

## CCA Project Overview

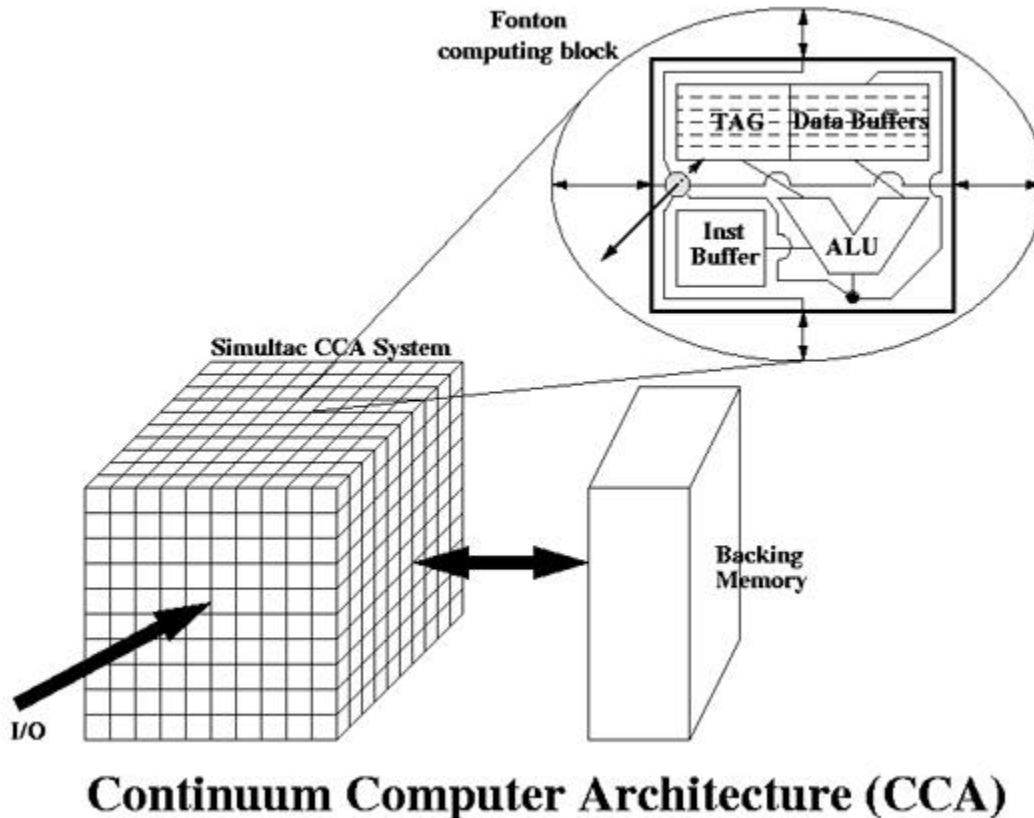
Paul C. Messina, Technical Contact

[messina@cacr.caltech.edu](mailto:messina@cacr.caltech.edu)

California Institute of Technology

Mail Code 158-79

Pasadena, CA 91125



## Introduction

Continuum Computer Architecture (CCA) is a research and development project in the domain of innovative computational models, methods, and mechanisms to radically alter the structures upon which high performance computer systems are based and to establish a new practical foundation for future ultrascale computing. CCA has been conceived to establish a new framework for the relationship between the functionality of the static physical implementation

technology and the computational demands of the dynamic execution abstraction. The CCA has been devised to exploit the opportunities afforded by the use of a single common underlying implementation technology to merge the historically separate functions of logic, communications, and storage into a single composite fine-grain physical element from which a global uniform computing medium may be realized through replication.

Investigation of continuum structures is motivated by the need to eliminate the major barriers to effective parallel computing imposed by conventional massively parallel processors (MPP) and the separation of memory from execution logic. The resulting challenges of memory access latency, parallel flow control overhead, resource utilization contention, and starvation from load imbalance are believed to demand this rethinking of high performance computing systems from the underlying semiconductor technologies through architectures, program models, and resource management strategies in order to enable orders-of-magnitude increases in performance beyond what can be projected for systems envisioned today. This project seeks to conduct the necessary initial exploratory research to devise a detailed hierarchical model for a continuum computer architecture, determine its correctness through emulation, evaluate its performance potential through detailed simulation, and assess its feasibility through design studies. If proven a viable approach, CCA will offer users a new path to achieving performance in the trans-teraflops regime towards petaflops capability in a ten year time frame.

### **Problems with Conventional Approaches**

Component density, clock speed, and parallel system structures together conspire to increase the distances between data and operation logic with a consequent negative impact on performance and ease of programming. Already, close to half the processor die area is consumed by cache just to offset this growing gap. Memory capacity is outstripping memory bandwidth with the added problem that the time to access every element within a memory chip is growing. Processor execution logic is the bottleneck resource and is in the critical path to execution performance. Increasingly complex support logic is incorporated to extract small amounts of additional performance with diminishing effect. The result is that as processor complexity has grown, performance delivered per device has steadily decreased.

Conventional parallel architecture is a poor fit to the natural evolution of parallel algorithm execution. The program counter driven flow control model of sequential execution is highly constraining. The segregated storage and processing centers require enormous data movement that is an overhead imposed merely by the fact that the logic to perform an operation is not part of the circuits used to store the operand value. Also, (and this is a subtle but key point) control hardware is designed to execute instruction streams (opcodes are

hardwired) but interpret complex data structures (data's only meaning is encoded in an instruction sequence dictating how it is used) which means data is dumb and requires the distant processor to make it useful.

Only one instruction stream can be executed at a time from a given program counter and all operand data must be acquired from a distinct storage unit (registers, the obvious but limited exception) even when buffered in cache. The bandwidth constraints to memory due to the von Neumann bottleneck combined with the increasing latency (measured in processor cycles) due to increasing clock rates and parallel system diameter together impose a severe barrier to highly parallel execution implicit in many application algorithms. While a number of techniques have been devised to reduce or hide some of the latency, even when successful, the use of load and store instructions themselves is a kind of overhead. Above that, dynamic workload balancing is also inhibited by the highly localized control and state management mechanisms centralized in the processor data paths, distinct from the memory and communications facilities.

### **Goals of a New Architecture Paradigm**

An innovative and therefore possibly speculative computational model must address the following fundamental issues so that the physical execution may closely match the demands of the abstract parallel algorithm:

1. Memory bandwidth must be dramatically increased such that all data storage is exposed and any combination of values can be used in a given cycle, independent of data distribution.
2. Transit distances between operand storage and operation logic must be minimized, in the presence of highly parallel physical structures.
3. The ratio of operation logic and operand storage must be substantially adjusted to balance logic and data access throughputs to match parallel algorithm demand for concurrent operations.
4. Execution order must be limited only by data precedence constraints and the structural attributes of compound data which are fundamental to the algorithm.
5. Flow control must be truly distributed, limited only by algorithmic parallelism and physical separation of operand and instruction information.
6. Resource allocation to abstract objects must be dynamic, seamless, and incur no critical path time penalty.

The CCA Project describes an innovative architectural concept that addresses all of these requirements to a significant degree.

### **Overview of Technical Approach**

The original inception of electronic digital computers required the separation of communication, logic, and storage units because each capability was achieved

using a distinct technology (magnetic core for storage, vacuum tubes for logic, pulse lines and transformers for communications). A new opportunity has emerged with the evolution of CMOS which permits effective implementation of all three functionalities by a single device technology (at least in large part). This has been demonstrated by Kogge, among others, in PIM (Processor In Memory). Thus, the need, if not the prejudice, for the separation of element types has been largely eliminated opening the way for a new physical structure that more closely matches the needs of parallel processing.

An atomic element called a "Continuum Computer Architecture Block" or just "CCA-block" (see diagram, top of document) is conceived that would combine logic, storage, and nearest neighbor communication. Both operand data and operation data (instructions) are treated as abstract structures distributed across large arrays (2 & 3-D) of blocks. Each CCA-block incorporates the equivalent of a few registers, a word-wide basic logic unit, interfaces to adjacent blocks, tag matching logic, and a small finite-state machine control table. All data elements, both operand and instruction data, are tagged according to type and runtime name which the CCA-blocks can detect, compare, and respond to. Abstract structures include links or pointers which tie the logical elements together across physical space. Naming is virtual and structures migrate by drifting across adjacent blocks by copying, much like a 3-D shift register. Pointers are changed as structures move through a protocol that retains pointer consistency between virtual objects.

The CCA-block's control table reflects a set of simple transformations or functions that govern how the block responds in the presence of operand or operation data with the right attributes. A block may act as a simple storage buffer, a pathway for moving structure elements, logic for updating pointer trees, perform an instruction, or a combination of these in a single cycle. All actions of a CCA-block are in response to local context.

The fundamental and innovative concept that defines the Continuum Computer Architecture approach is that together, the set of local reaction functions at the CCA-block level result in a full model of computation at the global level. This is analogous to biological symbiosis where many simple elements, cells, combine only locally to produce a complete complex entity (e.g. a human being). Thus, a large structure of such cells makes up an execution medium or continuum computer. Details of how this may be achieved are provided later.

The Continuum Computer Architecture addresses the requirements in the previous section to a significant degree:

1. Memory bandwidth is expanded as each CCA-block contains only a few registers, all of which have direct exposure to the block's external interfaces and internal operation logic.

2. Latency between every register and its nearest logic is within a single cycle, as is communication to the adjacent blocks.
3. The resources dedicated to logic are in the range of that committed to storage yielding a much higher potential processing throughput for a given system size and cost than conventional systems.
4. Two modes of execution are made possible by CCA and are mixed. One causes operand data structures to find instruction structures like vectors using virtual wormhole routing to perform array operations along a virtual pipeline of instructions. The other causes instruction data structures to "walk along" operand data structures, performing a series of operations at every site (node in the structure) and could be used on a tree for example to change or create a new tree in a data parallel fashion.
5. Flow control is completely distributed. Any block can perform any basic operation given the right data elements. Flow control is completely asynchronous and uses mechanisms similar to the "Futures" construct as well as more basic barriers for continuation.
6. Resource allocation is done continuously by objects slowly drifting back and forth within the execution medium across adjacent block boundaries. As crowding occurs, objects drift apart. As distances between structures grow, objects tend to drift together through affinity, all determined by local rules and partial exchange functions.

One particular problem with the proposed structure is that the storage capacity is less than that required by application programs executing at the envisioned speeds. Therefore, it will be necessary to embed the CCA structure in a specially devised outer memory shell that retains the structure information at high density but is passive.

## **Programming Model**

The programming model for the Continuum Computer Architecture is an asynchronous parallel paradigm based on the Futures construct. This differs from most parallel programming methodologies employed with conventional multiprocessor system architectures.

The Futures was first proposed almost two decades ago by Hewitt as part of the Actors model and incorporated as a major element of Halstead's MultiLisp environment on the Concert multiprocessor. Because of its importance to Continuum Computer Architecture, a brief description is provided. A Futures is a logical synchronization site local to a named object whose value is to be returned. If the value has been produced either by computing it or acquiring it from some other service, then referencing the Futures simply results in the value or structure pointer being returned. But if the desired object has not been produced, the Futures automatically and locally stores up the continuations of

the requesting threads. For the first request, a thread may be instantiated to derive the required value. Upon completion and the availability of the result, the list of continuations are reactivated and the new result is applied to each for their respective computations to proceed. A Futures allows other actions in which it is embedded to proceed until the actual value it is to produce is actually to be used. This is important as many operations manipulate structures instead of, say, adding two numbers attached to the structures and these kinds of activities can be overlapped, increasing parallelism and reducing critical path time to execution.

A simple example performs a sum-reduction in parallel of values at the leaf nodes of an unbalanced binary tree, using the Futures based parallel function here called `parallel-tree-sum` (please forgive the Lisp-like syntax but it does simplify writing the experimental support software):

```
(define parallel-tree-sum (binary-tree)
  (cond ((number? binary-tree) binary-tree)
        (else (add (future (parallel-tree-sum (left binary-tree)))
                    (future (parallel-tree-sum (right binary-tree)))))))
```

If the argument handed to the function is a number value, then that is what is returned. Otherwise, the argument is assumed to be an intermediate node in the binary tree. The Futures construct allows the left-hand subtree to be computed. The add operation does not execute immediately but its second argument is begun as well. Thus both halves of the tree are searched and summed simultaneously and recursively. An assignment operator can be wrapped within a Futures so that every reference to the left-side variable name will be managed by the Futures mechanism. Other more sophisticated parallel flow control structures can be devised with the same properties. While implementation of the Futures construct on conventional processor based architectures is often inefficient, the CCA implementation is a natural consequence of the underlying execution model as will be discussed below.

The programming model will also incorporate annotations related to data type, usage patterns, side-effects, and structures that will permit optimized manipulation. For example, pure value oriented transfers allow local garbage collection and expose parallelism that might be defeated by the possibility of side-effects. Vectors enable data transfers by wormhole routing and tightly coupled data layout for efficient time and space utilization. Simple fork-join operations are included when they would suffice instead of the more complex Futures. Other opportunities for optimization will be developed during the course of the research. The underlying intent is to reveal relative locality of elements within objects and of objects (data and instruction) with respect to each other. Such information can be used dynamically by the distributed execution model to keep things relatively close when desirable to reduce latency. Otherwise they can

be allowed to remain relatively distant to improve load balancing and reduce contention.

## **Execution Model**

A major objective of the CCA Project is to develop a complete and consistent low level execution model that establishes the operational relationship between the parallel programming model described above and the hardware level mechanisms capable of being supported by the CCA-blocks. Although only at its nascent stage of evolution, the execution model can be described in terms of its fundamental concepts and key attributes that distinguish it from other physical parallel architectures. The basic premise of the Continuum Computer Architecture concept is that a fully distributed data and control model can be devised such that a set of simple transformation rules at a local site can, when highly replicated, establish a global model of computation. This is justified because the semantics of conventional von Neumann based parallel computing models differs from a CCA-like distributed model in a few ways that can be replaced with local alternatives, thus retaining overall logical computation capability. Examples of control mechanisms to be changed are the use of a program counter for flow control, a stack pointer for local context, and a set of named registers for ALU operand buffering. A CCA distributed computation can replace the program counter with a linked list of instruction pointers. The local context can be established by a pointer to a structure specifying a process instantiation. And registers can be replaced by distributed and virtually named buffers (like reservation stations).

An important insight to making this possible is the realization that data objects (vectors, arrays, trees, and general graphs), procedures (representations of static streams of instructions with abstract referents to operands), and processes (active instantiations of procedures with applied data objects) can all be represented, physically distributed, and altered in position in the same general way. When procedure and data objects are aligned in the same physical space and logically related through process designations using tags, local actions can take place that modify the global program state, and direct the next actions to be performed, usually with a high degree of concurrency. The requirements are that virtual names can be mapped to physical locations, even when objects shift in place, and that object types are intrinsically self determined through tagging mechanisms including their association to specific active processes. These basic capabilities permit distributed logical structure through smart dynamic links, movement through 2-D and 3-D shift-register like mechanisms across the system, and actions through coinciding and alignment of instruction and data objects within CCA-blocks.

## **Structures and Addressing:**

Whether data, instruction, or process designators, all CCA information can be physically organized as scalars tightly bound (always physically adjacent) in small groups called complex elements, and loosely related in large compound structures through links. Values within complex elements always move together while elements comprising compound objects may have time varying physical relative positioning.

The virtual to physical translation involves links that include the abstract dynamic names (like the unique id of a thread instantiation) and hints as to where to find it. For example, two elements may be along the same column of CCA-blocks with the intervening distance changing. A link may direct a copy of an element to follow along that column until the destination object is located. This is reminiscent of the way the KSR-1 achieved cache coherence. When off column movement occurs, simple link redirectors are laid down and eventually garbage collected. Such techniques have been considered by other work including the J-Machine. When objects such as vectors are known to be of restricted form, advantage can be made through wormhole routing. When combined with a string of instructions distributed along a line of blocks, vectors can move through the sequence, having operations performed along the way. A physical set of blocks acts as a long execution pipeline while instruction and data streams are aligned and coincide. This is called "wormhole routing."

### **Procedures and Processes:**

Execution is performed by moving a structure of instructions over a structure of data. The vector example is the easiest to visualize as a string of data elements move along a string of instructions just as in a pipelined vector machine. The earlier example of a short sequence of instructions moving through a tree structure to produce the sum of its leaf node elements is more interesting. A string of instructions representing the function parallel-tree-sum is co-located with the root node of the tree in physical space through linking as described above. The instructions move across the CCA-block holding the complex element of the root node. Where the element is an intermediate node of the tree, the instruction string replicates (mitosis) and literally propagates along the left and right hand links to the next sub-trees. This continues until the leaf nodes are encountered. A subsection of the instruction string related to summation is perpetuated back up the tree (the rest of the instruction stream having been garbage collected). This is continued back up to the root node where the final sum value is delivered. Each operation was performed at the local CCA-block site where the complex element of the tree resides. The partial result values are carried along back up the tree along with the residual code segment.

Once at the root node, the dynamic link pointing to the process object that called the function originally, directs the result value back to the point where it will be

used in a continuing calculation. Alternately, it may be assigned to a global designator (a variable name) and accessed by multiple processes. Identifying the correct object, whether data or instruction, requires the static name and the dynamic process designation to be used in matching. Global data does not need a process id, but automatic or relatively local data does. Tags are created and modified dynamically and matching mechanisms are an intrinsic part of each CCA-block. The procedure is the textual description of what is to be performed while a process is an active instantiation of that procedure and defines the scope of name space in which the procedure may be applied in this particular instance. Early work by Arvind on the U-interpreter reflects the kinds of dynamic transformations necessary for this work.

### **Parallel Synchronization:**

The semantics of the Futures construct are realized using many of the same mechanisms described above. A Futures object is a linked list with a special tag. A datum that is wrapped in a Futures has a tag indicating if the value is available. If the value is available, it is returned. If the value is not available, the access request which is a continuation is stored temporarily on a list which is extended as requests are queued up for the same element. This is similar to I-structures found in the functional language Id. The linked list is built across a series of CCA-blocks connected by pointers. This structure is garbage collected when the value is returned and passed along to the requesting actions.

A forall-like operation is performed by replicating the instruction sequence of the inner loop and applying them to the respective data elements that distinguish the separate instances of the basic inner block. Such instruction sequences look like data vectors and move accordingly. Jumps and branches are achieved through links within the instruction sequence. Joins are achieved with a continuation using a counter variable, as would be expected, and barriers are just a variation of this.

Message passing and message driven computation (such as for object oriented processing) involves passing a pointer to the operands in a message structure and creating a method instantiation which is just another process.

### **Load Balancing:**

Distribution of work and objects in a continuously changing context is much easier in CCA than conventional architectures because the barrier and granularity of a move is as small as the time to shift to the neighbor blocks. Some cross block signals indicate when active information is next door. In this case, if there is room on the other side, an object will shift (drift) in that direction. When an object needs room such as when it needs to make a copy of itself (in parallel), a signal is conveyed to its neighbors to move aside. If things are tightly packed,

this signal is propagated through the adjacent objects until empty space is detected and objects begin to shift in that direction. This occurs automatically and is transparent to the user. However, not all elements should diverge in space, such as the elements comprising a data vector. Tags indicate when affinity should be maintained. This is either direct, such as the vector, or indirect through a process descriptor. The exact methods for achieving this most effectively is still to be determined.

The memory shell surrounding the CCA core structure will complicate the load balancing process because objects must transition from active within the core to suspended in the memory shell and back in a user transparent and performance indifferent way. It is an objective of this research program to develop effective means of achieving this seamless relationship, providing high performance implied by the CCA concept while benefiting from high-density storage capacity. There are many other details of the execution model that must be worked out as well. Both correctness and efficiency have to be achieved. But all preliminary studies of the potential have uncovered no issues that preclude a working low-level model. Latency is one possible problem and will have to be studied closely.

### **Design Complexity:**

It is an objective of the CCA Project to conduct a point-design study of the CCA-block and develop an exact design at the RTL logic abstraction. Therefore, it is too early in the evolution of these ideas to specify with certainty the degree of complexity or cost in devices implicit in the CCA-block functional requirements. Nonetheless, a representative set of operational units can be considered from the present understanding and a range of possible gate and bit counts projected. As almost all of the functionality of CCA is achieved through the resources and operation of the CCA-blocks integrated as one homogeneous ensemble, such projections may give some indication of total system size and performance when combined with the characteristics of the anticipated device technology for fabrication. There are many alternative design choices at this point. A possible one would include the following functional units comprising the CCA-block:

1. ALU - Each block is capable of performing a word-wide boolean operation every cycle although sustained operation is expected to be much less than this, dependent on the demands of the program. More complex operations such as floating point calculations are achieved through a sequence of such basic operations in time and possible space (along a string of CCA-blocks making up and ephemeral pipeline).

2. Local Object - The primary object that resides in the block, this may be a complex data element, an instruction being stored there, a node in a Futures tree, or some other house-keeping quantum of information. The local object resources include logic for matching the tag (or fields within the tag) of the local object to the search fields of transit objects.

3. Operand Object - The second operand accepted by the block and used for ALU operations. It is acquired by detecting the presence of appropriate Local Object tag bits. But once captured, it itself is not detectable by external transiting objects; thus, there is no requirement for tag matching logic. However, the tag is retained to build new objects.
4. Instruction Register - Each block may hold an active instruction that is being applied to the local and operand objects. In some blocks, an instruction which is part of a distributed compound instruction object may be stored as the local object. In this case, other objects such as operand data can locate it, supporting fixed instruction stream, dynamic data stream mode of operation. This is important for setting up ephemeral pipelines. An instruction includes the necessary operation codes to be interpreted by the CCA-block hardware and two continuation pointers that permit conditional flow control and, in the case of Futures, links pending tasks.
5. Temporary Buffer - Holds intermediate value for operations.
6. Transfer Buffers - Permits wormhole routing along paths of successive blocks. One buffer is required for each block interface (4 for 2-D, 6 for 3-D).
7. Control Element - Provides for the simple transformations implied by the possible incidence of instructions, data, and local content.

Many possible variations on this organization are being explored in the course of this research but the one described exemplifies the kinds of necessary structures and permits an estimate of required resources. Complexity here is measured in the aggregate gates and bits making up the data path of the block. The control logic is assumed to require about 15% of the total area which is comparable to experiences with other simple processing elements. While this is non-negligible, it does not dominate the total cost. The following table represents good estimates of these costs for each of the functional units within the CCA-block; but again there is opportunity for flexibility, and alternate designs may result in different assessments.

Functional Unit	Bits	Gates
ALU	0	211
Local Object	3 X 28	27
Operand	3 X 28	0
Instruction	3 X 26	0
Temp Buffer	27	0
Transfer bfr	26	26
Control	15%	15%
Total	<212	<212

The analysis above shows that at least approximately the number of gates and bits are comparable. While not a trivial piece of logic, the CCA-block is still small, consistent with its intended role as a building block of a much larger structure. It

is clear that the logic is dominated by the ALU requirements and the bit count is due mostly to the storage of the first and second operands. It may be that further analysis, especially of application program utilization will motivate an optimization that trades performance for space by slicing the ALU and requiring several cycles to complete an operation. But this may permit more CCA-blocks to be implemented in the same space and increase logic utilization as well. Analyzing this kind of trade-off is an important objective of the CCA research.

The cost estimates above assume a 3-D structure with each CCA-block directly interfaced to six nearest neighbors. As indicated earlier, an alternate structure is a 2 1/2 D organization in which a basically planar system is a few layers thick. Except on the surfaces of such a topology, the internal CCA-blocks would still be 3-D (6 faces) while the surface blocks would mostly have 5 interfaces (edge or point blocks having less). But even if a one cell thick pure 2-D structure were employed, the majority of the logic and gate requirements would remain and within the level of accuracy assumed above, would be basically the same.

One aspect of the resource requirements not discussed is that of I/O, the off-chip interconnect. Assuming a cubic structure comprising  $2(3s)$  CCA-blocks,  $3 \times 2(2s+9)$  blocks will have exposed external surfaces. If one word per cycle were to be transferred between nearest neighbor blocks on adjacent chips, then the number of interface pins would be on the order of  $2(2s+9)$ , a potential source of difficulty.

### **Technology Base:**

The major impact of Continuum Computer Architecture is on the use of future device technology. As described earlier, trends in semiconductor technology are aggravating the barriers between memory and processing logic. The advantages of the CCA approach increase as semiconductor technology advances continue. Also, as component densities increase, the kind of trade-off implied by the CCA structure becomes more feasible. It is anticipated that a threshold exists across these trends beyond which the CCA approach is favored over conventional architectures. Although sufficient analysis has not yet been performed to determine precisely where this breakpoint is (this is a complicated task involving many parameters), qualitative observations imply that it is at most in the next few years. Therefore, to be conservative, the base of technology assumed for CCA is advanced CMOS semiconductor as projected by the National Roadmap for Semiconductor Technology approximately a decade from now.

The base technology assumed (approx. year 2007) to be employed in the fabrication of the CCA system has the following characteristics:

feature size:	0.1 micron
---------------	------------

on-chip clock rate:	230 Hz
off-chip clock rate:	229 Hz
SRAM bits/chip:	232
Gates/chip:	224
I/O per chip:	212

In the above, the chip size assumed for the SRAM is somewhat less than twice that of the microprocessor chip from which the gate capacity is derived. Even so, the ratio of SRAM bits to logic gates per unit area is approximately two orders of magnitude. If the benefits of tight SRAM bit density can be exploited, at least in part, then the chip area cost for a single CCA-block is dominated by the the logic. Based on this (perhaps optimistic appraisal) and the complexity analysis in the previous section, an estimate of the number of blocks that can be implemented on a single chip in 2007 is 212 or about 4,000. At the 1 GHz on-chip clock rate expected for this technology, the peak performance is approximately 4 Tops (242) per chip. A major system configuration is limited in size by practical engineering considerations and might range between 10,000 and 100,000 chips although some studies suggest that a million dies might be feasible. A large system of 216 chips would have a peak (not-to-be-achieved) performance of 258 ops or a couple of hundred peta-ops in the year 2007.

A number of factors implicit in the structure and its assumed operation dramatically reduce this number to a plausible sustained performance metric such as flops. These factors include (but are probably not limited to):

- a. The utilization or percentage of ALUs active during any one cycle,
- b. The number of cycles required to complete a single ALU operation,
- c. The number of pipeline stages of ALUs in a sequence of CCA-blocks required to perform a basic floating point operation,
- d. The percentage of operations that are floating point in the application program instruction mix.

The first factor is probably low due to usage of the computing medium, the parallelism of the code and the latency for moving objects (data and instruction) to coincide. A guess is 2-4 or about 7% and is a little worse than achieved with conventional methods, so is considered conservative. The second factor should be 1 but could be as poor as 1/4 so will be estimated at 2-1 or one half. The third factor may vary significantly based on ALU design. Assuming a relatively simple ALU but with hooks that facilitate floating point operation, this is estimated at 2-5 or requiring about 32 pipeline stages. The fourth factor can be derived from instruction mix studies performed on scientific codes. While this may vary by more than a factor of two, a reasonable number is 1/4 or 2-2. The total performance degradation from these contributing factors is estimated at approximately 2-12 with the caveat that this may be off by more than a factor of

2 in either direction. Therefore, the estimated sustained performance in floating point operations is 246 flops or approximately 100 Teraflops in the year 2007.

One problem area revealed by this analysis concerns I/O from the previous section. It was shown that the total number of pins required per chip is approximately  $2(2s+9)$  where  $s$  is the log base 2 of the edge length for a cube of blocks on chip. For this technology,  $s=4$  and the requirement is for 217 pins or 32 times as many as are expected to be available. Clearly, a more elegant solution will have to be found than this brute force approach. Since, not every block will be transferring data between itself and nearest neighbor blocks every cycle, it is possible that I/O pins can be aggregated, sharing their bandwidth across collections of cells. This and other alternatives to the inter-chip I/O problem will be explored as a major objective of the proposed research.

A second problem is that of the memory shell, the external passive storage required to provide adequate storage capacity. A mix of SRAM and DRAM will probably be employed in a shallow memory hierarchy. DRAM technology will provide 2 GBytes per chip by the year 2007 according to the SIA projections. Studies of large applications programs have shown that many scientific and engineering simulations would require approximately 16 Terabytes of storage for a half-petaflops machine. This could be provided by 213 chips or about 8,000. The high speed buffer between this backing store and the CCA core would require an equivalent chip count, increasing the total chip count over that of the CCA-core by about 25%. The exact method of interface has yet to be determined.

### **Technical Plan:**

The CCA Project is the first to address the promise of continuum computing and therefore does not continue from a previous such project. The goal of the program is to establish the basic foundation for Continuum Computer Architecture by developing the set of local block functions and global dynamic operational strategies to produce correct parallel execution and prove viability of the approach. To achieve this, the following major technical activities are being undertaken:

1. A local and global assembler-level instruction set, representing the CCA-block local functions and global dynamic operations will be developed. The functional relationship to the memory shell will be devised.
2. An emulator will be implemented, representing the abstract functionality of the continuum blocks, permitting correct programs to be executed. The correctness of the programming/execution model will be validated and the

functionality of the elements will evolve through the use of this tool. The emulator will include the logical interface to the memory shell.

3. A simulator will be developed reflecting the physical distribution of the continuum blocks. Data structures permitting parallel operation will be studied and performance data acquired. Policies governing the management of the parallel resources will be devised and tested. The simulator will be ported to an MPP to provide adequate memory and processing capability for meaningful experiments. A small number of real-world applications will be ported to the CCA programming model and used in the performance and operation studies.

4. An RTL design of the continuum block will be developed. Estimates of logic complexity and die size will be carried out in the context of the SIA projections of semiconductor technology advances to assess engineering practicality including power requirements.

The Continuum Computer Architecture is unique and provides an innovative approach to Ultrascale computing. To the knowledge of the institute, no other research program in computer architecture has formulated an approach that is equivalent to that reflected by the CCA structures and methods as described in the last section. Nonetheless, many elements of the research relate to ideas that have, in different forms, been found in earlier work of others. This section touches on the important instances of related approaches reflected by prior art.

### **Cellular Automata:**

Cellular automata is a digital systems structure that has been studied since the earliest period of the digital electronic computer era and embodies the basic idea of symbiosis between a set of highly replicated sites governed by local state transformation rules and the aggregate global behavior of the system. While fundamental to both cellular automata and continuum computer architecture, the similarity in use of symbiotic relationships between the two architecture concepts ends at that point. Cellular automata routinely comprise cells, each of which incorporates a few bits of state, trivial bit level logic, and simple interface to nearest neighbor cells. The simple bit-level operations performed by cellular automata in the aggregate have been shown to statistically relate to such physical phenomena as fluid mechanics, population dynamics, and diffusion of materials. In the late 1940s, work of John von Neumann showed that cellular automata were Turing equivalent. But from a practical standpoint, they have not provided an effective means of achieving general computing. CCA differs from cellular automata in that the objects manipulated at the block level are multi-word constructs which are typed through tags. The symbiotic relationship is not between low level rules and some physical phenomena but rather between a relatively complex set of local transformations and a global model of general computation. Cellular automata can be considered as very special purpose and limited while CCA is general purpose.

## **Wormhole Routing:**

Efficient transfer of data through two and three dimensional networks was developed by Seitz and Dally in the form of Worm-hole Routing where a packet moves bit serially through a sequence of nodes arriving at a destination in time little more than that required for the leading bits to propagate through the distance. CCA borrows from this powerful mechanism and extends it in two important ways. The first is that the physical destination is not necessarily known exactly but is determined by a tag which is matched on the fly as the communications packet moves past. This was described in section F in some detail and more is said as it relates to virtual name consistency. The second extension is to merge wormhole routing which dynamically seeks a path through successive stages with vector pipelining that achieves high throughput of complex operations like floating point by breaking such operations up into a succession of simple actions, each performed by an adjacent logic element to the last one. The CCA provides a new concept called "Work-Hole" routing which sets up vector pipeline-like sequences of successive blocks and passes data streams through it. But such operational organizations are purely logical as they are achieved by laying out a sequence of instructions in the adjacent blocks. As soon as the vector operation is completed, the instructions go away and the work-hole dissolves making it truly ephemeral, as is the particular route between two points in worm-hole routing.

## **Futures:**

The Futures construct was developed by Hewitt for the Actors model about two decades ago and was employed by Halstead as the critical parallel construct in the MultiLisp execution environment developed for the Concert multiprocessor. Futures is also similar to I-structures devised by Arvind fifteen years ago as part of the functional programming language Id for dynamic data flow. The semantics of the Futures construct is borrowed directly from this previous work. While some variations occur in the CCA context, it is essentially the same concept. The difference between this previous body of work and the CCA usage is not in the semantics but in the implementation. The CCA-block incorporates dedicated logic to the implementation of the Futures construct in CCA and therefore it is part of the low level instruction set architecture. This allows the critical path time of a Futures to be as low as one machine cycle although releasing many suspended tasks may take a number of cycles. Also, the CCA Futures continuation list is organized as a tree rather than a linked-list to permit parallel spawning of suspended threads. It should be noted that Burton Smith's Tera computer, a multi-threaded architecture, includes memory tag bits to facilitate managing Futures in software.

Single cycle suspension of active threads of execution have been demonstrated in other contexts as well. Again, the Tera architecture will do this on a cycle by cycle basis as it selects successive operations from among a set of active threads. Dally's J-Machine does this for a couple of active threads and Alewife suspends threads and swaps context in just a few cycles. Of course, fine grain dataflow machines like Papodapoulos' Monsoon selected fine grain operations for execution only when operand values were available.

### **Distributed Name Consistency:**

The Continuum Computer Architecture supports a global name space with virtual objects; that is, object names are not tied to specific physical locations. This requires that a means of locating objects and performing virtual to physical translation be enabled. Research in cache consistency methods have some similarity to the approach used for CCA. Tags on data elements in the CCA-blocks are similar to tagged cache lines although the tags are not retained for the data objects in main memory. Snooping techniques developed for common bus multiprocessors such as the SGI power challenge have active logic that matches the addresses requested by other processors to the cache-line tags in the caches of every processor. But this method is restrictive in scaling. The KSR-1 employed a segmented ring bus where a request went around, testing the contents of the processor caches at succeeding sites. The Convex Exemplar at its highest level of SCI cache management uses a similar approach. This is similar to CCA but CCA can do this along any adaptive route that it might take in 2D or 3D space using temporary markers if need be at turns. The Dash and Alewife multiprocessors employ directory based schemes to permit higher degree of scaling for cache consistency. The direct mapped approach developed for DASH and used at the lower level by the Convex Exemplar employs a resource tag to indicate where copies exist. Alewife employs a linked reference tree (if necessary) to achieve the same end. CCA does not have the cache coherency problem usually because it does not have caches. But copies are migrated. The linked list reference tree method is modified for use by CCA to maintain associations between elements of state within the broader process context. The matching methods mentioned are used for the physical to virtual address translation similar to the KSR but at a much finer granularity.

The investigators have been directly involved in research in parallel system architecture and applications for almost two decades. This breadth of experience is critical to understanding the nature of the opportunities and challenges that constitute the intellectual contributions of the CCA research. The insights acquired through the course of this body of work have been of fundamental importance throughout the process of creating the innovative concepts underlying the Continuum Computer Architecture model. The investigators recognize that the concepts present a radical departure from conventional

methods. A brief discussion of some of the more germane accomplishments in this field by the investigators will demonstrate the basis from which the new ideas are derived and illustrate the capabilities of the investigators themselves.

An early eight processor special purpose computer architecture was developed for the simulation of power electronic circuits. This SPMD (single program stream, multiple data stream) architecture involved the close mapping of the application internal data set and associated task on to the distributed physical resources. The SPMD structure employed a single control processor that supervised the overall program execution and transfer of intermediate data values between processors. The contribution of this work is that it demonstrated the importance of the relationship between the distributed application abstract elements and the physical execution resources in achieving performance and efficiency.

The Concert Multiprocessor that was developed provided years of in-depth experience with an early large scale parallel computer comprising 64 processing nodes in a hierarchical shared memory structure. This work emphasized the importance of factors contributing to performance degradation including overhead, latency, contention, and starvation. To study these sources of loss, several instrumentation systems were developed and employed within the system to measure behavior of hardware and software. These experiences exposed the importance of locality to performance as well as the critical need for achieving a balance of resources that match the computational demands of the parallel applications.

Simultaneous Pascal, a parallel extension of Pascal, was developed to provide a high level programming interface and incorporated a runtime system for dynamic scheduling of threads. This framework was used for detailed studies of overhead demonstrating the limitations to parallel thread granularity and the performance opportunity derived from dynamic load balancing to reduce local starvation from insufficient work.

Multilisp, developed by Halstead, was ported to the Concert Multiprocessor and provided a significant environment in which to study the potential of the Futures construct as a basis for delineating parallelism in applications. Multilisp embodied many of the semantic elements that will comprise the CCA programming model. The strong distinction and control of value-oriented and side-effect based computation, the ability to manage processes as first class objects, and the manipulation of instruction streams as standard data objects all contribute to the operation of the CCA system and were explored using Multilisp on the Concert Multiprocessor.

Asynchronous control of fine grain execution was explored through the discipline of data flow architecture. The Yarc static dataflow architecture was designed and

implemented to achieve high throughput using costly (at the time) floating point and communications resources for applications in signal processing. This project provided experience with hardware support for data-driven fine grain flow control including distributed control state and functional semantics. It also exposed the importance of achieving relative proximity of closely related activity to constrain critical path time while achieving load balance through uniform distribution of less tightly coupled activities.

The Anida data flow architecture was developed and simulated to study techniques of managing variable granularity tasks with hardware supported synchronization and communication mechanisms. The Anida studies exposed the problems associated with fixed allocation of tasks to processors, especially for applications that were runtime dependent on intermediate results in directing program flow control. The Anida studies also revealed the limitations of relying on basic block threads delineated by successive conditional flow control operations and ways to circumvent these problems at higher levels of granularity without having to resort to speculative execution.

The ATD (Associative Template Dataflow) architecture was developed and simulated to study innovative mechanisms for managing distributed fine grain execution while balancing resource throughput. The unique property of the ATD architecture was the elimination of tokens as the means of communication and synchronization. Instead, it employed a multicast mechanism that established virtual channels between the source of a result value and all of the operations that depended on that value as input operands. This is very similar to the kind of mechanism to be used in CCA. A second key contribution of ATD to CCA is the use of nearest neighbor communications and exploitation of adjacency properties of physical organization to achieve extremely high memory bandwidth. The ATD incorporated dynamic scheduling of operations in time but spatial assignment was limited to compile time. In some cases this constraint proved to be a problem because the requirements of distribution would change as execution proceeded.

Mechanisms for maintaining a global name space through hardware in a scalable distributed environment was studied extensively using the Convex Exemplar scalable parallel processor. This system supports a two-level cache coherence strategy within an SMP and across clusters of SMPs. A series of detailed application driven studies demonstrated the value of a shared memory programming and execution environment but at the same time revealed the limitations due to lack of latency management other than caches. In many cases, it was shown that the policies employed by cache management did not match to data access patterns of applications sometimes causing serious performance degradation. The relatively heavy weight threads required by the low efficiency scheduling mechanisms restricted the use of dynamic scheduling for load

balancing and imposed more programmer direct control of parallel task operation. CCA adopts the key value of a shared name space and global mutable objects for memory efficiency.

The investigators have extensive experience with the development, porting, and evaluation of large parallel applications to MPPs. A set of parallel benchmarks were developed for the Earth and space sciences community to be used in system evaluation for NASA. These and other studies have revealed the nature, behavior, and requirements of a wide array of large scientific and engineering codes for parallel processing. The importance of data movement, conditional flow control, and variable parallelism has been examined in detail. The results of these studies is an important driver in deriving a balanced CCA system structure.

The investigators have been involved in several projects aimed at evaluating advanced computer architectures with applications. They participated in the creation of the PERFECT Club benchmark project and contributed to the first two years of that project's research. One of the investigators led an effort that evaluated the performance of seven applications on 15 computers with widely different architectures (SIMD, several MIMD architectures, VLIW, and vector). In another study, the data movement patterns of a number of scientific applications on parallel computers were measured and provided insight on the value of architectural features that support certain data movement patterns, such as multicast and partitioning of I/O and communications network traffic.

Opportunities and challenges of achieving ultrascale computing in the regime of a petaflops has been the focus of a two year collaborative effort with colleagues across the community. The investigators have been principals in the development of this body of knowledge resulting in a book published from this work: Sterling, T., Messina, P., Smith, P.H., "Enabling Technologies for Petaflops Computing," MIT Press, 1995. These detailed studies in advanced device technologies, parallel architecture, parallel applications characteristics, and system software issues has yielded almost unique understanding of the challenges of ultrascale computing. The limitations imposed by such problems as latency, memory bandwidth, chip power requirements, and I/O force a re-thinking of the way in which physical resources can be brought to bear on the requirements of applications with massive data sets and parallelism. This work has demonstrated that a Petaflops computer can be developed using conventional approaches but only from technology that will be available no sooner than 20 years from now. Furthermore, new methods of latency management and locality control will have to be devised in order for such systems to work effectively. It is within the context of this base of experience that the approach is defined to yield equivalent capabilities in ten years and with a framework that resolves many of the programming and resource management problems confronting the conventional methods.

## **CACR Computing Facilities (1996):**

The Center for Advanced Computing Research (CACR) operates a variety of computing resources that will be available for use in carrying out this project's research.

The largest scale machines are a 512 compute-node Intel Paragon L38, a 256 processor CRAY T3D, the 512 compute-node Intel Touchstone Delta, and a 52 compute-node Paragon A4.

Intel Paragon systems are distributed-memory Multiple Instruction, Multiple Data (MIMD) computers whose nodes are connected in a two-dimensional mesh with 200 megabytes per second bidirectional bandwidth. Paragons have several types of nodes: compute nodes, service nodes, I/O nodes, and network nodes. All are based on one building block: General Purpose (GP) nodes. Each Paragon GP node has an application processor and a message-passing co-processor both of which are Intel i860 XP RISC microprocessors.

The Paragon L38 has a total of 541 nodes: 512 compute nodes, six service nodes, 21 disk I/O nodes, and two HIPPI network nodes. All of Trex's nodes have 32 megabytes of memory. The compute nodes provide an aggregate peak speed of 38.4 gigaflops and 16.4 gigabytes of memory. The disk subsystem consists of 21 nodes each with five disks and a RAID3 controller with a total capacity of 105 gigabytes.

The CRAY T3D is also an MIMD computer; its nodes (each of which has two DEC Alpha processors) are connected by a three-dimensional toroidal network with 300 megabytes per second bidirectional bandwidth. Each processor has 64 megabytes of memory; aggregate memory capacity is 16 gigabytes.

The Intel Touchstone Delta System was a prototype for the Intel Paragon family of computers and is similar to the Paragon L38: 512 compute nodes, 8 gigabytes of memory, and 90 gigabytes of disk.

## **Mass Storage:**

A Convex C210 with the UniTree archival file system currently serves as the CACR archival file system. The mass storage system includes nine gigabytes of disk cache, two HIPPI interfaces, and two Metrum tape drives. In the spring of 1996, a new archival file system will be deployed; it will run the High Performance Storage System (HPSS) software instead of UniTree and the hardware base will be two large IBM RS6000 systems and an IBM 3494 tape robot (20 terabytes capacity) with four IBM 3490-C2A tape drives. These tape drives read/write at 10 MB/s.

### **File Servers and Workstations:**

CACR maintains a cluster of roughly 45 Unix systems: servers, workstations, xterminals, and PCs. The bulk (80%) of the machines are Sun workstations and servers. At the hub of this setup are two 690 MP model 512 Sun servers that act as file servers, compile engines for the Intel Supercomputers, central login machines for remote users, and routers. In addition, there are two 24-bit color workstations in the CACR visualization laboratory that act as front ends for a video recorder and color printer. With funding from this project we will acquire additional workstations running common LISP and with memory size and processor speed suitable for use in part of the early work of this project.