui to build Ticc-networks, start and run parallel programs, and debug and modify them as needed. The last two are still under design and development. All diagrams shown in this paper follow the Ticc-Gui format. The implementation consists of following classes: (1) **Cell** (Units of parallel computation) with subclasses, **Configurator** (Used to set up Ticc-network and modify them), **Csm** (Performs network related services to Cells), **Collator** (Collects and distributes data), and **Monitor** (Monitors activities in Ticc-network). (2) **CellFactory** (Defines and installs Cells with specified number of ports, port characteristics, and their security and privilege specifications); (3) **Port** (Allows cells to communicate with other cells and access virtual memories); (4) **ClockRings** (Encapsulates VirtualMemories, tunes agents) (5) **Agent** (Installed on ClockRings as needed. Collects and distributes signals, checks agreement protocols and synchronizes message delivery to cells in groups); **ImAgent** (Input Monitor Agent) and **OmAgent** (Output Monitor Agent) are subclasses of **Agent**; (6) **WatchRing** (Connects Agents on ClockRings to Ports on Cells. Enforces tuning); (7) **VirtualMemory** (Holds message and supports computations on the message); (8) **Message** (Encapsulates Data in VirtualMemories); (9) **VirtualMachine** (Used for message passing and book keeping) with subclass, **HealthDoctor**. HealthDoctor is used to monitor performance at ports, detect malfunctions and initiate self-repair. Ticc-provides the infrastructure for this by using the HealthDoctor to check times taken by ports to respond to messages against nominal ranges of times specified a priori for each port in a Ticc-network (analogous to checking pulse). Research and experimentation are necessary to learn how this facility may be used effectively.

7.2. API & GUI IN TICC-PPDE

API provides commands with suitable arguments to build and modify Ticc networks. Networks are built by installing cells (Figure 1), simple pathways (Figure 3) and probes (Figures 6a through 6c). Compound pathways are built by attaching probes to simple pathways as needed. We will refer to installed items as network components. Ticc-Gui provides convenient user interaction facilities to invoke methods in API, install components, and display them on Gui screen as soon as they are installed. API commands are briefly described below and illustrated in Figures 3 through 7.

---

4 Ticc-Gui was implemented by Kenson O'Donald, Manpreet S. Chahal and Rajesh S. Khumanthem, according to specifications given by this inventor.
00128 (Ticc-Ppde) **Commands in API:** 

*InstallCell:* Installs a cell of a specified subclass using CellFactory.  
*InstallPathway:* Installs a simple pathway shown in Figure 3, with given memory sizes.  
*Install Probe:* Probe is a cell with a watchRing attached to one of its ports. It is installed by connecting the watchRing to a specified agent (Figure 6a).  
*InstallCrProbe:* (Figure 6b). Cr-Probe is a Probe with an Agent attached to the free end of its watchRing; installs Cr-Probe on a clockRing at a specified place.  
*InstallMonProbe:* A monitor probe is a probe with a Monitor instead of a Cell. It is attached to an agent as shown in Figure 6a and is used to introduce breakpoints in parallel computations as explained later below.  
*InstallInMonProbe:* IM-Probe is an Input Monitor probe. It is like a CR-probe with an ImAgent, instead of regular Agent. It is attached to a watchRing near the port end of watchRing as shown in Figure 6c. It is used to trap data flowing into port and dynamically examine or modify them before they are given to the port.  
*InstallOutMonProbe:* OM-Probe is an Output Monitor probe. Like an IM-probe but with an OmAgent instead of an ImAgent. It is attached to a watchRing near the Agent end of watchRing as shown in Figure 6c and is used to trap data flowing out of port and dynamically examine or modify them before sending them out.  

00129 (Ticc-Ppde) One can browse through a Ticc-network using Ticc-Gui. After creating a network, it can be saved and reloaded at a later time when needed. Cells in a network may be programmed to dynamically install or remove any network component with out disturbing ongoing parallel computations.  

00130 (Ticc-Ppde) There are several other commands in API that are used in Ticc parallel program specification. We encountered some of them like, messageReadyo, poiiPorts (I), etc., in our discussions earlier. A complete list of all API commands may be found in the Ticc-Ppde user manual [29] (in preparation).  

7.3. **DYNAMIC UPDATING**  

00131 (Ticc-Ppde) **Pending & Agent Flags:** Two facilities in Ticc-Ppde make it possible to dynamically change pathways and cells without interfering with ongoing computations. One is the pending-flags facility mentioned in Section 5.3. The other is the agent-flag used with every agent. An agent will temporarily suspend its operations if its agent-flag is false and resume it only when it becomes true.
If a pending-flag is true it indicates that a message is due to arrive at some ports. Clearly, if an update would affect the flow of this message to those ports then it should be not be done before message delivery. Update could be done only after the pending-flags associated with those ports all become false. As pending-flags that interfere with a given update become false, one may temporarily block new messages from arriving at or being sent out from affected ports by setting the agent-flags to false for agents tuned to those ports. This will temporarily block traffic in affected pathways, thus allowing the updates to be done. By resetting the agent-flags to true after updates are done one may cause normal operations to resume.

Pending-flags and agent-flags are thus used to suitably modulate updating processes so that updating does not interfere with ongoing computations. This becomes possible in Ticc only because Ticc is self-scheduling and self-synchronizing. When message traffic is blocked in certain portions of a parallel computation network, other portions will automatically adjust their activities, by either slowing down or waiting for normal operations to resume.

Facilities for this kind of updating are built in features of Ticc-Ppde. Pending-flags and agent-flags are automatically checked before every installation of a network component at
any time. Thus, this kind of checking is not something that an application programmer should articulate. There is no need for an application programmer to anticipate and provide special facilities into an application program to accommodate updating contingencies that might be encountered during the lifetime of an application.

7.4. DYNAMIC DEBUGGING IN TICC-PPDE

00135 (Ticc-Ppde) Parallel Breakpoints: In the case of monitor probe shown in Figure 6a there is a special situation. Here the monitor cell of the monitor probe will join the group, say group G that is already tuned to the agent to which the monitor probe is attached. Thus, in each cycle of computation messages written by cells in G into virtual memory of the agent will be sent to the next group only after the monitor cell also has sent its completion signal to the agent. One may cause the monitor cell to do this only when it receives an appropriate trigger signal via its interrupt port. This trigger input may be controlled externally using a "mouse click". Until the trigger is issued, further computations performed by cells in the group may be halted using Level3 synchronization (Section 5.3). Since parallel computations in Ticc-Ppde are self synchronizing and are message driven, when computations in one group is halted or delayed the rest of the network would adjust to it automatically in the appropriate manner.

00136 (Ticc-Ppde) Thus, one may use monitor probes to introduce parallel breakpoints simultaneously at several points in a Ticc-network where agents are attached to virtual memories. Each monitor cell will run in its own assigned CPU, in parallel with all other cells in a network. We are now designing and developing a dynamic parallel debugging facility for parallel programs using this feature.

7.5. IN SITU TESTING AND DYNAMIC EVOLUTION IN TICC-PPDE

00137 Dynamic Evolution: (Ticc) A useful application of dynamic monitoring in Ticc is in situ testing of new versions of a cell in the same network context in which the old version works, without interfering with ongoing computations. This facility is useful to modify a network to meet new requirements or to correct bugs in existing code.

00138 (Ticc) The network arrangement for in situ testing is shown in Figure 7(a). Cells OLD, NEW and Checker are all tuned to the same agent and work in parallel. Thus in each cycle of
Figure 7: In Situ Testing Arrangements

computation they will all get the same inputs. OLD and NEW will write their responses into the virtual memory to which A₁ is attached. These outputs will be trapped by Checker using the OmProbes shown in the figure. Checker will check these outputs against each other and send its result to the output cell in Figure 7(a). The outputs produced by the output cell may be viewed dynamically. After sending the output, Checker will delete from the virtual memory the message written by NEW and then only send completion signal to A₁. At that point, A₁ will forward the message to the next group. Thus, the rest of the network would not even know that NEW had been installed in the network. When checking is satisfactory, the in situ network, OmProbes, OLD cell and watch ring connecting OLD to agent Ai may all be removed, thus leaving NEW in the network to take over the work of OLD. This will result in dynamic updating of the network with NEW in place of OLD.
7.6. COMPONENT BASED DEVELOPMENT

Use of component based parallel program development is illustrated in Figures 7(b) and 7(c). Figure 7(b) shows the encapsulated version of the in situ network module. This module may be used as shown in Figure 7(c) if a normalized Checker is used, whose operations are parameterized with OLD and NEW. This kind of software network module can be plugged into any network in the same way as hardware modules are plugged into larger hardware systems. Network encapsulation facilities and software module libraries have not yet been implemented in Ticc-Ppde.

We now present the parallel Latency-Test and the FFTJScalable test that were performed on parallel programs written with Ticc-Ppde.

8. Ticc-BASED PARALLEL PROGRAMS DEVELOPED IN TICC-PPDE

8.1. PARALLEL LATENCY TEST (AUGUST 2004)

Network used for parallel Latency-Test program and performance results are shown in Figure 8. Data on performance in T3D supercomputer shown in Table II was obtained from reference [27], others from [37]. We believe, the results presented here are promising, and give hope that latencies in Ticc would be much less than latencies in other systems.

Configurator was used to set up the network and start computations by sending an interrupt signal from its generalPort to the interruptPorts of cell_0 and cellM (see generalPort at the top of Configurator in Figure 8). These two cells then exchanged messages of specified length, ranging from 0 bytes to 10,000 bytes with each other for about 300,000 to 600,000 times in each execution session. Cells sent out messages from their generalPorts and received messages from other cells through their functionPorts. Each cell received and responded to messages asynchronously, i.e., it used "p-> messageReady () ;" to check for a message at port P, responded to it if there was one, or else immediately polled its next port. Every time a cell received a message, it copied the message into the virtualMemory of a pathway that connected it to the Configurator and sent it off to the Configurator. After doing this, it responded to the received message by constructing and sending a reply message to the other cell. When the Configurator received a message from a cell, it copied it and saved it in an output message vector. Thus, each message was written once and copied twice.
Each cell associated a distinct number with each message it sent, including reply messages. All exchanged messages and replies were of the same length, and each was constructed afresh every time a message or reply was sent. Latency times shown in Figure 8 included in them the times needed to construct and copy messages, and to perform security checks. Since there are three active cells in this network, at any given moment up to three messages may be exchanged in parallel.

At the end of 300,000 or 600,000 such message exchanges, total time taken was divided by the total number of messages exchanged to get the average latency. Time associated with the zero byte messages in Figure 8 is the average time needed to evaluate Ccp-sequences, CcpSeq(P) at ports.
Coding was relatively straightforward and simple. We will not enter into details here. We will present some coding details for the next example discussed in the following subsection. It may be

```
xint LT_Cell::pollPorts () {
  if (initialisation) {
    for (int i = 0; i < 2; i++) {
      /* Prepares msg and writes it into write memory MSG_LEN, a global variable, specifies
         the length of message. */
      generalPorts[i]->prepareMsg(MSG_LEN);
      /* Sends it off */
      generalPorts[i]->sendImmediate();
    }
  }
  initialization = false;
}

startTime = clock (">;
/* N_MAX is the maximum number of messages */
while (nOfMsgs < N_MAX) {
  if (functionPorts[i]->messageReady()) {
    /* Copies received msg and sends it off to configurator */
    Constructs a response message
    and replies back to the sender.
    */
    functionPorts[i]->respond(MSG_LEN);
    functionPorts[i]->sendImmediate();
  }
  prepareToTerminate();
  for (int i = 0; i < 2; i++) {
    /* Copies received msg and sends it off to configurator */
    Constructs a response message
    and replies back to the sender.
    */
    generalPorts[i]->respond(MSG_LEN);
    generalPorts[i]->sendImmediate();
  }
  nOfMsgs++;
} // End of while statement
prepareToTerminate();
interruptPort.sendImmediate();
endTime = clock();
return 0;
```

Table III: PollPorts 0 of LT_Cell.

Table III: PollPorts 0 of LT_Cell.

noted this Latency-Test program is not scalable, because the number of messages that may be exchanged at any given moment is limited by the rate at which Configurator could save messages. In order to make this scalable, each cell should be made to save its messages in its own separate output vector.
Poiiports for latency test cell, LT_cell, is shown in Table III. It is self-explanatory. Cell_0 and CellM in Figure 8 are instances of LT_Cell. Configurator saves messages forwarded to it and acknowledges receipt. poiiPorts () for the configurator is not shown here.

8.2. PARALLEL NON-SCALABLE FFT BENCHMARK (JUNE 2005)

Two networks for the FFT test are shown in Figures 9 and 10. We used both these networks to perform complex double precision 1D FFT computations [36]. Each FFT computation had S sample point inputs, for 64 ≤ S ≤ 4096. For a given S, one thousand FFT computations were done in each run of FFT parallel program, each on a distinct set of S sample points. Maximum of the power spectra for each FFT computation was printed out at the end together with timing statistics. Our multiprocessor had only 4 CPUs. Therefore, we used only 4 cells in our FFT computations.

In each FFT computation the sample points were distributed equally among four cells. For S sample points, the FFT computation consisted of \( \log_2(S) \) levels. At level zero, each cell did its computation on its share of S/4 input sample points. Thereafter at each level, \( 1 \leq L \leq \log_2(S) \) each cell did its computations on results obtained at level \((L-1)\) by itself and another cell as per rules of FFT computation (see [36]).

The configurator in Figure 9 will start computations by broadcasting a message to the interrupt ports of all the four cells. The pathway for this broadcast is shown at the bottom of Figure 9. Self-loop pathway is shown at the top of the figure. Self-loop pathway is a special case of a compound pathway where there is only one agent.
One may be notice in Table IV, for levels $1 \leq i < N/2$ (where $N$ is the number of processors), at the end of each level of computation, each cell sends a message through the self-loop using its `generalPorts` fi]. The agent on the self-loop will automatically synchronize messages sent by the four cells and make a synchronized message delivery back to trie same four cells (Section 5.5) When the message is received, each cell will pick up its share of data in the message as per rules of data exchange in FFT [36]. This will start computations in the four cells at the next level at nearly the same time (at most 8 Nanoseconds of each other). Only Level-1 synchronization was used.

Table IV: PollPorts for non-Scalable FFT.

```c
mt FFT_CellO : pollPorts ()
{
    if (initialization)
        { mstallS bCells (); initialization = false ; }
    nOfCycles = 0;
    startTime = clock ();
    while (nOfCycles < N_OF_FFTS) {
        /*level 0 computations : Takes input s from sample po int s. */
        doInputComputations ();
        generalPorts [1] -> sendImmediate ();
        //level 1 le L < H/2 computations
        for (mt 1 = 1; 1 < N/2, 1++) {
            generalPorts [1] -> receive ();
            doLoopComputation ();
            generalPorts [1] -> sendImmediate ();
            i
            /*Level N/2 computation No Message is sent out */
            generalPorts [1] -> receive ();
            doLoopComputation ();
            doSelfComputation ();
            nOfCycles++;
        }
        /*end of while loop */
        /*msg s sent to synchronize the state of f2ndAndSaveMaxPowerO */
        generalPorts [1] -> sendImmediate ();
        //receive msg the synchronizing msg.
        generalPorts [1] -> receive ();
        findAndSaveMaxPower ();
        endTime = clock ();
        /*Informing fft_conf that computations are becoming terminated */
        InterruptPort-sendImmediate ();
        /*prepares to release the processor. prepare oRelease () 
        V prepareToTerminate () ;
        return 0;
    }
}
```

Table IV: PollPortsO for the non-Scalable FFT.

Starting at level $(N/2 +1)$ through level $(L-1)$ each cell will have in its local data array all needed data to continue with the rest of FFT computations. It is not thus necessary to send messages any more via the self-loop.
00150  As the number of cells increases, synchronization delay and message delivery latency will also increase in the arrangement shown in Figure 10. In addition, since there is only one virtualMemory, memory blocking will also increase. These two factors will limit scalability.

8.3. PARALLEL SCALABLE FFT BENCHMARK (JUNE 2005)

00151  Figure 10 shows the network used for the scalable version of FFT. Here also increasing the number of cells would increase synchronization delay but, as we shall see, this synchronization is not done at every level of FFT computation. It is done only at the beginning of each new FFT computation on a new set of S sample points. Computations at successive levels of FFT computation need not be synchronized. They are automatically coordinated by messages exchanged by the cells at the end of each level. Since each cell sends out its message in parallel with other cells at each level of computation, message exchange latency will not increase. Since each cell at each level performs, its computation using a distinct local memory, memory blocking will not increase as number of cells increases. Thus, one may expect that the network in Figure 10 would be scalable. Hence the name. Its actual scalability remains yet to be tested.

00152  The network images in Figures 9 and 10 are copies of images produced by Ticc-Gui. Each network has a Configurator and four cells, cell_0 through cell_3. Each runs in its own assigned CPU. The two networks perform the same FFT computation using the same code except for different initializations and different poiPortsQ. Initializations and poiPortsQ had to be different because the networks are different. They both produced identical results for identical input sample points, because they are essentially the same. With only four cells, they both produced also identical timings, speed-
Table V: PollPorts for the Scalable FFT

```c
int FFT_Cell::pollPorts() {
    // installation
    if (installation) {
        // installation
        installSeedCells();
    }
    startTime = clock();
    // nOfCycles = current cycle number Starts at 0 N_OF_FFT = 1000
    nOfCycles = 0;
    while (nOfCycles < N_OF_FFT) {
        // synchronization step Receives acknowledgements from N/2 cells V
        for (unsigned int i = 1; i < N/2; i++) {
            // Writes and sends level-i data at functionPorts [i] */
            functionPorts[i] - > receive();
        }
        // Sends output to level 0 and saves the calculations at remaining levels
        doLevel0Computations();
        // Each cell will henceforth have its local data all information
        // needed to proceed with fft calculations at the remaining levels
        // doSelfComputations();
        for (unsigned int i = 1; i < N/2; i++) {
            functionPorts[i] - > sendImmediate();
        }
        nOfCycles++;
    }
    // Returns to while loop
    // Finds power of each spectra and saves it */
    findAndSaveMaxPower();
    // Sends termination signal to configurator
    interruptPort();
    prepareToTerminate();
    return 0;
}
```

Table V PollPorts () for Scalable FFT
ups and efficiencies, as shown in Table VI. This is an example of the kind of flexibility we mentioned in Section 3.3 paragraph 0038 of SUMMARY.

<table>
<thead>
<tr>
<th>Sample Size, S, per FFT</th>
<th>Parallel Time per FFT Micro Secs</th>
<th>Sequential Time per FFT Micro Secs</th>
<th>Speed Up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>1610</td>
<td>13040</td>
<td>8.10</td>
<td>220.50%</td>
</tr>
<tr>
<td>2048</td>
<td>750</td>
<td>3320</td>
<td>4.43</td>
<td>110.75%</td>
</tr>
<tr>
<td>1024</td>
<td>320</td>
<td>780</td>
<td>2.44</td>
<td>61.00%</td>
</tr>
<tr>
<td>512</td>
<td>180</td>
<td>300</td>
<td>1.67</td>
<td>41.28%</td>
</tr>
<tr>
<td>256</td>
<td>90</td>
<td>110</td>
<td>1.22</td>
<td>30.50%</td>
</tr>
<tr>
<td>128</td>
<td>50</td>
<td>60</td>
<td>1.20</td>
<td>30.00%</td>
</tr>
<tr>
<td>64</td>
<td>30</td>
<td>30</td>
<td>1.00</td>
<td>25%</td>
</tr>
</tbody>
</table>

Table VI: Timing Statistics for FFT

**00153 Comments:** All computations in Latency-Test program were asynchronous and all computations in FFT programs were synchronous. This is because FFT calculation required coordination or synchronization at each level. All coordination and synchronization were completely automatic. User did not have to do anything to invoke coordination of synchronization. In the non-scalable version, messages were exchange by cell-groups and in the scalable version, messages were exchanged individually by each cell, in parallel with other cells. As number of cells is increased the size of the FFT problem, S, will also increase, since grain size, S/N should remain the same throughout.

9. **CONCLUDING REMARKS**

00154 We introduced two new concepts, (i) a new model of parallel programming and (ii) integrated computation and communication. These two concepts naturally gave rise to the architecture of Ticc and Ticc-Ppde that we described here. Ticc-Ppde provides the environment and methods to use Ticc for parallel program development and execution. We discussed the benefits that ensue and new capabilities that they provide. The most important of these are (i) ease of parallel program development and maintenance, (ii) high execution efficiencies and (iii) potential for scalability.

00155 We believe Ticc-Ppde may profoundly change the technology of parallel programming, making parallel programming as ubiquitous as sequential programming is today, dramatically increasing supercomputer throughputs through increased efficiencies of operation, thereby enabling
high performance computing by less expensive desk-top multiprocessors. A 32-machine shared memory multiprocessor running Ticc-Ppde can easily outperform a 128-machine cluster.

00156 Opportunities offered by Ticc-Ppde for ease of programming, dynamic debugging and updating, and potentially unlimited scalability makes Ticc an attractive choice to meet future challenges we will face with massive parallelism when nano-scale computing becomes a reality. Ticc is also likely to change the structure and organization of future multiprocessors and supercomputers, and design of operating systems.

8. REFERENCES

00157 Please see section on Application Data.
CLAiMS

What is claimed are,

1. A system, hereinafter called The System, for writing parallel programs, πA for any application, A; (i) πA consisting of programs that run in multiprocessing computer systems with two or more processors (hereinafter referred to as The Multiprocessor), (ii) software system πA composed of software classes called Cell, Port, VirtualMemory, Agent and Message, (iii) software objects being cells, ports, virtualMemory, agents and messages which are instances of their corresponding classes, installed by πA when it runs in The Multiprocessor, (iv) these instances consisting of arbitrary number of cells, each cell containing an arbitrary number of ports, ports of different cells interconnected by pathways, each pathway containing one virtual memory and an arbitrary number of agents, the collection of all such cells and pathways being called the Ticc-Application-Network, N_A, for application A (hereinafter referred to as the network, N), (v) each Cell and Message class in πA containing specifically defined application dependent software data structures and methods, (vi) each cell, c, in N being capable of performing computations in parallel with all other cells in N by exchanging messages also in parallel with other cells, via pathways connected to ports of c, (vii) each such c running in a processor of The Multiprocessor, with the following,

\[ C = \{c_i | 1 \leq i \leq n_c(N)\}, \]  

being the set of all cells in N, where \( n_c(N) \) is the total number of cells in N,

\[ \tau(Ci) = \{Pi | 1 \leq i \leq n_p(Ci)\} \]  

being the set of all ports that belong to cell Ci, where \( n_p(Ci) \) is the total number of ports in Ci, and

\[ P = \text{UNION} \{(\tau(Ci) | c \in C)\} \]  

(read right side of (3) as "union of the set of all \( \tau(c) \) such that c is a member of set C") making use of (1a) methods for installing and modifying cells, ports, agents and pathways in N;

(1b) methods for allocating virtual memories to pathways from memory areas of a hardware memory unit that is shared by all processors in a Shared Memory Multiprocessor;

(1c) methods for automatically dynamically allocating a CPU to each cell in N;

(1c) methods for allocating virtual memories to pathways from memory areas of a collection of distributed hardware memory units, where each distributed hardware memory unit is shared by a processor-group containing a limited number of processors, each processor in such a
processor-group being assigned to run a unique cell in C, whereby forming a corresponding
cell-group consisting only of cells that are run by processors in the said processor-group,
cells in each such cell-group being capable of writing into virtual memory areas of pathways
attached to ports of a limited number of neighboring cell-groups;

(1d) methods for allocating virtual memories to pathways in a manner that minimize memory
blocking and memory contention and thus contribute to scalability;

(1e) methods for dynamically installing new cells and new pathways in N, dynamically modifying
existing pathways and cells, without service interruption and without loss of data, while the
multiprocessing system is running π_A; and

(1f) methods for developing dynamic self-diagnosis and self-repair facilities for application
system, π_A.

2. Methods as recited in claim 1 further including steps for organizing and running parallel programs π_A,
defined by n > 1 sequential processes S = {S_k | 11 ≤ k ≤ n} where each S_k running in parallel with other
sequential processes in S, and all sequential processes together constituting the intended parallel
computation defined by π_A, by employing the following additional steps,

(2a) steps for cutting up each sequential process S_k into a collection of threads, Th(S_k), Th =
UNION {Th(S_k) | S_k ∈ S}, being the set of all such threads, and methods for distributing the
threads in Th among ports p ∈ P (P defined in equation (3)) in a manner such that, if Th(p) is
the set of all threads allotted to port p and for any thread t ∈ Th(p), p is the port of t, then (i)
for any cell c in C (defined in equation (1)) the set of all threads of c is Th(c) = UNION{Th(p) | p ∈ π(c)}, (π(c) defined in equation (2)), and UNION (Th(c) | c ∈ C) = Th, (ii) no cell c could by
itself perform all of the computations specified by any one sequential process, S_k ∈ S, (iv)
pairs (p, q) of ports belonging to a cell c may be mutually independent in the sense that p
would never use data generated by q and vice versa, (v) no more than one thread of c
would be active at any time performing computations, and (vi) when all threads in Th
terminate their respective computations N would have performed precisely the intended
computation of π_A;
(2b) steps for suspending and resuming computations performed by threads without loss of data and without invoking assistance from the operating system that runs The Multiprocessor;

(2c) steps for automatic asynchronous message driven activation of threads, activating the right thread belonging to a port, p of a cell c, when a message is received by cell c at port p, and once activated, allowing the thread to complete its computations even if such computations were suspended in the middle and later resumed;

(2d) steps for enabling each thread, \( t \in \text{Th} \), to send a message by itself using the pathway attached to its port, in parallel with other threads in Th without message interference and without invoking assistance from the operating system that runs The Multiprocessor;

(2e) steps for guaranteeing high-speed parallel message delivery without message interference, the number of such messages sent at any time being limited only by the number of cells in N, thereby facilitating scalability;

(2f) steps for automatic asynchronous message driven scheduling and activation of threads in such a manner that control flow of computations in N is always isomorphic to message flow, thereby enabling parallel program development without specifying methods for process activation, synchronization and coordination;

(2g) steps for different levels of synchronization thread activations and data distributions, with increasing precision; and

(2h) steps for automatic enforcement of application system security and privilege specifications at the time of message delivery, at the time of cell or pathway installation, or at the time of dynamic reconfiguration of network N.

3. **Methods and steps as recited in claims 1 and 2 further including,**

(3a) steps for starting and stopping a parallel program;

(3b) steps for specifying parallel breakpoints in N to temporarily suspend parallel computations at the specified breakpoints and examine data in various virtual memories, in order to dynamically debug a parallel program;
(3c) steps for dynamically testing new versions of cells in \( N \), in parallel with old versions, in the same network context in which the old version operates, and after satisfactorily completing the test to replace the old version with the new one, all without interfering with ongoing computations, thus enabling dynamic evolution of \( \pi_A \);

(3d) steps for encapsulating any well-defined network into a component, which can be plugged into any larger network containing matching port interfaces;

(3e) steps for building a library of such components, which may be downloaded and used to build new parallel programs,

(3f) steps for dynamically displaying parallel outputs while the program is running, without interfering with ongoing threads; and

(3g) steps for simplifying parallel program development through use of pathway abstraction and causal communication primitives in the programming language and elimination of need for user intervention to coordinate, schedule and synchronize parallel computations.
LEGEND:
generalPorts TO SEND OUT SERVICE REQUESTS AND RECEIVE REPLIES.
functionPorts TO RECEIVE SERVICE REQUESTS AND SEND BACK REPLIES.
interruptPort TO START, END AND INTERRUPT COMPUTATIONS
statePort TO GET AND SET THE STATE OF A CELL
diagnosisPort TO DIAGNOSE MALFUNCTIONS
comPort TO REQUEST SERVICES FROM Communications System Manager, Csm, TO DYNAMICALLY INSTALL/REMOVE CELLS AND PATHWAYS IN A NETWORK.

Figure 1: Schematic Diagram of a Cell.
Figure 2: Two Models of Parallel Processes.
Ccp-Sequence whose execution will cause C1 to deliver a message to C2
[1]. C1:c→P1; P1:c→A1; A1:s→A2; A2:s→P2; P2:s→C2;
Ccp-Sequence whose execution will cause C2 to deliver a reply to C1
[2]. C2:c→P2; P2:c→A2; A2:s→A1; A1:s→P1; P1:s→C1;

Figure 3: A Simple Pathway
Figure 4: A Compound Pathway: Model of sequential Computation.
Figure 5: Model of Parallel Computation
Figures 6: Illustrating probe attachment schemes
Figure 7: In Situ Testing Arrangements.
Figure 8: Network used for Latency Test
Figure 9: Network for Non-Scalable FFT
Figure 10: Network for Scalable FFT