



# **BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA**

## **Facultad De Ciencias De La Computación**

### **Tesis:**

Programación y monitoreo de  
aplicaciones paralelas con  
Charm++

### **Que para obtener el Título de:**

LICENCIADO EN CIENCIAS DE LA  
COMPUTACIÓN

### **Presenta:**

Eric Olmedo González

### **Asesor de Tesis:**

Dr. Manuel Isidro Martín Ortiz

Noviembre del 2009  
Puebla, Puebla.

# Índice General

|                                                                       |    |
|-----------------------------------------------------------------------|----|
| Introducción .....                                                    | 1  |
| Arquitectura De Computadoras.....                                     | 3  |
| 1.1    Arquitectura de Von Neumann .....                              | 3  |
| 1.2    La Arquitectura Harvard .....                                  | 4  |
| 1.3    Clasificación de Flynn.....                                    | 5  |
| 1.3.1 SISD (Single Instruction Single Data) .....                     | 6  |
| 1.3.2 SIMD (Single Instruction Multiple Data) .....                   | 6  |
| 1.3.2.1 Computadora Vectorial .....                                   | 6  |
| 1.3.2.2 Procesadores vectoriales memoria a memoria .....              | 7  |
| 1.3.3 MIMD (Multiple Instruction Multiple Data).....                  | 8  |
| 1.3.3.1 Sistemas de Memoria Compartida .....                          | 9  |
| 1.3.3.1.1 Sistema UMA (Uniform Memory Access) .....                   | 10 |
| 1.3.3.2 Sistemas de Memoria Distribuida. ....                         | 10 |
| 1.3.3.3 Sistemas de Memoria Compartida Distribuida .....              | 11 |
| 1.3.3.3.1 Sistema NUMA (Non Uniform Memory Access) .....              | 12 |
| 1.3.3.3.2 CC-NUMA (Cache Coherent Nonuniform Memory Access).....      | 13 |
| 1.3.3.3.3 Cluster Jerárquico .....                                    | 13 |
| 1.3.3.3.4 COMA (Cache Only Memory Access) .....                       | 14 |
| 1.3.4 MISD (Multiple Instruction Single Data) .....                   | 14 |
| Clusters De Computadoras .....                                        | 15 |
| 2.1    Definición de cluster.....                                     | 15 |
| 2.2    Historia de los Cluster de Computadoras.....                   | 17 |
| 2.3    Clasificación de un cluster.....                               | 19 |
| 2.4    Componentes de un Cluster .....                                | 21 |
| 2.4.1 Nodos.....                                                      | 21 |
| 2.4.2 Sistemas Operativos .....                                       | 22 |
| 2.4.3 Conexiones de Red .....                                         | 22 |
| 2.4.4 Almacenamiento .....                                            | 22 |
| 2.4.5 Middleware.....                                                 | 22 |
| 2.4.6 Protocolos de Comunicación .....                                | 23 |
| 2.4.7 Ambientes de programación paralela .....                        | 23 |
| 2.5    Ventajas y desventajas de un cluster .....                     | 23 |
| 2.6    TOP500.org, las 500 computadoras más poderosas del mundo ..... | 24 |
| 2.6.1 IBM Roadrunner .....                                            | 26 |
| 2.6.2 Sun Constellation Linux Cluster: Ranger .....                   | 26 |
| 2.6.3 Sun Constellation Linux Cluster: JUROPA .....                   | 27 |
| 2.6.4 Aitzaloe.....                                                   | 29 |
| Programación Distribuida y Paralela.....                              | 30 |

|         |                                                            |    |
|---------|------------------------------------------------------------|----|
| 3.1     | Sistemas Distribuidos .....                                | 30 |
| 3.1.1   | Características de los sistemas distribuidos.....          | 31 |
| 3.1.2   | Ejemplos de algunos sistemas .....                         | 35 |
| 3.1.3   | Programación Distribuida.....                              | 35 |
| 3.2     | Programación Paralela .....                                | 35 |
| 3.2.1   | ¿Qué es Paralelismo?.....                                  | 36 |
| 3.2.2   | ¿Qué es Cómputo Paralelo?.....                             | 36 |
| 3.2.3   | ¿Cuándo usar paralelismo? .....                            | 36 |
| 3.2.4   | Características de un programa paralelo.....               | 37 |
| 3.2.4.1 | Granularidad.....                                          | 37 |
| 3.2.4.2 | Paralelismo implícito y explícito.....                     | 38 |
| 3.2.4.3 | Diseño de Algoritmos Paralelos .....                       | 38 |
| 3.2.4.4 | Sincronización .....                                       | 38 |
| 3.2.4.5 | Latencia.....                                              | 39 |
| 3.2.4.6 | Aceleración Y Eficiencia .....                             | 39 |
|         | Charm++.....                                               | 40 |
| 4.1     | Historia de Charm++ .....                                  | 40 |
| 4.2     | Introducción a Charm++ .....                               | 41 |
| 4.2.1   | Objetos Chare .....                                        | 42 |
| 4.2.2   | Mensajes .....                                             | 43 |
| 4.2.3   | Métodos de entrada .....                                   | 43 |
| 4.2.4   | Proxies .....                                              | 43 |
| 4.2.5   | Colecciones de Chare .....                                 | 44 |
| 4.2.5.1 | Arreglos de Chares.....                                    | 44 |
| 4.2.5.2 | Grupos de Chares.....                                      | 44 |
| 4.2.5.3 | Grupos nodo .....                                          | 44 |
| 4.3     | Sistema de ejecución de Charm++ .....                      | 45 |
| 4.3.1   | La capa de Máquina.....                                    | 46 |
| 4.3.2   | Converse .....                                             | 46 |
| 4.3.3   | Cola de mensajes .....                                     | 46 |
| 4.3.4   | Planificador .....                                         | 46 |
| 4.4     | Proceso de compilación y ejecución de una aplicación ..... | 47 |
| 4.4.1   | Charmc .....                                               | 48 |
| 4.4.2   | Charmxi y el archivo interfaz .....                        | 48 |
| 4.4.3   | Ejecutando un programa Charm++ .....                       | 48 |
| 4.4.4   | Charmrun .....                                             | 48 |
| 4.4.5   | El archivo Nodelist.....                                   | 49 |
| 4.5     | Ejemplo de un programa Charm++ .....                       | 49 |
| 4.5.1   | El archivo de cabecera (main.h) .....                      | 49 |
| 4.5.2   | El archivo Fuente (main.C) .....                           | 50 |
| 4.5.3   | El Archivo Interfaz (main.ci).....                         | 51 |
| 4.5.4   | El archivo Makefile .....                                  | 51 |
| 4.6     | Salida en pantalla del programa "Hola Mundo" .....         | 52 |
| 4.7     | Elementos de una aplicación en Charm++ .....               | 53 |
| 4.7.1   | Objetos Chare .....                                        | 53 |

|                                                                                                         |    |
|---------------------------------------------------------------------------------------------------------|----|
| 4.7.1.1 Creación de una Chare.....                                                                      | 54 |
| 4.7.1.2 Invocación de métodos de una chare.....                                                         | 55 |
| 4.7.1.3 Acceso local .....                                                                              | 56 |
| 4.7.2 Envío de Parámetros.....                                                                          | 56 |
| 4.7.3 Mensajes .....                                                                                    | 57 |
| 4.7.3.1 Creación y borrado de mensajes.....                                                             | 58 |
| 4.7.3.2 Mensajes con arreglos de tamaño variable .....                                                  | 59 |
| 4.7.3.3 Empaquetado de mensajes de tamaño variable .....                                                | 60 |
| 4.7.3.4 Creación de Prioridades .....                                                                   | 62 |
| 4.7.4 Métodos de entrada .....                                                                          | 64 |
| 4.7.4.1 Atributos de los métodos de entrada.....                                                        | 64 |
| 4.7.5 Arreglos de Chares .....                                                                          | 66 |
| 4.7.5.1 Declarando un Arreglo Unidimensional (1D).....                                                  | 66 |
| 4.7.5.2 Crear un Arreglo.....                                                                           | 67 |
| 4.7.5.3 Mensajes en arreglos.....                                                                       | 67 |
| 4.7.5.4 Broadcast.....                                                                                  | 67 |
| 4.7.5.5 Reducciones sobre arreglos .....                                                                | 68 |
| 4.7.5.6 Borrado de arreglos.....                                                                        | 68 |
| 4.7.5.7 Declarando índices de arreglos multidimensionales o definidos por el usuario .....              | 68 |
| 4.7.5.8 Índice definido por usuario .....                                                               | 69 |
| 4.7.5.9 Inserción Dinámica de Elementos.....                                                            | 70 |
| 4.7.6 Grupos de chares .....                                                                            | 71 |
| 4.7.6.1 Creación de un grupo .....                                                                      | 72 |
| 4.7.6.2 Invocación de métodos y funciones de los Grupos.....                                            | 72 |
| 4.7.7 Objetos Grupos Nodo (NodeGroup).....                                                              | 73 |
| 4.7.7.1 Invocación de métodos en un Grupo Nodo.....                                                     | 74 |
| 4.7.8 Balance de carga. ....                                                                            | 74 |
| 4.7.8.1 Estrategia de migración de objetos basado en la medición.....                                   | 75 |
| 4.7.8.2 Estrategia de Balance de carga disponible .....                                                 | 75 |
| 4.7.8.3 Balance de carga en arreglos chare .....                                                        | 76 |
| 4.7.8.4 Opciones para usar el módulo de balance de carga en el compilador y en tiempo de ejecución..... | 77 |
| 4.7.8.5 Otras opciones útiles.....                                                                      | 78 |
| 4.7.9 Quiescence .....                                                                                  | 79 |
| 4.7.10 Reducciones .....                                                                                | 80 |
| 4.7.10.1 ¿Por qué usar reducciones?.....                                                                | 80 |
| 4.7.10.2 ¿Cómo usar reducciones? .....                                                                  | 81 |
| 4.7.10.3 Tipos De Reducciones.....                                                                      | 82 |
| 4.7.11 Callbacks .....                                                                                  | 83 |
| 4.7.11.1 Interfaz del Cliente .....                                                                     | 84 |
| 4.7.11.2 Método de Entrada Callback.....                                                                | 84 |
| 4.7.12 Empaquetamiento y desempaquetamiento de datos PUP .....                                          | 86 |
| 4.7.12.1 El uso de PUP .....                                                                            | 86 |
| 4.7.12.2 Operador PUP.....                                                                              | 87 |
| 4.7.12.3 PUP como bytes .....                                                                           | 88 |

---

|                                                                                         |     |
|-----------------------------------------------------------------------------------------|-----|
| 4.7.12.4 Sobre carga PUP .....                                                          | 88  |
| 4.7.12.5 Modos que se utilizan en la rutina PUP .....                                   | 88  |
| 4.7.12.6 Ciclo de un objeto PUP .....                                                   | 89  |
| 4.7.12.7 Estructuras y clases enviadas a través de la red definidas por el usuario .... | 89  |
| 4.7.13 Otros Comandos .....                                                             | 91  |
| 4.7.13.1 Comandos de entrada y salida .....                                             | 92  |
| 4.8 Projections .....                                                                   | 93  |
| 4.8.1 Rastreo Selectivo .....                                                           | 94  |
| 4.8.2 Eventos definidos por el usuario .....                                            | 94  |
| 4.8.3 Modos de Rastreo .....                                                            | 95  |
| 4.8.4 Herramienta de visualización de rendimiento .....                                 | 95  |
| 4.9 Debugger .....                                                                      | 96  |
| 4.9.1 Generar la herramienta de depuración .....                                        | 97  |
| 4.9.2 Record Replay .....                                                               | 98  |
| 4.9.3 Sistema de Depuración Integrado de Charm++ .....                                  | 98  |
| 4.9.4 Opciones del depurador .....                                                      | 100 |
| 4.9.5 El Depurador .....                                                                | 100 |
| 4.9.5.1 Visualización de la Memoria .....                                               | 103 |
| 4.9.6 Inspector framework .....                                                         | 105 |
| 4.9.7 Limitaciones del Depurador .....                                                  | 106 |
| Aplicaciones De Ejemplo .....                                                           | 107 |
| 5.1 Calcular el valor de $\pi$ por el método de Montecarlo .....                        | 108 |
| 5.2 Encontrar los primos que existen en un rango .....                                  | 121 |
| 5.3 Factorizar un número con el método de Fermat .....                                  | 133 |
| 5.3.8 Método de Fermat .....                                                            | 134 |
| 5.4 Buscar factores de números más grandes .....                                        | 149 |
| Conclusiones .....                                                                      | 156 |
| Apéndice A. Instalación de Charm++ .....                                                | 158 |
| Apéndice B Ejecutando un programa .....                                                 | 162 |
| Bibliografía .....                                                                      | 164 |

# Índice De Figuras

|                                                                                    |     |
|------------------------------------------------------------------------------------|-----|
| Fig. 1.1 Diagrama de la Arquitectura de Von Newmann .....                          | 4   |
| Fig. 1.2 Taxonomía de Flynn.....                                                   | 5   |
| Fig. 1.3 Modelo SISD (Single Instruction Single Data).....                         | 6   |
| Fig. 1.4 Modelo SIMD (Single Instruction Multiple Data).....                       | 6   |
| Fig. 1.5 Modelo MIMD (Single Instruction Multiple Data).....                       | 8   |
| Fig. 1.6 Sistemas de Memoria Compartida. ....                                      | 9   |
| Fig. 1.7 Sistemas de Memoria Distribuida. ....                                     | 11  |
| Fig. 1.8 Sistemas de Memoria Compartida Distribuida. ....                          | 12  |
| Fig. 1.9 MISD (Multiple Instruction Single Data). ....                             | 14  |
| Fig. 4.1 Representación de los objetos en un cluster y en cada procesador .....    | 42  |
| Fig. 4.2 Diagrama General de la composición de Charm++ .....                       | 47  |
| Fig. 4.3 Prioridades de tipo bitvector en un árbol.....                            | 63  |
| Fig. 4.4 Ciclo de vida de un objeto con la rutina pup.....                         | 89  |
| Fig. 4.5 Ventana de uso del procesador o procesadores .....                        | 96  |
| Fig. 4.6 Ventana del depurador de Charm++ .....                                    | 99  |
| Fig. 4.7 Depurador cuando un punto de ruptura es alcanzado.....                    | 101 |
| Fig. 4.8 Elementos de una matriz en un punto detenido de la aplicación.....        | 102 |
| Fig. 4.9 El depurador puede usar instancias de gdb sobre los procesadores .....    | 103 |
| Fig. 4.10 Vista de la memoria principal .....                                      | 104 |
| Fig. 4.11 Estadísticas de la memoria .....                                         | 104 |
| Fig. 4.12 Memoria después de ejecutar la prueba de acceso .....                    | 105 |
| Fig. 4.13 Árbol de la variables y funciones virtuales .....                        | 106 |
| Fig. 5.1 Representación de puntos aleatorios en cuadrado y círculo inscritos. .... | 108 |
| Fig. 5.2 Gráfica de uso de procesamiento con 1 procesador .....                    | 116 |
| Fig. 5.3 Gráfica de uso de procesamiento con 2 procesadores.....                   | 116 |
| Fig. 5.4 Gráfica de uso de procesamiento con 3 procesadores.....                   | 117 |
| Fig. 5.5 Gráfica con 4 procesadores .....                                          | 117 |
| Fig. 5.6 Gráfica con 5 procesadores .....                                          | 117 |
| Fig. 5.7 Gráfica con 6 procesadores .....                                          | 118 |
| Fig. 5.8 Gráfica con 7 procesadores .....                                          | 118 |
| Fig. 5.9 Gráfica con 8 procesadores .....                                          | 118 |
| Fig. 5.10 Gráfica con 9 procesadores .....                                         | 118 |
| Fig. 5.11 Gráfica con 10 procesadores .....                                        | 118 |
| Fig. 5.12 Gráfica con 11 procesadores .....                                        | 118 |
| Fig. 5.13 Gráfica con 12 procesadores .....                                        | 119 |
| Fig. 5.14 Gráfica con 13 procesadores .....                                        | 119 |
| Fig. 5.15 Gráfica con 14 procesadores .....                                        | 119 |
| Fig. 5.16 Gráfica con 15 procesadores .....                                        | 119 |
| Fig. 5.17 Gráfica con 16 procesadores .....                                        | 119 |

|           |                                                                                   |     |
|-----------|-----------------------------------------------------------------------------------|-----|
| Fig. 5.18 | Gráfica con 17 procesadores .....                                                 | 119 |
| Fig. 5.19 | Gráfica con 18 procesadores .....                                                 | 120 |
| Fig. 5.20 | Gráfica con 19 procesadores .....                                                 | 120 |
| Fig. 5.21 | Gráfica con 20 procesadores .....                                                 | 120 |
| Fig. 5.22 | Gráfica de procesamiento con 1 procesador .....                                   | 128 |
| Fig. 5.23 | Gráfica con 2 procesadores .....                                                  | 128 |
| Fig. 5.24 | Gráfica con 3 procesadores .....                                                  | 128 |
| Fig. 5.25 | Gráfica con 4 procesadores .....                                                  | 128 |
| Fig. 5.26 | Gráfica con 5 procesadores .....                                                  | 128 |
| Fig. 5.27 | Gráfica con 6 procesadores .....                                                  | 128 |
| Fig. 5.28 | Gráfica con 7 procesadores .....                                                  | 128 |
| Fig. 5.29 | Gráfica con 8 procesadores .....                                                  | 129 |
| Fig. 5.30 | Gráfica con 9 procesadores .....                                                  | 129 |
| Fig. 5.31 | Gráfica con 10 procesadores .....                                                 | 129 |
| Fig. 5.32 | Gráfica con 11 procesadores .....                                                 | 129 |
| Fig. 5.33 | Gráfica con 12 procesadores .....                                                 | 129 |
| Fig. 5.34 | Gráfica con 13 procesadores .....                                                 | 129 |
| Fig. 5.35 | Gráfica con 14 procesadores .....                                                 | 129 |
| Fig. 5.36 | Gráfica con 15 procesadores .....                                                 | 129 |
| Fig. 5.37 | Gráfica con 16 procesadores .....                                                 | 129 |
| Fig. 5.38 | Gráfica con 17 procesadores .....                                                 | 129 |
| Fig. 5.39 | Gráfica con 18 procesadores .....                                                 | 129 |
| Fig. 5.40 | Gráfica con 19 procesadores .....                                                 | 129 |
| Fig. 5.41 | Carga de los 20 procesadores en la aplicación .....                               | 130 |
| Fig. 5.42 | La carga de todos los procesadores .....                                          | 131 |
| Fig. 5.43 | La carga del segundo procesador tarda más su ejecución. ....                      | 131 |
| Fig. 5.44 | Carga de cada procesador .....                                                    | 131 |
| Fig. 5.45 | Porcentajes de carga en cada procesador de una función que se ejecuta .....       | 131 |
| Fig. 5.46 | Animación de la carga de trabajo en los procesadores al ejecutar la aplicación... | 132 |
| Fig. 5.47 | Gráfica de utilización del CPU para 1 procesador .....                            | 141 |
| Fig. 5.48 | Gráfica para 2 procesadores .....                                                 | 142 |
| Fig. 5.49 | Gráfica para 3 procesadores .....                                                 | 142 |
| Fig. 5.50 | Gráfica para 4 procesadores .....                                                 | 142 |
| Fig. 5.51 | Gráfica para 5 procesadores .....                                                 | 142 |
| Fig. 5.52 | Gráfica para 6 procesadores .....                                                 | 142 |
| Fig. 5.53 | Gráfica para 7 procesadores .....                                                 | 142 |
| Fig. 5.54 | Gráfica para 8 procesadores .....                                                 | 142 |
| Fig. 5.55 | Gráfica para 9 procesadores .....                                                 | 142 |
| Fig. 5.56 | Gráfica para 10 procesadores .....                                                | 142 |
| Fig. 5.57 | Gráfica para 11 procesadores .....                                                | 143 |
| Fig. 5.58 | Gráfica para 12 procesadores .....                                                | 143 |
| Fig. 5.59 | Gráfica para 13 procesadores .....                                                | 143 |
| Fig. 5.60 | Gráfica para 14 procesadores .....                                                | 143 |
| Fig. 5.61 | Gráfica para 15 procesadores .....                                                | 143 |
| Fig. 5.62 | Gráfica para 16 procesadores .....                                                | 143 |
| Fig. 5.63 | Gráfica para 17 procesadores .....                                                | 143 |
| Fig. 5.64 | Gráfica para 18 procesadores .....                                                | 143 |

---

|           |                                                                                      |     |
|-----------|--------------------------------------------------------------------------------------|-----|
| Fig. 5.65 | Gráfica para 19 procesadores .....                                                   | 143 |
| Fig. 5.66 | Gráfica de procesamiento con 20 procesadores .....                                   | 144 |
| Fig. 5.67 | Gráfica de llamadas a funciones y comunicación. ....                                 | 145 |
| Fig. 5.68 | Llamadas a funciones de charres entre procesadores .....                             | 145 |
| Fig. 5.69 | Llamada tipo <i>Broadcast factores::termina()</i> hacia todos los procesadores ..... | 146 |
| Fig. 5.70 | Porcentaje de carga sobre cada procesador respecto a las funciones. ....             | 146 |
| Fig. 5.71 | Tabla de llamadas a funciones .....                                                  | 147 |
| Fig. 5.72 | Total de mensajes enviados.....                                                      | 147 |
| Fig. 5.73 | Total de mensajes recibidos .....                                                    | 147 |
| Fig. 5.74 | Total de mensajes externos recibidos .....                                           | 147 |
| Fig. 5.75 | Gráfica de comunicación contra tiempo.....                                           | 148 |
| Fig. 5.76 | Animación de la carga de los procesadores.....                                       | 148 |

# Índice De Tablas

|           |                                                                                                             |     |
|-----------|-------------------------------------------------------------------------------------------------------------|-----|
| Tabla 1   | Lista de los clusters que se encuentran en el Top500.....                                                   | 26  |
| Tabla 2   | Tabla de resultados y gráfica de tiempos de ejecución .....                                                 | 115 |
| Tabla 3   | Gráfica de los valores obtenidos de $\pi$ .....                                                             | 115 |
| Tabla 4   | Tabla de tiempos y gráfica comparativa tiempo vs. Procesadores .....                                        | 127 |
| Tabla 5   | Tabla comparativa de número de instrucciones en comparación de la longitud del<br>número a factorizar ..... | 134 |
| Tabla 6   | Tabla de resultados y gráfica de tiempos de ejecución .....                                                 | 141 |
| Tabla 7   | Tabla de tiempos y números que se utilizaron .....                                                          | 155 |
| Tabla 8   | Gráfica de tiempos de ejecución expresada en logaritmo .....                                                | 155 |
| Tabla A.9 | Versiones a elegir para la construcción de un Sistema Cham++.....                                           | 159 |

---

# Resumen

La adquisición por parte de la Dirección de la Facultad de Ciencias de la Computación de un cluster de computadoras, la necesidad de tener herramientas para el uso y aprovechamiento de este tipo de recurso de cómputo de alto rendimiento, y la motivación de conocer más acerca del tema de supercómputo se ven reflejados como parte de los objetivos en el presente trabajo de investigación, acerca de un lenguaje de programación paralela que nos ofrece varias ventajas respecto a otras opciones como MPI, PVM y OpenMP que se encuentran entre los lenguajes más conocidos y utilizados en el área de la programación paralela y distribuida. Antes de adentrarnos en este lenguaje daremos una revisión a conceptos básicos de las Arquitecturas de computadoras como están conformadas, también hablaremos de los Sistemas Distribuidos y Programación Paralela ¿qué son?, ¿para que sirven?, ¿cómo utilizarlos? además del concepto de Cluster de Computadoras, por último mostraremos el objetivo principal de este trabajo: dotar al alumnado de la Facultad con otra herramienta en programación paralela en este caso Charm++ que se presenta como una alternativa nueva con una ventaja principal que es la transparencia de la comunicación entre procesadores tomando como paradigma la programación orientada a objetos, dejando al Sistema de Ejecución de Charm++ la parte de comunicación física.

# Introducción

Actualmente en muchas ramas de las ciencias, la complejidad de los problemas que se estudian requieren cada vez mas el uso de una supercomputadora, siendo estas máquinas poderosas que pueden desarrollar varios miles de millones de operaciones por segundo. Las supercomputadoras tradicionales son máquinas que pueden realizar el procesamiento de tareas de forma paralela, pero tienen un costo muy alto que llega a decenas de millones de dólares.

Regularmente las operaciones para la solución de problemas de esta naturaleza implican, numerosos y repetitivos cálculos matemáticos sobre un conjunto de datos, y usando técnicas de programación paralela pueden ser resueltas en un lapso de tiempo aceptable. El equipo computacional necesario para poder realizar estas tareas son computadoras difíciles de adquirir y de alto costo de mantenimiento ya sea de hardware o software y solo pueden ser costeadas por grandes empresas o instituciones, dejándolas en muchos casos fuera del alcance de universidades y medianas empresas.

Con el avance en la construcción de microprocesadores de bajo costo (como las computadoras de escritorio), es posible obtener una gran capacidad de cómputo en computadoras personales. Sin embargo no es suficiente para resolver problemas de gran complejidad, ya que exige un alto grado de procesamiento que excede las prestaciones de estos equipos.

En la década de los años 90, varios proyectos de investigación realizados por diversas universidades y centros de investigación se dedicaron a crear un conjunto de máquinas o cluster de máquinas, interconectadas a través de una Red de Área Local (LAN), que actuara como una sola máquina con suficiente capacidad y recursos para poder resolver problemas complejos, creando así sus propias supercomputadoras con un presupuesto bajo y en numerosos casos aprovechando el equipo que se tenía como obsoleto.

Durante varios años la computación secuencial ha sido el modo habitual de la programación. Y esto se debe a la adopción del modelo propuesto por Von Neumann, cuya estabilidad ha sido la base para el desarrollo de diversas aplicaciones. Sin embargo, desde los primeros días de la computación secuencial se tenía la noción que, tarde o temprano, se tendría que recurrir a la computación paralela para resolver problemas que demandaban

mucho poder de cómputo que en máquinas secuenciales no sería óptimo o posible de realizar.

El cómputo paralelo utiliza más de una unidad de procesamiento al mismo tiempo para resolver un problema. El procesamiento paralelo consiste en resolver un problema dividiendo éste en partes más pequeñas y procesándolas de manera simultánea, esto significa que un programa se puede ejecutar más rápidamente porque existen más de un procesador que trabajan al mismo tiempo y de manera coordinada.

Las aplicaciones de computación paralela se pueden encontrar en una amplia gama de disciplinas, tales como la predicción del clima, exploración petrolera, procesamiento de imágenes, fusión nuclear, modelado de escenarios, como el océano o la biosfera o recientemente del genoma humano; en general varios tipos de simulaciones, y últimamente la edición de películas hechas por computadoras en específico, el renderizado de éstas ya que requieren mucho tiempo de procesamiento.

La adquisición por parte de la Facultad de Ciencias de la Computación de un cluster de computadoras, y la necesidad de tener herramientas para el uso y aprovechamiento de éste tipo de recurso de cómputo de alto rendimiento es el propósito principal de este trabajo, además de la falta de documentos para el uso de las herramientas que este disponible para los alumnos de la facultad y, la motivación de conocer más acerca del tema de supercómputo.

En el presente documento se revisarán conceptos como el de cluster de computadoras, un poco de su historia y como han llegado a ser una herramienta indispensable no solo en el campo de la computación si no en varias disciplinas, para obtener resultados en ocasiones de manera más rápida en otras lo que se requiere es mayor precisión.

Se revisarán los tipos de arquitecturas de computadoras, de cómo éstas se han visto modificadas para adaptarse al avance tecnológico y las necesidades actuales de cómputo.

Se explicará el concepto de paralelismo y qué es la programación paralela para conocer la forma en que un programa debe ser paralelizable, con el fin de saber cuando es necesario realizar un programa de manera paralela de manera eficiente.

Por último se darán a conocer las herramientas primero, de programación paralela *Charm++* para la programación de aplicaciones paralelas, sus ventajas y desventajas respecto a otras herramientas como MPI y PVM que son las de mayor difusión; y de *Projections* que nos servirá para el monitoreo de aplicaciones y conocer qué sucede cuando un programa se ejecuta en paralelo. La explicación de ejemplos que nos ayuden a reforzar la parte teórica y nos de un mejor panorama de cómo realizar una aplicación con Charm++. Y por último las conclusiones a las que hemos llegado tras este trabajo de investigación.

# Capítulo 1

## Arquitectura De Computadoras

Vamos a definir como una *arquitectura de computadoras* al diseño conceptual y la estructura operacional fundamental del sistema. Es decir, es un modelo y una descripción funcional de los requerimientos y las implementaciones de diseño para las partes que conforman una computadora, centrando la atención en la forma en que la unidad central de proceso trabaja internamente y accede a la memoria. También suele definirse como la forma de seleccionar e interconectar componentes de hardware para crear computadoras según los requerimientos de funcionalidad, rendimiento y costo.

La arquitectura John Von Neumann es la más usada en la actualidad en las computadoras, pero no la única, de acuerdo con las necesidades han ido apareciendo nuevos tipos de arquitecturas. En este capítulo revisaremos algunas importantes.

### 1.1 Arquitectura de Von Neumann

*Von Neumann* estableció una estructura para una arquitectura de computadora que utiliza el mismo dispositivo de almacenamiento tanto para las instrucciones como para los datos a diferencia de otra arquitectura que existía en su época la de Harvard. Esta arquitectura fue establecida en el *documento First Draft of a Report on the EDVAC* en 1945 por John Von Neumann, en vista a la construcción del sucesor de la computadora *ENIAC*. Algo poco conocido es que su contenido fue desarrollado por Presper Eckert, John Mauchly, Arthur Burks, y otros investigadores varios meses antes de que Von Neumann redactara el borrador del informe.

La mayoría de computadoras modernas están basadas en ésta arquitectura, cada computadora personal, microcomputadora, minicomputadora y supercomputadora es una máquina de Von Neumann en la forma de procesar la información; esto es, una instrucción o tarea debe realizarse completamente antes de pasar a la siguiente. Este tipo de arquitectura utiliza el mismo dispositivo de almacenamiento para instrucciones y para datos.

A continuación presentamos un diagrama de cómo están conformadas las computadoras con arquitectura de Von Neumann:

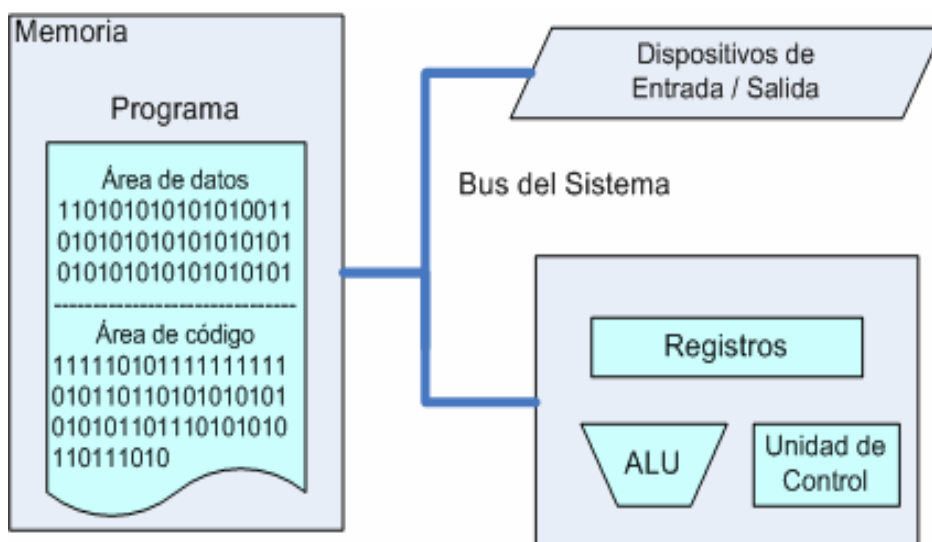


**Fig. 1.1 Diagrama de la Arquitectura de Von Newmann**

Una computadora con la arquitectura Von Neumann realiza los siguientes pasos de forma secuencial:

- Obtiene la siguiente instrucción desde la memoria en la dirección indicada por el contador de programa y la guarda en el registro de instrucción.
- Aumenta el contador de programa en la longitud de la instrucción para apuntar a la siguiente.
- Define qué va a realizar la instrucción mediante la unidad de control. Ésta se encarga de coordinar el resto de componentes de la computadora para realizar una función determinada.
- Se ejecuta la instrucción. Ésta puede cambiar el valor del contador del programa, permitiendo así operaciones repetitivas. El contador puede cambiar también cuando se cumpla una cierta condición aritmética, haciendo que la computadora “pueda tomar decisiones”, que pueden alcanzar cualquier grado de complejidad, mediante la aritmética y lógica anteriores.
- Regresa al primer paso.

## 1.2 La Arquitectura Harvard

La arquitectura Harvard que apareció casi al mismo tiempo que la de Von Neumann en 1944 es otra forma de modelar hardware en una computadora programable, ésta separa el

almacenamiento y el camino de las señales eléctricas en donde viajan las instrucciones y los datos: el bus. El término se origina de la computadora **Harvard Mark I**. Esta computadora poseía una unidad de almacenamiento pequeña contenida en su totalidad en la unidad de procesamiento, es decir, tenía algo similar a un disco duro dentro del CPU, como la memoria cache actual.

Así que, para cambiar el programa a ejecutar o el grupo de instrucciones, para ser más precisos, debían de apagar a la Mark, lo cual era tedioso. En contraste con una arquitectura de Von Neumann, la arquitectura Harvard es definitivamente de mayor velocidad, pero no era conveniente a la hora de ejecutar varios grupos de instrucciones distintas. Los cálculos se realizaban con una cinta de papel perforada y una serie de interruptores accionados manualmente y con paneles de control con conexiones especiales. Un ejemplo de computadoras programables que usan la arquitectura Harvard, es el famoso microcontrolador **PIC**.

### 1.3 Clasificación de Flynn

Michael Flynn propuso en 1966 un mecanismo de clasificación para las computadoras. La clasificación de Flynn es una forma clásica de organizarlas, y aunque no abarca todas las posibles arquitecturas, cubre una importante variedad. El método de Flynn se basa en el número de instrucciones y de la secuencia de datos que la computadora utiliza para procesar información. Puede haber secuencias de instrucciones sencillas o múltiples y secuencias de datos sencillas o múltiples. Esto da lugar a 4 tipos de computadoras, de las cuales solamente dos son aplicables a las computadoras paralelas.

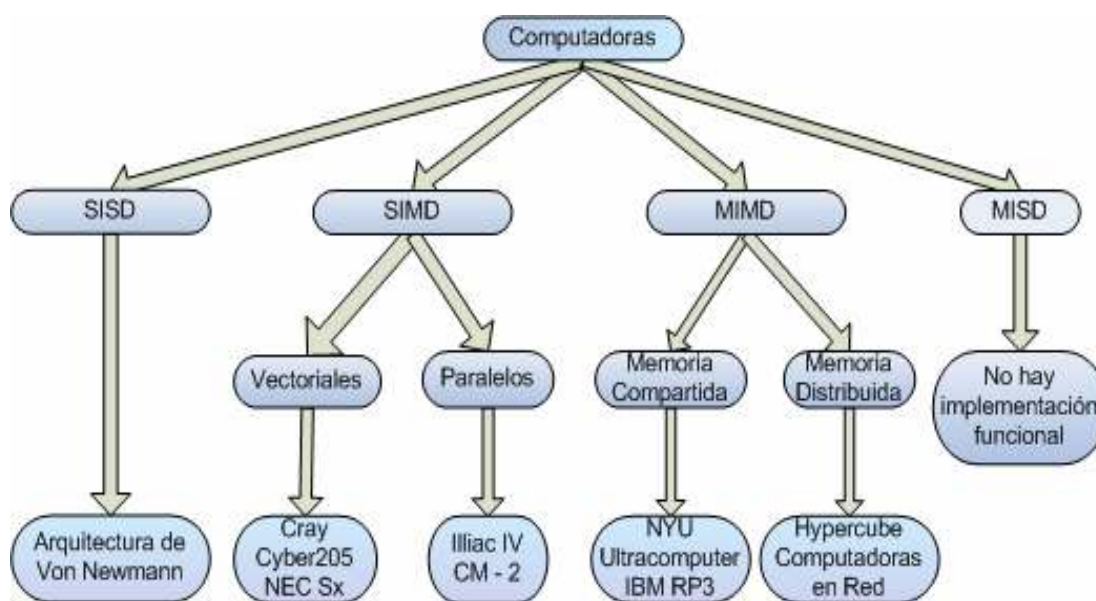


Fig. 1.2 Taxonomía de Flynn

### 1.3.1 SISD (Single Instruction Single Data)

Es el modelo más sencillo y tradicional de computación donde se ejecuta una sola instrucción por ciclo de reloj. La unidad de procesamiento recibe una sola secuencia de instrucciones que operan sobre una secuencia de datos. Los algoritmos para las computadoras SISD no contienen paralelismo alguno, por que están constituidas de un solo procesador.

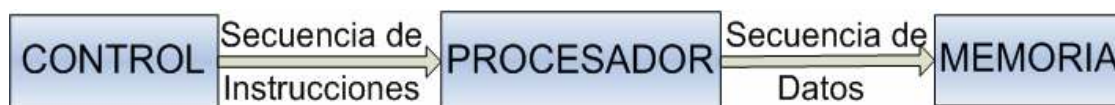


Fig. 1.3 Modelo SISD (Single Instruction Single Data).

### 1.3.2 SIMD (Single Instruction Multiple Data)

A diferencia del modelo *SISD* en esta clase se tienen múltiples procesadores, que de forma sincronizada ejecutan la misma secuencia de instrucciones, pero sobre diferentes datos. El tipo de memoria que estos sistemas utilizan es distribuida y en este modelo hay *N* **secuencias de datos**, una por procesador y en cada paso todos los procesadores ejecutan la misma instrucción, cada uno sobre diferentes datos.

Las máquinas con arreglos de procesadores como ICL DAP (Distributed Array Processor) y computadoras vectoriales canalizadas como CRAY 1 & 2 y CIBER 205 tienen una arquitectura *SIMD*.



Fig. 1.4 Modelo SIMD (Single Instruction Multiple Data).

#### 1.3.2.1 Computadora Vectorial

Un procesador vectorial o arreglo de procesadores, es un diseño de procesador donde se puede aplicar una instrucción u operación sobre un vector de datos, es decir, de manera

simultanea se realiza una operación matemática sobre múltiples datos, en contraste con los procesadores escalares que solo pueden tomar un dato a la vez.

Una computadora vectorial contiene un conjunto de unidades aritméticas especiales denominadas pipelines. Una pipeline son unidades de cálculo especializadas para realizar alguna operación sobre algún dato, así si un procesador tiene dos pipelines puede ejecutar dos instrucciones en un paso, duplicando (teóricamente) su rendimiento. Estas pipelines ejecutan operaciones aritméticas sobre los elementos de un vector, produciendo una ejecución más eficiente.

Algunas computadoras vectoriales, como la Cray Y-MP, contienen registros vectoriales. Un registro contiene un único valor. Los registros vectoriales pueden contener muchos elementos de un vector al mismo tiempo. Por ejemplo, los registros vectoriales de la Cray Y-MP contienen 64 elementos, mientras que los de la Cray C90 contienen 128 elementos. Los contenidos de estos registros pueden ser enviados a (o recibidos por) una pipeline vectorial en un paso. Si un procesador vectorial posee registros vectoriales, los elementos del vector que se van a procesar se cargan directamente en el registro vectorial. El vector que se obtiene a partir de una operación vectorial se introduce en un registro vectorial antes de que se pueda almacenar de nuevo en la memoria mediante una operación de almacenamiento vectorial. Esto permite que se pueda utilizar en otra operación sin necesidad de volver a leer el vector, y también permite que el almacenamiento se pueda solapar con otro tipo de operaciones. En este tipo de computadoras, todas las operaciones aritméticas ó lógicas vectoriales son operaciones registro a registro, es decir, sólo se realizan operaciones vectoriales sobre vectores que ya se encuentran almacenados en los registros vectoriales. Por este motivo, a estas computadoras se les conoce como procesadores vectoriales de registro vectorial.

Por ejemplo, para la suma de dos matrices A y B. Siendo A y B de orden 2 y teniendo 4 procesadores la suma es de la siguiente forma:

$$A+B = \begin{bmatrix} A_{11} + B_{11} = C_{11} & A_{21} + B_{21} = C_{21} \\ A_{12} + B_{12} = C_{12} & A_{22} + B_{22} = C_{22} \end{bmatrix}$$

La misma instrucción es ejecutada en los 4 procesadores (sumando dos números) y los 4 ejecutan las instrucciones simultáneamente. Esto toma un solo paso en comparación de los cuatro pasos que le toma a una máquina secuencial.

### 1.3.2.2 Procesadores vectoriales memoria a memoria

Este tipo de procesadores vectoriales permite que operaciones realizadas con vectores se tomen directamente los datos de la memoria hasta los pipelines vectoriales y que los resultados se escriban directamente en la memoria. Los elementos del vector se toman de la memoria en lugar de proceder de un registro, se requiere más tiempo para conseguir que la

operación vectorial comience a realizarse. Esto es debido en parte al coste del acceso a la memoria. Un ejemplo de procesador vectorial memoria a memoria era el CDC Cyber 205.

Debido a la capacidad de superponer el acceso a la memoria y la posible reutilización de los vectores ya utilizados, los procesadores vectoriales de registro vectorial suelen ser más eficientes que los procesadores vectoriales memoria a memoria. Sin embargo, a medida que la longitud de los vectores utilizados para un cálculo se incrementa, esta diferencia en el rendimiento entre los dos tipos de arquitecturas tiende a desaparecer. De hecho, los procesadores vectoriales memoria a memoria pueden llegar a ser más eficientes si la longitud de los vectores es muy grande. Sin embargo, la experiencia ha demostrado que la longitud de los vectores suele ser mucho más corta de la necesaria para que esta situación llegue a producirse.

### 1.3.3 MIMD (Multiple Instruction Multiple Data)

En este modelo a diferencia del **SISD** es asíncrono. Cada procesador en un sistema **MIMD** puede ejecutar su propia secuencia de instrucciones y tener sus propios datos, no tiene un reloj central, así que los procesadores operan asincrónicamente, o en términos simples, pueden estar haciendo diferentes cosas con diferentes datos al mismo tiempo. Esta característica es la más general y poderosa de esta clasificación.

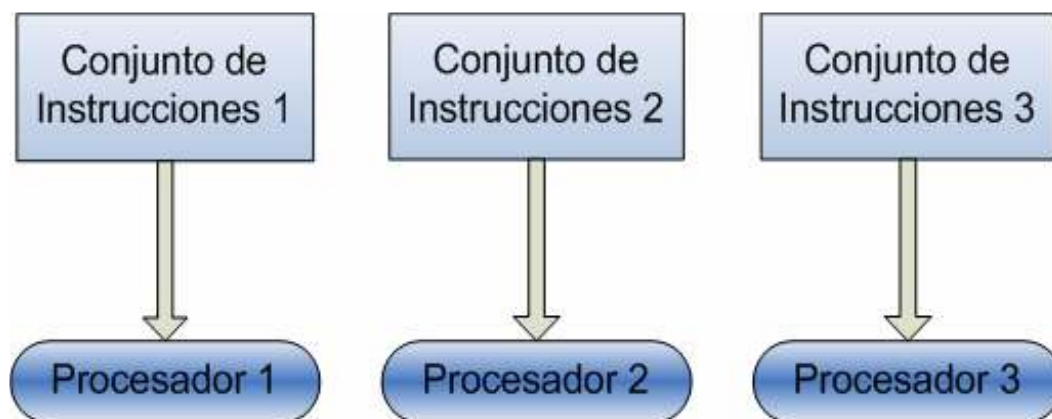


Fig. 1.5 Modelo MIMD (Single Instruction Multiple Data).

Los sistemas MIMD se subdividen a su vez en:

- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.
- Sistemas de Memoria Compartida Distribuida.

### 1.3.3.1 Sistemas de Memoria Compartida

En este tipo de sistemas, cada procesador tiene acceso a toda la memoria que existe. Se tienen tiempos de acceso a memoria uniformes ya que todos los procesadores se encuentran igualmente comunicados con la memoria principal, las lecturas y escrituras de todos los procesadores tienen exactamente la misma latencia; y además el acceso a memoria es por medio de un canal común.

Ventajas:

- La facilidad de la programación en estos sistemas ya que todos los datos son visibles para todos los procesadores.

Desventajas:

- Es difícil el acceso simultáneo a memoria.
- Poca escalabilidad de procesadores, debido a que se puede generar un cuello de botella al incrementar el número de procesadores.
- La razón principal de su alto precio es la memoria.

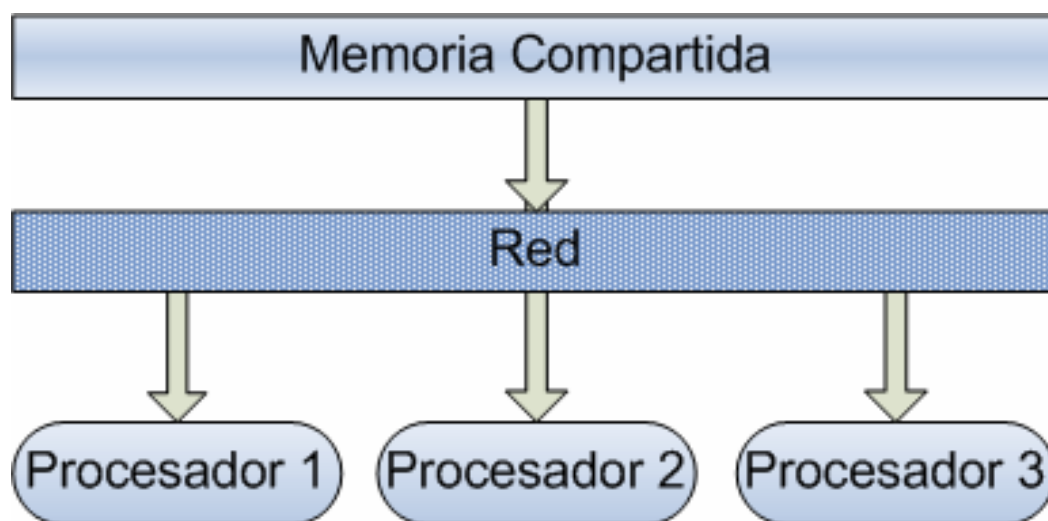


Fig. 1.6 Sistemas de Memoria Compartida.

Las computadoras MIMD con memoria compartida son sistemas conocidos como de multiprocesamiento simétrico (SMP). Ejemplos de computadoras que siguen este modelo son: SGI/Cray Power Challenge, SGI/Cray C90, SGI/Onyx, ENCORE, MULTIMAX, SEQUENT y BALANCE, entre otras.

### 1.3.3.1.1 Sistema UMA (Uniform Memory Access)

UMA es un sistema multiprocesador donde la memoria física es uniformemente compartida por todos los procesadores, esto quiere decir que todos los procesadores tienen el mismo tiempo de acceso a todas las partes de la memoria y cada procesador tiene su propia caché privada y además de compartir los periféricos. Los sistemas multiprocesador son sistemas fuertemente acoplados, debido al alto grado en que comparten recursos ya sea hardware o software y al alto nivel de interacción entre procesadores, lo que hace que un procesador dependa de lo que haga otro.

El sistema de interconexión debe ser rápido y puede ser de uno de los siguientes tipos:

- *Bus común*: Bus compartido por varias computadoras.
- *Red crossbar*: Todos los nodos están conectados con todos, de manera que cualquier nodo puede comunicarse con cualquier otro nodo en un único paso.
- *Red multietapa*: Se suele hablar de redes multietapa cuando se utilizan switches centralizados.

Este modelo es conveniente para aplicaciones de propósito general y de tiempo compartido por varios usuarios.

Dentro de los sistemas UMA existen también categorías:

**Sistema Simétrico**: Cuando todos los procesadores tienen el mismo tiempo de acceso a todos los componentes del sistema (incluidos los periféricos), reciben el nombre de sistemas multiprocesador simétrico. Los procesadores tienen la misma prioridad sobre los periféricos y cada procesador tiene la misma capacidad para procesar.

**Sistema Asimétrico**: Son sistemas con procesadores maestros y procesadores esclavos, en donde sólo los primeros pueden ejecutar aplicaciones y el tiempo de acceso para diferentes procesadores no es el mismo. Los procesadores esclavos ejecutan el código del usuario bajo la supervisión del maestro, por lo tanto cuando una aplicación es ejecutada en un procesador maestro dispondrá de una cierta prioridad.

### 1.3.3.2 Sistemas de Memoria Distribuida.

En estos sistemas se cuenta con memoria propia local y los procesadores pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos.

Esta comunicación se le conoce con el nombre de *Paso de Mensajes*.

Ventajas:

- La escalabilidad. Mientras que la demanda de los recursos crece, se puede agregar más memoria y procesadores.

Desventajas:

- El acceso remoto a memoria es lento.
- La programación puede ser complicada, a causa de que hay que conocer en donde se encuentra la información requerida.

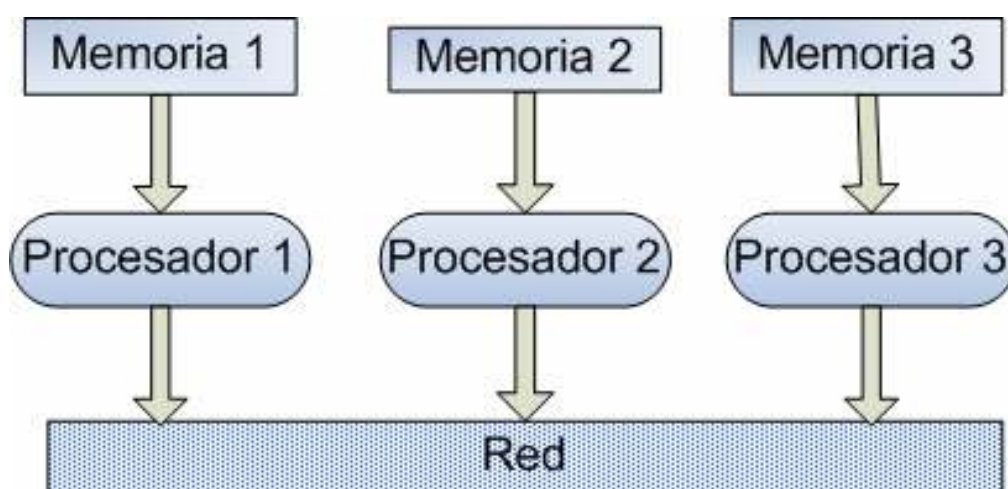


Fig. 1.7 Sistemas de Memoria Distribuida.

Las computadoras MIMD de memoria distribuida son conocidas como sistemas de procesamiento en paralelo masivo (MPP), donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. En esta clasificación entran los clusters de computadoras. Algunos ejemplos de máquinas con este sistema son IBM SP2 y SGI/Cray T3D/T3E.

### 1.3.3.3 Sistemas de Memoria Compartida Distribuida

Este sistema es un conjunto de procesadores que tienen acceso a una memoria compartida común, pero sin un canal compartido. Cada procesador posee su memoria local y se comunica con otros por medio de un dispositivo de alta velocidad, así, todos ven la memoria de cada uno como un espacio de direcciones global.

El acceso a la memoria de diferentes clusters se realiza bajo el esquema de *Acceso a Memoria No Uniforme (NUMA)*, la cual toma menos tiempo en acceder a la memoria local de un procesador que acceder a memoria remota de otro procesador.

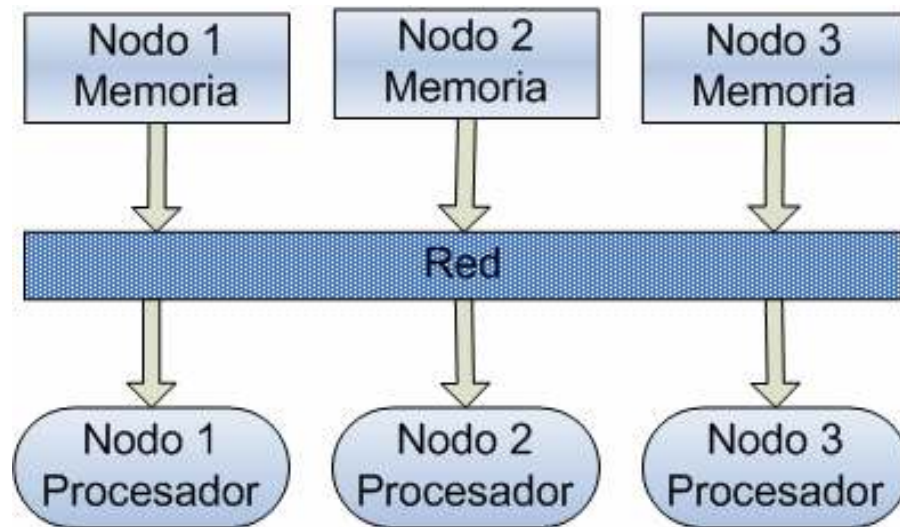


Fig. 1.8 Sistemas de Memoria Compartida Distribuida.

Ventajas:

- Presenta escalabilidad como en los sistemas de memoria distribuida.
- Es fácil de programar como en los sistemas de memoria compartida.
- No existe el cuello de botella.

Desventajas:

- La topología de red.
- La administración de la red.

#### 1.3.3.3.1 Sistema NUMA (Non Uniform Memory Access)

Un sistema multiprocesador NUMA es un sistema de memoria compartida donde el tiempo de acceso depende de donde se encuentre localizado el espacio de memoria a al cual se desea tener acceso.

El acceso a memoria, por tanto, no es uniforme para los diferentes procesadores. Existen memorias locales asociadas a cada procesador y estos tienen acceso a datos de su memoria local más rápido que a las memorias de otros procesadores, debido a que primero debe ser aceptado el acceso por el procesador al cual le corresponde ese módulo de memoria local.

Todas las memorias locales conforman una memoria global compartida, físicamente distribuida y accesible por todos los procesadores.

### 1.3.3.3.2 CC-NUMA (Cache Coherent Nonuniform Memory Access)

La finalidad de los sistemas CC-NUMA es reducir el tiempo de acceso a los datos que se encuentran en otro nodo empleando una cache de nivel 3 (**L3**) llamada cache remota o simplemente **RC** en la que almacena los datos que haya solicitado a otros nodos.

Al igual que el Sistema NUMA tienen su memoria físicamente distribuida en cada nodo. En estas máquinas, cada nodo posee una parte del espacio compartido de direcciones y una memoria cache que puede utilizarse para copiar datos que se encuentran en otro nodo. Debido a que la memoria física está uniformemente repartida sobre los nodos y su tamaño es igual al de la memoria direccionable, analizando una dirección determinar es posible conocer en que nodo se encuentra el bloque de memoria que contiene el dato.

Cuando se produce algún fallo en la cache de un nodo, la información guardada se solicita a la memoria local de ese nodo o a la memoria cache de otro nodo, de acuerdo con su dirección física. Si este dato en la cache es modificado, debe de actualizarse en la memoria del nodo al que pertenece ese dato. De esto tenemos que la cantidad de datos que pueden copiarse está limitado al tamaño de la cache (SRAM) en cada nodo. Una desventaja de los sistemas CC-NUMA es que el rendimiento depende de la distribución inicial de los datos en las diferentes memorias cache locales, ya que su distribución incorrecta supone más solicitudes remotas que locales. Sucede lo mismo si cada nodo intenta tener acceso a subconjuntos de datos de diferente tamaño.

Existen diferentes implementaciones de esta arquitectura. Un ejemplo es el sistema DASH, que utiliza una arquitectura a dos niveles: en el nivel superior, cada nodo de proceso está conectado a una red de interconexión de tipo malla. Dentro de cada cluster de utiliza un topología basada en bus para conectar los procesadores.

Otra implementación de los sistemas CC-NUMA son los multiprocesadores basados en la arquitectura Opteron de AMD los cuales no necesitan algún controlador externo en dicha implementación.

En los últimos años han aparecido nuevas técnicas para reducir la latencia de los sistemas CC-NUMA, entre las que pueden nombrarse: el uso de mecanismos de encaminamiento adaptativo y de canales virtuales, el uso de procesadores de protocolo en lugar de controladores hardware, y la utilización de sistemas de migración dinámica de página que tenga soporte a nivel de sistema operativo.

### 1.3.3.3.3 Cluster Jerárquico

Otro modelo NUMA que nace como una mezcla de los modelos UMA y NUMA, es el cluster jerárquico en el que se combinan las memorias locales y las globales obteniendo una cierta escalabilidad del sistema.

Los procesadores aparecen distribuidos en clusters (1 sistema UMA o un 1 sistema NUMA). Estos clusters están conectados a la memoria global compartida. El sistema en su

totalidad es un sistema NUMA, ya que el acceso a memoria es no uniforme por parte de los clusters.

La ventaja de estos sistemas con respecto a los sistemas UMA, es que el acceso a memoria local es mucho más rápido.

#### 1.3.3.4 COMA (Cache Only Memory Access)

Los sistemas COMA son un caso especial de los sistemas NUMA. Este tipo de sistemas no ha tenido mucha trascendencia, al igual que los sistemas SIMD.

Las memorias distribuidas son memorias cache, por este motivo es un sistema muy restringido en cuanto a la capacidad de memoria global. No hay jerarquía de memoria en cada módulo procesador. Todas las caches forman un mismo espacio global de direcciones. El acceso a la cache remota se realiza a través de directorios distribuidos.

#### 1.3.4 MISD (Multiple Instruction Single Data)

En este modelo, una secuencia de instrucciones pasan a través de múltiples procesadores.  $N$  procesadores, cada uno con su propia unidad de control comparten una memoria común. El paralelismo es alcanzado dejando que los procesadores realicen diferentes cosas al mismo tiempo sobre los mismos datos.

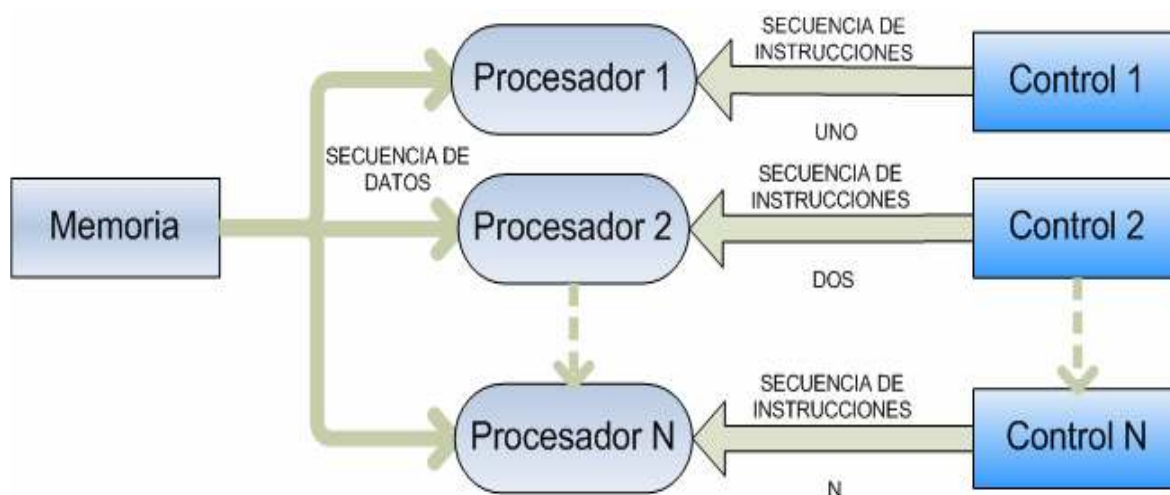


Fig. 1.9 MISD (Multiple Instruction Single Data).

Las máquinas MISD son útiles en cálculos donde la misma entrada puede estar sujeta a diferentes operaciones. Actualmente no hay alguna máquina que esté dentro de esta categoría.

# Capítulo 2

## Clusters De Computadoras

Los *mainframes* y las *supercomputadoras* han sido las principales herramientas para el procesamiento masivo durante los varios años pero que, además son construidas solamente por unos cuantos fabricantes y para un grupo muy reducido de usuarios que necesitan gran potencia de cálculo, como varias empresas, gobiernos y algunas universidades. Pero existen otros grupos que no pueden hacer frente al costo económico que supone adquirir una máquina de estas características, es aquí donde toma importancia la idea de poder disponer de gran potencia de cálculo, pero a un bajo precio. Así nació el concepto de cluster, cuando los pioneros de la supercomputación intentaban repartir diferentes procesos entre varias computadoras, para luego poder recoger los resultados que dichos procesos habían de producir. Con un hardware fácil de conseguir (en costo) se tuvo la idea que podrían obtenerse resultados en tiempos muy parecidos a los obtenidos con aquellas máquinas que suponen una mayor inversión, como se ha venido probando desde aquel entonces.

### 2.1 Definición de cluster

*Cluster* (o también castellanizado como *clúster*) es un término inglés utilizado y estudiado en diferentes áreas técnicas en la actualidad. Su traducción literal es *racimo* o *grupo*, básicamente se refiere a todas las entidades que pueden trabajar en conjunto para lograr un fin específico en un algún ámbito.

Existen diferentes tipos de Cluster, según el área en que se desarrollen, entre los que podemos mencionar:

- Cluster de empresas
- Cluster de música (acordes)
- Clusters de peces

- Cluster de computadoras
- Cluster de sistemas de archivos
- Cluster, misión de la Agencia Espacial Europea (ESA)
- Clusters industriales

De esta definición tenemos que un cluster de computadoras es un conjunto de computadoras que pueden trabajar de manera conjunta y coordinada en la solución de un mismo problema, y desde otro punto de vista, también puede considerarse como un tipo particular de computadora paralela. Este término puede aplicar tanto a computadoras de alto rendimiento construidas para este propósito como a un grupo de simples computadoras de escritorio. Un cluster solo define un grupo de computadoras y estas pueden tener cualquier característica lo único que deben cumplir es tener una interconexión y una interacción para poder realizar alguna tarea en conjunto y sea visto como una única computadora, más potente que las comunes de escritorio. De manera más simple, cluster es un grupo de computadoras unidas mediante una red, de tal forma que el conjunto es visto como una sola computadora con grandes prestaciones. Los clusters son la principal alternativa para el alto desempeño en computación llamado High Performance Computing (HPC) y juegan hoy en día un papel importante en la solución de problemas de la ciencia, la ingeniería y hasta del comercio moderno.

Para construir un cluster no es necesaria la intervención de personal altamente especializado de alguna compañía de computadoras, esto debido a que prácticamente todos los clusters han tenido un carácter académico aunque no necesariamente en su uso, pero sí en el proceso de configuración, implementación y operación; por ejemplo, *Beowulf* que fue construido por Donald Becker y Thomas Sterling, investigadores de la NASA y el Tecnológico de California, respectivamente; mientras que el proyecto *Cluster X* (también conocido como *Terascale Cluster*) fue dirigido por Srinidhi Varadarajan y Jason Lockhar, ambos académicos del Tecnológico de Virginia.

Los cluster han empezado a ser usados en áreas donde los **mainframes** o **supercomputadoras** eran utilizadas en el pasado, con el tiempo han pasado a formar parte significativa en el avance de varias áreas como:

- Investigación científica
- Investigación Militar
- La aeronáutica
- La industria automotriz
- La industria de la construcción
- Investigación sobre mecánica del suelo
- La química
- Efectos visuales y entretenimiento, cine y televisión

- Internet
- Y varios tipos de simulaciones como la de los mercados de valores

O en algunas investigaciones específicas, como los son:

- Identificar nuevas partículas sub-atómicas
- Simulación de las corrientes marinas
- Simulación de eventos astronómicos
- Investigación para descubrir una droga para el SIDA
- Exploración de recursos de petróleo y gas
- Investigación del genoma humano
- Simuladores de vehículos de guerra y entrenamiento militar

## 2.2 Historia de los Cluster de Computadoras

En el verano de 1994 se presentó Beowulf, el primer cluster ampliamente conocido por la comunidad de cómputo científico. Sterling y Don Becker, trabajando para el CESDIS (Center of Excellence in Space Data and Information Sciences) bajo el patrocinio del Proyecto de las Ciencias de la Tierra y el Espacio (ESS) de la NASA, construyeron un Cluster de Computadoras que consistía en 16 computadoras personales con procesadores Intel DX4 (486) a 200 MHz, conectadas mediante un switch Ethernet con un ancho de banda de 10 Mbps; tenía un rendimiento teórico de 3.2 GigaFLOPS (miles de millones de operaciones de punto flotante por segundo) y fue utilizada como una **computadora paralela personal**; el sistema operativo se instalaba manualmente en cada máquina y el usuario debía contar con experiencia en el uso de herramientas para iniciar procesos en paralelo. La máquina fue un éxito inmediato y su idea de proporcionar sistemas basados en equipos que en cierta forma tenían abandonados, para satisfacer requisitos de cómputo específicos y se propagó rápidamente a través de la NASA y en las comunidades académicas y de investigación.

El siguiente cluster importante fue *Avalon*, construido en el **Laboratorio Nacional de Los Álamos**, entre 1997 y 1998; fue el primero en entrar a la lista de las 500 computadoras más potentes del mundo ([www.top500.org](http://www.top500.org)) donde ocupó el **lugar 315** de la lista en el primer semestre de 1998, hecho de gran importancia, pues comprobó que los clusters, además de ser baratos, podían competir en rendimiento, por lo menos en algunas aplicaciones, con las computadoras paralelas de los fabricantes más reconocidos del mundo. Para *Avalon* se conectaron 70 computadoras a través de un switch Fast Ethernet de 100 Mbps, cada una con un procesador Alpha a 533 MHz, 128 MBytes de RAM y 3.2 GBytes de disco duro (casi 9 GBytes de RAM y más de 200 GBytes de almacenamiento en total). Posteriormente creció a 140 procesadores y llegó a ocupar el **lugar 113** en el **Top500** durante el segundo semestre de 1998.

*Cplant* (*Computational plant*), construido en el *Laboratorio Nacional Sandia*, dio un paso importante en la evolución de los clusters. *Cplant* es todo un proyecto que contempla la manera en que un equipo de esta naturaleza puede crecer y renovarse de manera continua. La primera versión del Cplant, que entró al *Top500* en el primer semestre de 1999, contaba con 150 procesadores Alpha conectados a través de un switch *Myrinet*; sin embargo, el paso importante se dio en el segundo semestre del mismo año cuando *Cplant* pasó de 150 a 580 procesadores, para ocupar el *lugar 44* del *Top500*.

Para utilizar de forma eficiente un número tan grande de procesadores se adaptaron sistemas de administración de recursos propios de otras máquinas masivamente paralelas, como la Intel Paragon. Durante los últimos años, *Cplant* llegó hasta los 1590 procesadores.

Otro cluster relevante es *Score*, instalado en el Centro de Investigación de Tsukuba, ya que fue el primero en contar con más de 1000 procesadores (1024) lo que lo llevó a ocupar el *lugar 36* en el Top500 del primer semestre de 2001.

En la actualidad la mayoría de las máquinas listadas son *clusters*, en la última lista de Junio de 2009, el cluster *Roadrunner* del departamento de defensa de los Estados Unidos, ocupa el *lugar número 1* en la lista del *Top500*. Este cluster esta compuesto de *12.960 procesadores Cell de 8 núcleos* (1 powerPC+7SPE) y *6.480 procesadores AMD Opteron de doble núcleo*, totalizando 116.640 núcleos, *51,8 TB de memoria RAM* y una *red Voltaire Infiniband*. El sistema operativo que utiliza esta máquina es RedHat Enterprise Linux, y administrada con *xCAT*, un software para gestionar entornos de supercomputación. Fue creado por IBM en conjunto con Laboratorio Nacional de Los Álamos, laboratorio del Departamento de Energía de los Estados Unidos en Nuevo México.

Recientemente para la creación de clusters se han usado otro tipo de máquinas no solo con computadoras personales, en los Estados Unidos el Dr. Frank Mueller, profesor asociado de Ciencia Informática en la Universidad del estado de Carolina del Norte creo un cluster con *8 consolas de videojuego Playstation 3*. Según el Dr. Mueller, cada *PS3* contiene 6 núcleos operacionales especializados en procesamiento de números y un núcleo general, que resultaron en 64 procesadores. El resultado se traduce en una alta capacidad de procesamiento de datos.

El cluster del Dr. Mueller logra el poder de una pequeña supercomputadora a un costo de unos US \$5,000, lo que es menor al precio de algunas estaciones de trabajo actuales con grandes prestaciones, pero que a diferencia de este cluster, sólo se compara como una fracción todo del poder procesamiento de las consolas en conjunto.

Aunque la Playstation 3 es muy buena en el procesamiento de números, algo esencial de la ciencia informática, el Dr. Mueller reconoce que al momento están limitados por los 256 MB de memoria RAM de las máquinas. El cluster de PS3 creado por el Dr. Mueller, aunque realmente poderoso, no alcanza a entrar en la lista *top500*, sin embargo el Dr. estima que con unas 10.000 consolas *PS3* cualquiera podría crear la computadora más rápida del mundo (claro esta con ciertas limitaciones de red y precisión).

## 2.3 Clasificación de un cluster

Un cluster de computadoras puede ser clasificado por su aplicación, por infraestructura o características. Entre los tipos que podemos encontrar regularmente se encuentran los siguientes:

**Alta disponibilidad (Fail-over o High-Availability):** este tipo de cluster esta diseñado para mantener uno o varios servicios disponibles incluso a costa del rendimiento, ya que su objetivo principal es que el servicio nunca tenga interrupciones como por ejemplo una base de datos.

Si se produce un fallo del hardware o de las aplicaciones de alguna de las máquinas que componen el cluster, el software debe ser capaz de arrancar automáticamente los servicios que han fallado en cualquiera de las otras máquinas del cluster. Y cuando la máquina que ha fallado se recupera, los servicios son nuevamente migrados a la máquina original. Esta capacidad de recuperación automática de servicios nos garantiza la integridad de la información, ya que no hay pérdida de datos, y además es transparente para los usuarios, que no tienen por qué notar cuando se ha producido una falla.

No hay que confundir un cluster de alta disponibilidad con un cluster de alto rendimiento. El segundo es una configuración de equipos diseñado para proporcionar capacidades de cálculo mucho mayores que la que proporcionan los equipos individuales (lo describiremos más adelante), mientras que el primer tipo de cluster está diseñado para garantizar el funcionamiento ininterrumpido de ciertas aplicaciones. En un sistema real, si falla uno de los componentes, es reparado o sustituido por un nuevo componente.

**Alto Rendimiento (HPC o High Performance Computing):** este tipo de cluster está diseñado para obtener el máximo rendimiento de la aplicación utilizada incluso a costa de la disponibilidad del sistema, es decir el cluster puede sufrir caídas, este tipo de configuración esta orientada a procesos que requieran mucha capacidad de cálculo.

**Balance de Carga (Load-balancing):** este tipo de cluster esta diseñado para equilibrar la carga de trabajo entre varios nodos, esto permite tener, por ejemplo, un servidor sin caídas a causa de una carga excesiva de peticiones en un momento específico (excepto que se sobrepase la capacidad de todas las máquinas en conjunto). Actualmente un cluster del tipo Balance de Carga (Load balancing) también es de alta Disponibilidad (Fail-Over) y regularmente con número de nodos muy grande. Los clusters Beowulf entran en este tipo por que han sido diseñados específicamente para tareas de tipo masivo.

**Alta escalabilidad:** la escalabilidad es la capacidad de un equipo para hacer frente a volúmenes de trabajo cada vez mayores sin, por ello, dejar de prestar un nivel de rendimiento aceptable. Este tipo de clusters ha sido concebido para amoldarse o adaptarse en todo momento, y con costo mínimo, a necesidades que pueden ser variables o cambiantes.

La *modularidad* es un elemento básico de este tipo de clusters para lograr la alta escalabilidad, permitiendo el crecimiento de la capacidad del servicio al añadir más nodos al

cluster o elementos particulares como disco o memoria. Por otra parte, un sistema no escalable requeriría renovar completamente la arquitectura que estuviera actualmente en uso lo que se traduce en un costo igual o mayor al del sistema actual.

Existen dos tipos de escalabilidad:

- **Escalabilidad del hardware** (también denominada “**escalamiento vertical**”). Se basa en la utilización de un gran equipo cuya capacidad se aumenta a medida que lo exige la carga de trabajo existente.
- **Escalabilidad del software** (también denominada “**escalamiento horizontal**”). Se basa, en cambio, en la utilización de un cluster compuesto de varios equipos de mediana potencia que funcionan en tándem de forma muy parecida a como lo hacen las unidades de un RAID (Redundant Array of Inexpensive Disks o Array redundante de discos de bajo coste). Se utilizan el término RAC (Redundant Array of Computers o Array redundante de equipos) para referirse a los clusters de escalamiento horizontal. Del mismo modo que se añaden discos a un array RAID para aumentar su rendimiento, se pueden añadir nodos a un cluster para aumentar también su rendimiento.

**Clusters de Alta Confiabilidad (HR o High Reliability).** Este tipo de clusters se utilizan en entornos donde se necesite conocer como se va a comportar el sistema en ciertos escenarios. Este tipo de clusters son los más difíciles de implementar, no están basados solamente en proporcionar los servicios de alta disponibilidad, sino además de ofrecer un entorno altamente confiable. Esto implica una gran sobrecarga en el sistema.

Dar a un cluster que tenga un sistema en común entonces sus nodos, la capacidad de alta confiabilidad implica gastar recursos necesarios para evitar que aplicaciones dejen de responder. Esto es, en los clusters de alta disponibilidad si un servicio se ha caído el sistema lo reinicia, pero no existe manera de conservar el estado anterior del servidor, solo mediante *checkpoints*, pero en conexiones de tiempo real esto no es suficiente. Por otro lado, los clusters de alta confiabilidad tratan de mantener el estado de las aplicaciones, no solamente utilizan el último *checkpoint* del sistema y reinician el servicio. Regularmente este tipo de clusters son utilizados en el ámbito empresarial y su funcionalidad depende de un hardware especializado.

**Clusters Homogéneos.** En este tipo de clusters todas las computadoras tienen configuraciones hardware idénticas, además de usar el mismo sistema operativo. Todos los nodos que componen el cluster tienen una arquitectura y recursos similares, así que las diferencias entre cada nodo son mínimas o nulas.

**Clusters Heterogéneos.** A diferencia del tipo anterior, se pueden tener arquitecturas totalmente diferentes así como sistemas operativos, este es el tipo más utilizado ya que permite la reutilización de los equipos de cómputo que ya se tiene como obsoleto. Pero implica poner más atención en balance de la carga, ya que los equipos no son todos de la misma capacidad y al conjuntarlos si un equipo de poca potencia tiene tareas que necesitan

gran de capacidad procesamiento tardará más de lo esperado en realizar las tareas programadas.

## 2.4 Componentes de un Cluster

En general, un cluster necesita de varios componentes de software y hardware para poder funcionar:

- Nodos
- Sistemas Operativos
- Conexiones de Red
- Middleware (comunicación entre nodos)
- Almacenamiento
- Protocolos de Comunicación y servicios
- Ambientes de Programación Paralela

### 2.4.1 Nodos

Por definición, un **nodo** es un punto de intersección o unión de varios elementos que confluyen a un mismo lugar. Ahora bien, dentro del ambiente computacional la palabra **nodo** puede referirse a conceptos diferentes según en ámbito donde lo utilicemos. Para el caso de los cluster de computadoras un nodo es una computadora, un sistema multiprocesador o una estación de trabajo (*Workstation*). Conectadas dentro de una red cada una de las máquinas que la conforman es un nodo, y si la red es Internet, cada servidor constituye también un nodo.

Un cluster puede estar conformado por nodos dedicados o por nodos no dedicados.

**Nodos dedicados:** en este caso los nodos no disponen de teclado, mouse ni monitor y su uso está exclusivamente dedicado a realizar tareas de procesamiento en conjunto con los demás nodos.

**Nodos no dedicados:** los nodos disponen de teclado, mouse y monitor y su uso no está exclusivamente dedicado a realizar tareas de procesamiento como cluster, también se utiliza como estación de trabajo independiente cuando no se esta trabajando en paralelo.

Es recomendable que los nodos tengan características en común, es decir, que sean similares en arquitectura y sistemas operativos, ya que si se conforma un Cluster con Nodos totalmente heterogéneos (existe una gran variedad de procesadores y memoria, en términos de velocidad y capacidad) puede llegar a ser ineficiente si el *middleware* no tiene la capacidad de equilibrar la carga de procesos y toma a todos los nodos por igual.

## 2.4.2 Sistemas Operativos

En este punto debe tenerse en cuenta un sistema que permita interactuar un nodo con otro de la manera más transparente que sea posible, ya que si no es de esta manera los nodos pueden desperdiciar recursos en establecer comunicación debido a que el sistema no está diseñado para este tipo de tareas, en particular se utilizan sistemas Linux, que ha sido el más utilizado desde el nacimiento de los clusters aunque existen casos donde se utiliza Windows en particular los orientados a redes (NT, 2000, 2003 y 2008) para crear clusters de computadoras.

## 2.4.3 Conexiones de Red

Es la interconexión física entre los nodos, esta puede ir desde el cable de red hasta la fibra óptica. Regularmente utilizan *Fast Ethernet* o cable de red, debido a que es el medio más económico para realizar la comunicación con velocidades de 10 Mbps a 1 Gbps. En varias universidades se utiliza *Myrinet* que es una red de interconexión de cluster de altas prestaciones. Físicamente consiste en dos cables de fibra óptica, upstream y downstream, unidos en un único conector, que llegan a alcanzar velocidades de 10 Gbps con latencia de 3 microseg. Muchas de las supercomputadoras que aparecen en la lista de *top500* utilizan esta tecnología de comunicación.

## 2.4.4 Almacenamiento

Para el almacenamiento de información se utilizan varios tipos de tecnologías como el *NAS* (del inglés *Network Attached Storage*) es el nombre dado a una tecnología de almacenamiento dedicada a compartir unidades de almacenamiento de una computadora (Servidor) con computadoras personales o servidores clientes a través de una red, también se tiene el *SAN* (del inglés *Storage Area Network*) que tiene una relación de uno a uno con el servidor, donde el un cliente reserva un archivo y es exclusivo para ese cliente a diferencia que el *NAS* que es compartido por todos los clientes. El protocolo más utilizado comúnmente es *NFS* (*Network File System*), sistema de archivos compartido entre servidor y los nodos. Sin embargo existen sistemas de archivos específicos para clusters como *Lustre (CFS)* y *PVFS2*.

## 2.4.5 Middleware

El *Middleware* es un software de conectividad que ofrece un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. Funciona como una capa de abstracción de software distribuida, que se sitúa entre las capas de aplicaciones y las capas inferiores (sistemas operativos y red). El *Middleware* nos abstrae de la complejidad y heterogeneidad de las redes de comunicaciones subyacentes, así como de los sistemas operativos y lenguajes de programación, proporcionando una interfaz *API* para la fácil programación y manejo de aplicaciones

distribuidas. Dependiendo del problema a resolver y de las funciones necesarias, serán útiles diferentes tipo de servicios de middleware.

Por lo general el middleware del lado cliente está implementado por el Sistema Operativo, el cual posee las librerías que implementan todas las funcionalidades para la comunicación a través de la red. Un ejemplo de *Middleware* es el *RPC* o *llamadas de procedimientos remotos* que permite a un programa ejecutar código en otra máquina remota sin tener que preocuparse por las comunicaciones entre ambos equipos.

## 2.4.6 Protocolos de Comunicación

Los Protocolos de Comunicación o también Protocolos de red son reglas que permiten el flujo de información entre distintas computadoras que manejan diferentes lenguajes, por ejemplo, dos computadoras conectadas en una misma red pero que cuentan con protocolos diferentes no podrán comunicarse, para que esto sea posible, es necesario que ambas computadoras "*hablen*" el mismo lenguaje, por tal motivo, el protocolo TCP/IP fue creado para las comunicaciones en Internet, así cualquier computadora que se conecte a Internet necesita tener este protocolo para intercambiar información.

## 2.4.7 Ambientes de programación paralela

Hay diferentes formas de hacer procesamiento paralelo, entre las más conocidas y usadas podemos destacar NUMA, PVM y MPI. Las máquinas de tipo NUMA (Non-Uniform Memory Access) tienen acceso compartido a la memoria donde pueden ejecutar su código de programa.

En el kernel de Linux hay ya implementado NUMA, que hace variar el número de accesos a las diferentes regiones de memoria. PVM y MPI son herramientas que han sido ampliamente utilizadas y muy conocidas por la gente que utiliza supercomputación.

MPI (Message Passing Interface) es un estándar abierto de bibliotecas de paso de mensajes. MPICH es una de las implementaciones más usadas de MPI, tras MPICH se puede encontrar LAM, otra implementación basada en MPI también con bibliotecas de código abierto.

PVM (Parallel Virtual Machine) es una variante de MPI que también es ampliamente usado para funcionar en entornos Beowulf. PVM habita en el espacio de usuario y tiene la ventaja que no hacen falta modificaciones en el kernel de Linux, básicamente cada usuario con derechos suficientes puede ejecutar PVM.

## 2.5 Ventajas y desventajas de un cluster

Los beneficios de un cluster de computadoras son variados, pero también tiene sus desventajas, a continuación nombramos algunas ventajas:

- El principal beneficio es el costo que implica construir un cluster, ya que con equipos de escritorio puede crearse un cluster.
- Gran poder de procesamiento, en comparación incluso con estaciones de trabajo y que significarían un costo mayor que el construir un cluster.
- Si se crea un cluster de alta escalabilidad puede aumentar la potencia del mismo agregando un nuevo nodo o algún elemento como memoria, disco duro y en algunos casos, el procesador.
- El aprovechamiento de equipo. Pueden utilizarse computadoras que en la actualidad no sean potentes pero en conjunto pueden competir con varias estaciones de trabajo.
- Se conectan, por lo general, por LAN de alta velocidad que es barata a comparación de otro tipo de conectividad.

Entre las desventajas tenemos:

- Si no se configura correctamente un cluster puede no tener la potencia de procesamiento que esperamos.
- Descubrir y diagnosticar fallas es complicado. Y crece si es un cluster heterogéneo.
- La programación es más compleja, las supercomputadoras vienen con software propietario que ayuda a realizar las tareas en paralelo, en los cluster se debe elegir alguna otra opción que regularmente siempre es programación explícita.
- Saturación en la red. Los cuellos de botella se llegan a generar dependiendo de cómo sea diseñada la comunicación.

Teniendo en cuenta estos puntos, debe hacerse una planeación de un cluster para que este sea funcional y además escalable que es una parte primordial en un cluster, crecer sin modificaciones grandes en el sistema y que también aumente su rendimiento sin que su costo sea una limitante.

## 2.6 TOP500.org, las 500 computadoras más poderosas del mundo

El *Top 500* es una lista con las computadoras comerciales más poderosas del mundo en términos de procesamiento. Este proyecto inicia en 1993, esta lista se publica 2 veces al año en Junio y Noviembre. La primera actualización se realiza en Junio, coincidiendo con la

International Supercomputer Conference, y la segunda actualización se realiza en Noviembre en la IEEE Supercomputer Conference. Del total de estas 500 computadoras mayor poder de procesamiento 410 son clusters de computadoras. LINPACK es una prueba desarrollada por Jack J. Dongarra del Argonne National Laboratory de los Estados Unidos, que se ha convertido, en los últimos años, en una prueba de referencia para medir la eficiencia de sistemas dedicados esencialmente al cálculo científico y técnico.

Los valores **R<sub>max</sub>** es el rendimiento máximo obtenido para la prueba de **LINPACK** y **R<sub>peak</sub>** es el pico de rendimiento teórico, en otros términos es el total de operaciones de punto flotante que realiza en un ciclo de reloj. Los datos están dados en TFlops.

| Pos | Equipo          | Lugar                                                                        | Sistema                                                                                                                                       | Procs  | R<br>max | R<br>peak |
|-----|-----------------|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|--------|----------|-----------|
| 1   | Roadrunner      | DOE/NNSA/L<br>ANL<br>Estados Unidos                                          | BladeCenter<br>QS22/LS21 Cluster,<br>PowerXCell 8i 3.2 Ghz /<br>Opteron DC 1.8 GHz,<br>Voltaire Infiniband<br>IBM                             | 129600 | 1105     | 1456.7    |
| 8   | Ranger          | Texas<br>Advanced<br>Computing<br>Center/Univ. of<br>Texas<br>Estados Unidos | SunBlade x6420,<br>Opteron QC 2.3 Ghz,<br>Infiniband<br>Sun Microsystems                                                                      | 62976  | 433.2    | 579.38    |
| 10  | JUROPA          | Forschungszentr<br>um Juelich<br>(FZJ)<br>Alemania                           | Sun Constellation,<br>NovaScale R422-E2,<br>Intel Xeon X5570, 2.93<br>GHz, Sun M9/Mellanox<br>QDR Infiniband/Partec<br>Parastation<br>Bull SA | 26304  | 274.8    | 308.28    |
| 15  | Magic<br>Cube   | Shanghai<br>Supercomputer<br>Center<br>China                                 | Dawning 5000A, QC<br>Opteron 1.9 Ghz,<br>Infiniband, Windows<br>HPC 2008<br>Dawning                                                           | 30720  | 180.6    | 233.47    |
| 16  | GPC             | SciNet/Universi<br>ty of Toronto<br>Canada                                   | iDataPlex, Xeon E55xx<br>QC 2.53 GHz, GigE<br>IBM                                                                                             | 30240  | 168.6    | 306.03    |
| 18  | EKA-<br>Cluster | Computational<br>Research<br>Laboratories,<br>TATA SONS<br>India             | Cluster Platform 3000<br>BL460c, Xeon 53xx<br>3GHz, Infiniband<br>Hewlett-Packard                                                             | 14384  | 132.8    | 172.61    |

|     |           |                                                                                                                            |                                                                                            |      |       |       |
|-----|-----------|----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|------|-------|-------|
| 439 | Aitzaloea | Supercomputing<br>and<br>Visualization in<br>Parallel<br>Laboratory,<br>Universidad<br>Autónoma<br>Metropolitana<br>México | Lufac Cluster, Intel<br>Xeon E54xx 3.0 Ghz,<br>Infiniband<br>LUFAC Computacion<br>SA de CV | 2120 | 18.48 | 25.44 |
|-----|-----------|----------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|------|-------|-------|

**Tabla 1 Lista de los clusters que se encuentran en el Top500**

La prueba consiste en la resolución de un sistema, denso y aleatorio, de 100 ecuaciones lineales con 100 incógnitas, utilizando subrutinas **FORTAN** de los paquetes **LINPACK** y **BLAS**. El tiempo de ejecución (tiempo de Unidad Central) se divide por el número de operaciones (adiciones y multiplicaciones) en punto flotante (Flops), para dar una medida de la potencia de cálculo del sistema, evaluada en Mflops/seg.

### 2.6.1 IBM Roadrunner

En primer lugar la supercomputadora Roadrunner construida por IBM en el Laboratorio Nacional de Los Álamos en Nuevo México para La Administración Nacional de Seguridad Nuclear del Departamento de energía de los Estados Unidos. Es la primera computadora en alcanzar un rendimiento sostenido de un 1 petaFLOPS, en Noviembre de 2008 alcanzo los **1.456 petaFLOPS** alcanzando el 1er. lugar de la lista top500.

Ubicado en un área de 1,100 m<sup>2</sup> cuenta con 12,960 procesadores tipo PowerXCell 8i mejorados, que inicialmente fueron diseñados para consolas Playstation 3, 6912 procesadores Opteron, con 92 km de fibra óptica para la comunicación en un sistema Triblade con InfiniBand.

Utiliza como sistema operativo Red Hat Enterprise Linux y xCAT software de computación distribuida. Tiene un consumo de energía estimado de 3 Mega vatios.

Tuvo un costo de 133 millones de dólares. En un principio fue instalado en el centro de IBM en Poughkeepsie (NY), después fue trasladado al Laboratorio Nacional de Los Álamos en Nuevo México, donde se dedicará a actuar como patrón de seguridad del arsenal de armas nucleares para Estados Unidos además de utilizarse para el estudio de problemas relacionados con el clima, la astronomía y la genómica.

### 2.6.2 Sun Constellation Linux Cluster: Ranger

**Range** es el sistema de cómputo más grande del mundo abierto para la investigación en cómputo científico, ingeniería y tecnología principalmente. Construida para el Centro de

Computación Avanzada de Texas (TACC sus siglas en ingles) de la Universidad de Texas, Estados Unidos consta de 3,936 nodos, un total de 62,976 procesadores Opteron quad-core, 13 TeraBytes de memoria y 1.73 petaBytes de almacenamiento compartido. Utiliza como sistema operativo CentOS y alcanza un rendimiento de : **579.4 TeraFLOPS**.

En Septiembre de 2006, la Fundación Nacional de la Ciencia (**NSF**) proporcionó al Centro 59 millones de dólares para comprar un sistema de supercomputo. Nombrado como **Ranger** construido por **Sun Microsystems** como su primer sistema **Sun Constellation System** en Febrero de 2008 y en Junio entra en la lista Top 500.

El Centro de Cómputo realiza investigaciones y desarrolla actividades con patrocinio de programas federales como:

- El programa NSF TeraGrid
  - El proyecto Computational Chemistry Grid
- El programa NSF Information Technology Research
- El programa de computo Avanzado para el Departamento de Energía
- Programa de Modernización de Computo de Alto Rendimiento del Departamento de Defensa
- El programa Power Grid para la National Aeronautics and Space Administration (NASA).

### 2.6.3 Sun Constellation Linux Cluster: JUROPA

Este cluster es parte del proyecto **JuRoPA** (*Jülich Research on Petaflop Architectures*) del Centro de Investigación de Jülich (*Jülich Research Centre*) en Alemania que a su vez está integrado al programa europeo PRACE (*Partnership for Advanced Computing in Europe*). Con 300 TeraFLOPS de rendimiento está dedicado exclusivamente a las aplicaciones del proyecto **Fusion** de la Unión Europea. El principal objetivo del proyecto es acelerar la investigación sobre la fusión nuclear.

El proyecto **JuRoPA** se compone de 2 clusters de computadoras **Juropa-JSC** y el **HPC-FF**. Estos cluster tienen los siguientes componentes:

#### ***Juropa-JSC***

- 2208 nodos cada uno con 2 procesadores Intel Xeon X5570 (Nehalem-EP) quad-core a una velocidad de 2.93 Ghz con SMT (Simultaneous Multithreading)
- 17664 procesadores en total
- 24 GB de memoria (DDR3, 1066 MHz)

- Conexión IB QDR HCA (via QNEM Network Express Module)
- Un pico de rendimiento 207 TeraFLOPS
- Sistema Sun Blade 6048
- Red Infiniband QDR con una topología Fat Tree no bloqueante
- Sun Data Center Switch 648

### ***HPC-FF***

- 1080 nodos cada uno con 2 procesadores Intel Xeon X5570 (Nehalem-EP) quad-core a 2.93 Ghz con SMT (Simultaneous Multithreading)
- 8640 procesadores en total.
- 24 GB memory (DDR3, 1066 MHz)
- Conexión Infiniband Mellanox ConnectX QDR HCA
- Pico de rendimiento de 101 TeraFLOPS
- Con tecnología Bull NovaScale R422-E2
- Red Infiniband QDR con una topología Fat Tree no bloqueante
- Mellanox MTS3600 Switch

### **Sistema completo**

- 3288 nodos.
- Memoria principal de 79 TeraBytes
- 26304 procesadores en Total
- Pico de rendimiento de 308 TeraFLOPS
- 2 Nodos configurados como maestro
- Sistema Operativo SUSE SLES 11

### Herramientas de programación:

- ParaStation Cluster Management
- Torque / Moab Batch como Sistema de Administración de Recursos
- Intel Professional Fortran, C/C++ Compiler, Intel Cluster Tools, Intel Math Kernel Library
- ParTec MPI
- OpenMP

## 2.6.4 Aitzaloo

**Aitzaloo** es una supercomputadora de la Universidad Autónoma Metropolitana, campus Iztapalapa. Entro en funcionamiento el 30 de noviembre de 2008 y forma parte del Laboratorio de Supercómputo y Visualización en Paralelo (LSVP). La **UAM** lo adquirió de un proveedor poco conocido **LUFAC** a un costo de 12 millones de pesos como parte de la celebración por el 50 Aniversario de la Computación en México.

Su nombre deriva de los vocablos del idioma náhuatl, **Aitia** (*donde se llevan a cabo varias tareas*) y **Zalohua** (*donde se aprende*).

**Aitzaloo** contiene 2160 núcleos de procesamiento (en 540 procesadores Quad Core de Intel) con una capacidad de 18.48 TeraFLOPS, 4320 GB de RAM. Es la computadora más potente en América Latina, posicionada en el lugar número 439 en la lista **top500** en Junio de 2009. En la lista anterior en Noviembre de 2008 apareció en el lugar 226.

### Nodos de CAR (HPC)

- Número de Nodos: 270 nodos con dos procesadores por nodo.
- Procesadores: Intel Xeón Quad-Core a 3 Ghz
- Número de procesadores: 540 procesadores Quad-Core (2160 Cores en total)
- FSB: 1600 MHz
- Memoria: 16GB en RAM por nodo.
- Capacidad de Cómputo: 18.4 TFlops.
- Comunicaciones: Infiniband.
- Sistema Operativo: Linux Centos 5.2
- Tipo de uso: Divisional

### Sistema de Almacenamiento Lustre

- Sistema: Lustre 100TB
- Discos Duros: 150HD de 1TB en Raid 1 y Raid 6
- Comunicaciones: Infiniband y Gigabit Ethernet.

# Capítulo 3

## Programación Distribuida y Paralela

Durante los últimos 20 años el avance en la tecnología ha tenido un cambio significativo en la computación de usar *supercomputadores* o *mainframes* para cálculos hacia las redes de computadoras. El crecimiento de las velocidades de los procesadores de escritorio, la potencia de procesamiento de los mismos, la velocidad de la memoria, la capacidad de almacenamiento y las velocidades de la redes, que en otra época estaban solo disponibles en las *supercomputadoras* o *mainframes*, ha cambiado la forma de realizar cómputo utilizando redes de computadoras para resolver problemas que requieren una gran cantidad de cálculos, entrando en áreas que antes eran exclusivas para las grandes supercomputadoras.

### 3.1 Sistemas Distribuidos

Los *sistemas distribuidos* ha hecho posible tomar un grupo de computadoras, estaciones de trabajo o servidores y hacerlos trabajar como un solo equipo. En este entorno los usuarios pueden acceder a recursos locales o remotos en el sistema sin preocuparse en qué computadora se encuentren.

Ahora en general definición de un sistema distribuido es una colección de computadoras completamente independientes unas de otras y que se muestran ante los usuarios como una única computadora. Esta definición tiene 2 puntos importantes, el primero se refiere al hardware de las computadoras, lo que es necesario para la interconexión y el segundo se refiere al software, como hacer para que el usuario vea una sola máquina.

### 3.1.1 Características de los sistemas distribuidos

Como no se tiene una definición concreta reconocida sobre los sistemas distribuidos podemos tomar las siguientes características como las principales de estos sistemas:

**Compartición de Recursos:** El término *recurso* es bastante abstracto, puede referirse a algo físico como una impresora o algo no tangible como un archivo; pero no es nuevo ni aparece por primera vez en el área de los sistemas distribuidos. Los sistemas multiusuario clásicos siempre han provisto la compartición de recursos entre usuarios. Los recursos de una computadora multiusuario se comparten de manera transparente entre los usuarios y por el contrario, los usuarios de estaciones de trabajo monousuario o computadoras personales dentro de un sistema distribuido no obtienen automáticamente los beneficios de la compartición de recursos.

Los recursos en un sistema distribuido están físicamente encapsulados en una computadora y sólo pueden ser accedidos por otras mediante la red. Para que la compartición de recursos sea efectiva, estos deben ser administrados por un programa que ofrezca una interfaz de comunicación, permitiendo que el recurso sea accedido, usado y modificado de una manera fiable y consistente. De lo anterior encontramos un nuevo término: **gestor de recursos**.

Un gestor de recursos es un módulo de software que administra un conjunto de recursos de un tipo en particular. Cada tipo de recurso requiere políticas y métodos específicos junto con requisitos comunes para todos ellos.

**Abierto:** Un sistema es abierto si puede ser ampliado de diversas formas. Un sistema puede ser abierto o cerrado respecto al hardware, si es posible agregar periféricos, memoria o interfaces de comunicación; o con respecto al software, en este caso si es posible agregar más características al sistema operativo como protocolos de comunicación y servicios de compartición de recursos. Que un sistema distribuido sea abierto se determina por el grado en el que nuevos servicios de compartición de recursos se pueden agregar sin perjudicar ni duplicar los ya existentes.

Básicamente los sistemas distribuidos cumplen una serie de características:

1. Las interfaces clave en el software del sistema están especificadas de manera concreta y están a disposición de los desarrolladores. Es decir, las interfaces son públicas.
2. Los sistemas distribuidos abiertos deben otorgar un mecanismo uniforme de comunicación entre procesos e interfaces publicas para el acceso a lo recursos compartidos.
3. Los sistemas distribuidos abiertos pueden construirse a partir de hardware y software heterogéneo. Pero cada componente debe cumplir ciertas características si quiere evitarse tener problemas de integración.

**Concurrencia:** Cuando existen varios procesos en una única máquina decimos que están ejecutándose de forma concurrente. Si la computadora tiene un único procesador, la concurren-

cia se da alternando la ejecución de los distintos procesos. Si la computadora tiene  $N$  procesadores, entonces se pueden estar ejecutando estrictamente a la vez hasta  $N$  procesos.

En los sistemas distribuidos hay muchas máquinas, cada una con uno o más procesadores. Es decir, si hay  $M$  computadoras en un sistema distribuido con un procesador cada una, entonces pueden estar ejecutándose hasta  $M$  procesos en paralelo.

En un sistema distribuido basado en el modelo de compartición de recursos, procesamiento paralelo ocurre por dos motivos:

- Varios usuarios utilizan o interactúan simultáneamente con programas de aplicación. Con pocas posibilidades de algún conflicto, ya que normalmente las aplicaciones se ejecutan aisladamente en la computadora del usuario y no entran en conflicto con las aplicaciones ejecutadas en las computadoras de los demás usuarios.
- Varios procesos servidor se ejecutan de manera concurrente, cada uno respondiendo a las diferentes peticiones de los procesos cliente. Las peticiones para tener acceso a los recursos de un servidor en concreto pueden ser encoladas y ser procesadas ya sea secuencial o bien concurrentemente por múltiples instancias mediante el gestor de recursos. Cuando esto ocurre los procesos servidor deben sincronizar sus acciones para asegurarse de no crear conflictos.

**Escalabilidad:** Los sistemas distribuidos operan de manera efectiva y eficiente a niveles diferentes. El nivel más pequeño consiste en dos computadoras y un servidor, mientras que un sistema distribuido construido alrededor de una red de área local simple podría contener cientos de computadoras, varios servidores de archivos, servidores de impresión y otros servidores de propósito específico. Se pueden llegar a conectar varias redes de área local entre sí, que llegan a contener miles de computadoras y formar así un único sistema distribuido, permitiendo que los recursos sean compartidos entre todos ellos, este sistema es conocido como Grid.

En un sistema distribuido lo ideal es que el software del sistema y de cualquier aplicación no debería cambiar cuando es ampliado. La necesidad de escalabilidad no es solo un problema de prestaciones de servicios de red o de hardware, sino que está íntimamente ligada con los aspectos del diseño de los sistemas distribuidos. En el diseño del sistema debe tenerse siempre en cuenta la necesidad de escalabilidad o de lo contrario aparecerán serias limitaciones. En concreto, el trabajo necesario para procesar una petición de acceso a un recurso compartido debe ser prácticamente independiente del tamaño de la red.

**Tolerancia a Fallos:** Los sistemas no están exentos de fallas, cuando éstas se producen en el software o en el hardware, los programas podrían producir resultados incorrectos o podrían pararse antes de terminar las tareas que estaban realizando. El diseño de sistemas tolerantes a fallos se basa en dos cuestiones, complementarias entre sí: Redundancia de hardware (uso de componentes redundantes) y recuperación del software (diseño de programas que sean capaces de recuperarse de los fallos).

La redundancia de hardware se refiere al uso de uno o más componentes de hardware para realizar tareas idénticas.

La recuperación del software tiene relación con aplicaciones especiales que sean capaces de recuperar el estado de los datos permanentes antes de que se produjera el fallo.

**Transparencia:** La transparencia oculta al usuario y al programador la separación de los componentes de un sistema distribuido, de manera que todas las partes que lo integran se perciben como una sola entidad, en vez de un grupo de componentes independientes.

El manual de referencia RM-ODP [ISO 1996a] que ofrece cinco puntos de vista para crear ambiente distribuidos abiertos, nos proporciona ocho formas de transparencia. Estas proveen un resumen útil de la motivación y metas de los sistemas distribuidos. Las transparencias definidas son:

- *Transparencia de Acceso:* Permite el acceso a los recursos remotos como si se trataran de recursos locales.
- *Transparencia de Localización:* Permite el acceso a los recursos sin la necesidad de conocer su ubicación
- *Transparencia de Concurrencia:* Permite que varios procesos operen concurrentemente utilizando objetos de información compartidos y de forma que no exista interferencia entre ellos.
- *Transparencia de Replicación:* Permite utilizar múltiples instancias de los objetos de información para incrementar la fiabilidad y las prestaciones sin que los usuarios o los programas de aplicación tengan por que conocer la existencia de las réplicas.
- *Transparencia de Fallos:* Permite a los usuarios y programas de aplicación completar sus tareas a pesar de la ocurrencia de fallos en el hardware o en el software.
- *Transparencia de Migración:* Permite el movimiento de objetos de información dentro de un sistema sin afectar a los usuarios o a los programas de aplicación.
- *Transparencia de Prestaciones.* Permite que el sistema sea reconfigurado para mejorar las prestaciones mientras la carga varia.
- *Transparencia de Escalado:* Permite la expansión del sistema y de las aplicaciones sin cambiar la estructura del sistema o los algoritmos de la aplicación.

Las dos más importantes son las transparencias de acceso y de localización; su inclusión o exclusión afecta fuertemente el uso de los recursos distribuidos. A menudo se les denomina transparencias de red.

Los sistemas distribuidos han sido ampliamente estudiados y creados o desarrollados por varios investigadores, pero no significa que tengamos el suficiente conocimiento de como

crear un sistema distribuido, los sistemas creados han sido hechos a la medida de necesidades concretas. De los sistemas desarrollados podemos tomar las ventajas más importantes:

- *Incremento del rendimiento:* tener varios equipos en un sistema distribuido permite que las aplicaciones sean procesadas en paralelo, esto es lo que incrementa el rendimiento.
- *Recursos compartidos:* Los sistemas distribuidos ofrecen un acceso eficiente a los recursos de los componentes del sistema, por ejemplo compartir bases de dato, recursos de cómputo o espacio de almacenamiento.
- *Posibilidad de crecimiento:* el diseño de sistemas distribuidos es modular y adaptable, esto es, que tiene la posibilidad que el sistema crezca con pocos cambios en la configuración del mismo. Esto resuelve la problemática de cuando un problema rebasa la capacidad del sistema.
- *Alto grado de confiabilidad, disponibilidad y tolerante a fallos:* Tener varios equipo que puedan obtener resultados efectivos, que se pueda acceder a los recursos y que las fallas en algún elemento no repercutan en las tareas que realizamos es una característica primordial en estos sistemas.
- *El costo:* Con el paso del tiempo el rendimiento de estos sistemas va en aumento mientras el costo de los recursos utilizados se reduce en varios casos.

Estas ventajas no son fáciles de obtener ya que el diseño de un sistema de carácter general envuelve un gran número de características que deben cumplirse tanto físicas como de configuración, de modo que es más fácil crear sistemas de acuerdo a necesidades específicas o dirigidos a ciertas áreas.

Varios factores son la causa que no se pueda generalizar un sistema distribuido, los aspectos como la configuración de la comunicación, las características de la plataforma, la administración de tareas, las políticas del uso de los recursos, el control de la concurrencia y la seguridad entre otros. A continuación presentamos unos puntos que consideramos como principales:

- La más importante es la falta de completo entendimiento de la teoría del cómputo distribuido, especialmente en la distribución de procesos de propósito general, Sape Mullender en su libro ***Distributed Systems*** compara el diseño de un sistema distribuido con el sistema ferroviario de EE.UU. el cual tomo un siglo en madurar y tener un completo entendimiento para su aprovechamiento efectivo.
- El comportamiento independiente y asíncrono de los recursos de un sistema y el control del software complican el objetivo para funcionar como un solo sistema, más centralizado.
- La red de comunicación entre computadoras agrega otro nivel de complejidad para estos sistemas. El diseñar un sistema distribuido no solo implica el diseño de

hardware y de software, también debe contar con un diseño de comunicación efectiva, revisar correctamente aspectos como la sincronización, la consistencia y cómo actuar ante fallas cuando se tiene un entorno heterogéneo.

### 3.1.2 Ejemplos de algunos sistemas

- El más utilizado actualmente: Internet, es un sistema compuesto de varias redes a las cuales se puede acceder a sus recursos no importando donde se encuentren, ni las características del equipo donde se alojan, solamente escribimos la dirección y tenemos acceso a la información, claro esta, que también existen políticas de uso.
- El sistema de un banco, donde cada sucursal tiene una computadora que lleva un control de sus cuentas localmente, además de tener un servidor general donde se tiene la información de todas las sucursales. Cada sucursal puede realizar transacciones de un banco a otro sin importar donde se encuentren las sucursales.
- Otro ejemplo es una línea de producción con robots, para ir de un proceso a otro hay robots monitorean o revisan defectos en la fabricación, y se comunican con otro robot, para que no continúen con el proceso si un material esta defectuoso, esto para el usuario solamente es una línea de producción, no se ve por separado cada robot de la misma.

### 3.1.3 Programación Distribuida

La **programación distribuida** es un paradigma de programación cuyo fin es desarrollar sistemas distribuidos, que sean abiertos, escalables, transparentes al usuario y tolerantes a fallos. En programación distribuida, un conjunto de computadoras conectadas por una red son usadas colectivamente para realizar tareas.

Por otro lado en los sistemas paralelos, la solución a un problema importante es dividida en pequeñas tareas que son repartidas y ejecutadas para conseguir un alto rendimiento.

Casi cualquier lenguaje de programación que tenga acceso al hardware del sistema puede manejar la programación distribuida, considerando una buena cantidad de tiempo y código. Algunos lenguajes que podemos nombrar que fueron específicamente diseñados para programación distribuida son: Ada, Alef, E, Erlang, Limbo y Oz.

## 3.2 Programación Paralela

Recientemente han aparecido máquinas paralelas de multiprocesadores simétricos (SMP) con 2 o 4 procesadores que están al alcance de varios usuarios, antes estos procesadores eran considerados como servidores empresariales. Podríamos pensar que el cambio hacia estos equipos tendrá como resultado que nuestras aplicaciones se ejecuten más

rápido, pero en la mayoría de los casos no es así, esto es debido a que tales aplicaciones fueron diseñadas para ser secuenciales y en muchas ocasiones con lenguajes que son de naturaleza secuencial lo cual que no permite explotar al máximo las capacidades de los nuevos procesadores, incluso recompilar estas aplicaciones en base a las nuevas arquitecturas para que aprovechen el equipo sería un problema por la misma naturaleza del diseño, lo mejor es crearlo desde cero y diseñado para funcionar en paralelo.

En la actualidad el paralelismo se crea usando lenguajes secuenciales con librerías que proporcionen funcionalidades paralelas. Para máquinas SMP's usualmente utilizan hilos (threads), procesos o tareas, agregan un mecanismo de sincronización y candados o monitores para aprovechar la capacidad de estos equipos, los que van más allá de usar solo computadoras SMP's y programan aplicaciones para clusters de computadoras utilizan el paso de mensajes.

### 3.2.1 ¿Qué es Paralelismo?

Es un mecanismo que involucra más de un proceso a la vez, es decir, la realización de varias tareas o procesos al mismo tiempo. También se describe como la creación, destrucción e interacción de procesos que distingue a los paradigmas de programación.

### 3.2.2 ¿Qué es Cómputo Paralelo?

Se le llama cómputo paralelo a la ejecución de más de un cómputo o cálculo al mismo tiempo usando más de un procesador en una computadora.

La meta es reducir al mínimo el tiempo total de cómputo distribuyendo la carga de trabajo entre los procesadores disponibles. Para fines prácticos es necesario tener en mente que la paralelización es un factor básico para tener un alto desempeño en los equipos de cómputo.

Una de las razones principales para utilizar el paralelismo en el diseño de hardware o software es obtener un alto rendimiento o mayor velocidad al ejecutar un programa.

La velocidad de procesamiento no es solamente la razón para utilizar el paralelismo. La construcción de aplicaciones más complejas han requerido una computadora más rápida, y las limitaciones en el desarrollo adicional de computadoras seriales han llegado a ser más y más evidentes.

### 3.2.3 ¿Cuándo usar paralelismo?

Esta es una pregunta que comúnmente se hace un programador, y la respuesta no es muy sencilla, dado que debe analizarse si se cuenta con el hardware y software necesario y determinar si vale la pena paralelizar el programa o viceversa.

En cuestiones de equipo el proceso de paralelizar un programa involucra, que el programador conozca un poco más de la arquitectura de la supercomputadora o **cluster** sobre el cual pretende paralelizar su código, conocer el número de procesadores con los que se cuenta, la cantidad de memoria, espacio en disco, los niveles de memoria disponible, el medio de interconexión, entre otros.

En términos de software el usuario debe conocer qué sistema operativo esta manejando, si los compiladores instalados permiten realizar aplicaciones con paralelismo, si se cuenta con herramientas como PVM, MPI, Cilk o Charm++ (en Sistemas Distribuidos), Power C, Power Fortran u OpenMP (en sistemas de memoria compartida).

### 3.2.4 Características de un programa paralelo

Un programa en paralelo tiene que cumplir ciertos requisitos para un funcionamiento correcto, de lo contrario existe la posibilidad que el funcionamiento de la aplicación que creemos no tenga los resultados esperados o sea inviable en comparación de una solución secuencial. A continuación se muestran algunas características que deben cumplir.

#### 3.2.4.1 Granularidad

La **granularidad** en algunos casos la definen como el número de procesadores en un sistema, aquí lo definiremos con el número de unidades básicas o mínimas de procesamiento en que se divide una aplicación. Dichas piezas puede ser una sentencia de código, una función o un proceso que se ejecutarán en paralelo.

En paralelismo la granularidad tiene dos categorías, *paralelismo de grano fino* y *paralelismo de grano grueso*.

De *grano fino* es cuando el código se divide en una gran cantidad de piezas pequeñas. Es a un nivel de sentencia donde un ciclo se divide en varios subprocesos que se ejecutarán en paralelo. Se le conoce además como Paralelismo de Datos.

De *grano grueso* es a nivel de subrutinas o segmentos de código, donde las piezas son pocas y de cómputo más intensivo que las de grano fino. Se le conoce como Paralelismo de Tareas

Las variaciones entre los paradigmas están motivadas por varios factores:

Diferencia de esfuerzo invertido en escribir programas paralelos. Algunos lenguajes requieren menos esfuerzo para el programador, mientras que otros requieren menos trabajo pero, generan un código menos eficiente.

Un determinado paradigma de programación puede ser más eficiente que otros al programar sobre determinadas arquitecturas paralelas.

Distintas aplicaciones tienen diferentes tipos de paralelismo y por tanto se han desarrollado diferentes lenguajes de programación para explotarlos.

### 3.2.4.2 Paralelismo implícito y explícito

**Explícito:** El algoritmo paralelo debe especificar explícitamente cómo cooperan los procesadores para resolver un problema específico. La tarea del compilador es sencilla, en cambio la del programador es bastante difícil.

**Implícito:** Se usa un lenguaje de programación secuencial y el compilador inserta las instrucciones necesarias para ejecutar el programa en una computadora paralela. El compilador tiene que analizar y comprender las dependencias para asegurar un mapeo eficiente.

### 3.2.4.3 Diseño de Algoritmos Paralelos

Diseñar algoritmos paralelos no es tarea fácil y llega a ser un proceso altamente creativo. Inicialmente se deben explorar los aspectos independientes de la máquina y los específicos dependientes de la máquina se dejan para más tarde. El diseño involucra cuatro etapas las cuales se presentan como secuenciales pero que en la práctica no lo son.

- **Partición:** El cómputo y los datos sobre los cuales se opera se descomponen en tareas (Granularidad). Se ignoran aspectos como el número de procesadores de la máquina a usar y se concentra la atención en explotar oportunidades de paralelismo.
- **Comunicación:** Se determina la comunicación requerida para coordinar las tareas. Dependiendo de las herramientas que utilicemos se pueden definir estructuras y algoritmos de comunicación.
- **Agrupación:** El resultado de las dos etapas anteriores es evaluado en términos de eficiencia y costos de implementación. De ser necesario, se agrupan tareas pequeñas en tareas más grandes.
- **Asignación:** Cada tarea es asignada a un procesador tratando de maximizar la utilización de los procesadores y de reducir el costo de comunicación. La asignación puede ser estática (se debe establecer antes de la ejecución del programa) o en tiempo de ejecución mediante un algoritmo de balance de carga.

### 3.2.4.4 Sincronización

Este tema tiene que ver con la forma de evitar que uno o más procesos se solapen en lapsos o en segmentos que contengan demandas precisamente sobre los mismos recursos, por ejemplo: usar la o las mismas variables globales, usar el mismo dispositivo de Entrada o Salida, entre otros. Un caso elemental de este tipo de conflicto lo constituye el intento

simultáneo, por parte de más de un procesador de acceder al mismo bloque de Memoria Principal.

En general, mediante mecanismos de hardware, cada vez que un procesador consigue el acceso a memoria principal a través del bus, todas las demás deben tener bloqueada esa posibilidad. El bus precisamente actúa como uno de los medios para administrar los múltiples intentos de acceso a la memoria principal, al menos durante los ciclos en que este ocupado.

### 3.2.4.5 Latencia

En la programación paralela, principalmente con los cluster de memoria distribuida, utilizamos conjuntos de datos bastante grandes, que en ocasiones tienen que viajar de un nodo a otro en la red. Uno de los problemas principales para obtener un mayor rendimiento en las aplicaciones es la *latencia*.

La latencia es el lapso que se ocupa en la transacción de información, desde la solicitud hasta la recepción de los datos. Trabajando con datos masivos la latencia se presenta de dos formas: al acceder a los datos en el disco duro o memoria y en la comunicación entre los procesadores.

Para que una aplicación sea efectiva debe tener una latencia baja en estos aspectos. Existen varias técnicas para minimizar el efecto de la latencia sobre el rendimiento de una aplicación. Una es el acceso por bloques que sean relativamente grandes, esto es, tomar grandes bloques de información en lugar de partes pequeñas así tendremos pocas solicitudes de gran cantidad de información en lugar de gran cantidad de peticiones por pocos datos, el motivo principal es la reducción de los tiempos desde la petición hasta la recepción de la información, así una sola petición de muchos datos tiene menos tiempos de solicitudes, confirmación, el acceso a los datos, el envío y la recepción de datos, que hacerlo por partes.

### 3.2.4.6 Aceleración Y Eficiencia

Dos de las medidas más importantes para apreciar la calidad de la implementación de un algoritmo paralelo en multicomputadoras y multiprocesadores son la aceleración y la eficiencia. La aceleración de un algoritmo paralelo ejecutado usando  $n$  procesadores es la razón entre el tiempo que tarda el mejor algoritmo secuencial en ser ejecutado usando un procesador en una computadora y el tiempo que tarda el correspondiente algoritmo paralelo en ser ejecutado en la misma computadora usando los mismos  $n$  procesadores. La eficiencia de un algoritmo paralelo ejecutado en  $n$  procesadores es la aceleración dividida por  $n$ . La ley de Amdahl permite expresar la aceleración máxima obtenible como una función de la fracción del código del algoritmo que sea paralelizable, significa que es el algoritmo el que decide la mejora de velocidad, no el número de procesadores. Finalmente se tiene que llegar a un límite donde no se puede paralelizar más el algoritmo.

# Capítulo 4

## Charm++

Charm++ es una librería que en tiempo de ejecución que permite a objetos de C++ comunicarse unos con otros de manera eficiente utilizando el paradigma de paso de mensajes (similar a **MPI**) de forma asíncrona y puede ser utilizado tanto en máquinas con memoria distribuida como máquinas con memoria compartida. El modelo de programación es similar a CORBA, Java RMI o RPC donde se realizan invocaciones de métodos remotos, pero dirigida hacia máquinas paralelas de alto rendimiento.

Se ha demostrado la escalabilidad de Charm++ hasta mil procesadores con un balance de carga y migración de objetos efectiva. El paralelismo en Charm++ es escrito de forma explícita, es decir, el lenguaje no transforma código serial a paralelo automáticamente. Sin embargo esto no representa un problema grande ya que un programa serial puede ser dividido en varias partes para su procesamiento en paralelo con pocos cambios en el código.

### 4.1 Historia de Charm++

El proyecto Charm fue desarrollado por el grupo investigación del Laboratorio de Programación Paralela de la Universidad de Illinois en Urbana-Champaign. El seguimiento de este proyecto dio como resultado el lenguaje de programación Charm++, diseñado para nuevas computadoras y nuevos paradigmas de programación; y esencialmente para facilitar el desarrollo de nuevas aplicaciones de procesamiento paralelo. El primer prototipo desarrollado a finales de la década de los ochenta fue denominado **Chare Kernel 1.0**, desarrollado por Wennie Shu y Kevin Nomura que trabajaban con Laxmikant Kale. Este **Kernel** consistía solamente en una serie de librerías con métodos básicos de invocación remota.

El segundo prototipo **Charm Kernel 2.0** estuvo a cargo del equipo formado por Wayne Fenton, Balkrishna Ramkumar, Vikram Saletore, Amitabh B. Sinha y Laxmikant Kale y el traductor fue escrito por Manish Gupta. Esta versión fue reescrita totalmente

teniendo cambios mayores respecto de su antecesor, incluyó extensiones del lenguaje C para denotar *chares*, mensajes e invocaciones remotas asíncronas. Charm 3.0 estuvo enfocado a mejorar su sintaxis y contiene una característica importante como es la abstracción de información compartida y los *chare groups* llamados anteriormente **Branch Office Chares (BOC)**. En esta ocasión el proyecto estuvo a cargo de Attila Gursoy, Balkrishna Ramkumar, Amitabh B. Sinha y Laxmikant Kale, con un nuevo traductor escrito por Nimish Shah.

Fue hasta 1993 cuando apareció el Kernel 4.0 el cual fue creado teniendo como base el lenguaje de programación C++ con lo cual cambio su nombre a Charm++ y la programación a orientada a objetos paralelos, este empleaba un traductor que analizaba sintácticamente todo el código C++ mientras lo traducía a su nueva sintaxis con las extensiones. En 1999 fue formalmente renombrado como **Charm Kernel** y aparece la versión 5 la cual fue rescrita en su totalidad en C++ y para el año 2000 fueron agregadas las funcionalidades de balance de carga y migración de objetos entre procesadores. Actualmente se encuentra en la versión 6 con pocos cambios respecto de la versión 5 como ofrecer compatibilidad con las plataformas BlueGene/P, PPC Linux, Windows Xp de 64 bits, Infiniband IBVERBS y Myrinet MX entre otras además se mejora el rendimiento en la comunicación entre maquinas SMP.

## 4.2 Introducción a Charm++

Desde la perspectiva de un programador una aplicación creada con Charm++ es simplemente una colección de objetos llamados *chare*<sup>1</sup> que realizan alguna tarea en conjunto. Cada objeto tiene un estado asociado a él un estado o una información y se comunican entre sí mediante el paso de mensajes, cuando un objeto en particular recibe un mensaje, este ejecutará algún *método de entrada* para procesar dicho mensaje. Este *método* puede realizar uno o más cálculos u operaciones, y también tiene la capacidad de enviar mensajes a otros objetos *chare*, además de poder almacenar el contenido de un mensaje para ejecutarlo posteriormente, o simplemente quedarse en un estado de espera sin realizar tarea alguna. Así un objeto *chare* puede enviar un mensaje a otro objeto que a su vez puede enviar otro mensaje a otro objeto y este también enviar un nuevo mensaje a otro objeto diferente y así sucesivamente. Este es el punto que le da más valor a una aplicación Charm++ debido a que no depende del número de procesadores tampoco del número tareas que se tienen por procesador, ni del tipo de operaciones que tenga que realizar o incluso de cómo un dato es enviado a otro procesador, en la aplicación solo tenemos un grupo de objetos *chare* la demás información como el total de procesadores que se podrán utilizar o el tipo de interconexión que se utilizara estará disponible hasta la ejecución de la aplicación y para ello esta destinado lo que llamaremos el **Sistema de Ejecución de Charm++**. Este sistema funciona como administrador de cada uno de los objetos *chare* para su procesamiento físico en los procesadores, ya que redirige redirige los mensajes sobre la interconexión física para llegar a su destino, esto queda fuera de la vista del programador que solo se encargará de decidir que objetos tendrán que comunicarse. A continuación presentamos de forma visual esta representación de los objetos desde la visión programador y su interacción sobre varios procesadores.

<sup>1</sup> Definición para las tareas diarias domesticas o del campo en ingles antiguo, también significaba callejón.

Ahora vamos a definir algunos conceptos que identifican el entorno de Charm++:

*Paradigma de programación:* Al referirnos a paradigma queremos decir el esquema que se utiliza para escribir un programa. Charm++ no es un lenguaje de programación por sí mismo, si no que utiliza al lenguaje C++ como su base o plataforma y Charm++ son librerías que agregan una funcionalidad y una estructura sobre la cual el lenguaje C++ puede realizar aplicaciones o programas paralelos.

*Orientado a Objetos:* Un programa es dividido en un conjunto de objetos que interactúan entre sí. En **Charm++** son llamados *chares* y pueden contener algún dato, enviar y recibir mensajes, y además podrán ejecutar alguna tarea como respuesta a un mensaje recibido (esto es, ejecutar algún método o función mediante la llamada *entry method* o método de entrada).

*Paso asíncrono de mensajes:* los objetos *chare* se comunican entre si mediante mensajes (como en MPI). Por mensaje asíncrono se define un mensaje que no esta vinculado a los objetos *chare*, esto quiere decir, que si el código que se ejecuta sobre un *chare* decide enviar un mensaje a otro, una vez que envía un mensaje el código continua su ejecución sin la necesidad de recibir alguna respuesta por parte *chare* del destino. En sentido contrario se cumple de igual manera se puede recibir un mensaje sin afectar la continuidad en la ejecución del objeto que lo esta enviando.

Cada aplicación en Charm++ debe contar con ciertos elementos, a continuación mencionamos en general estos, posteriormente los describiremos con mayor detalle.

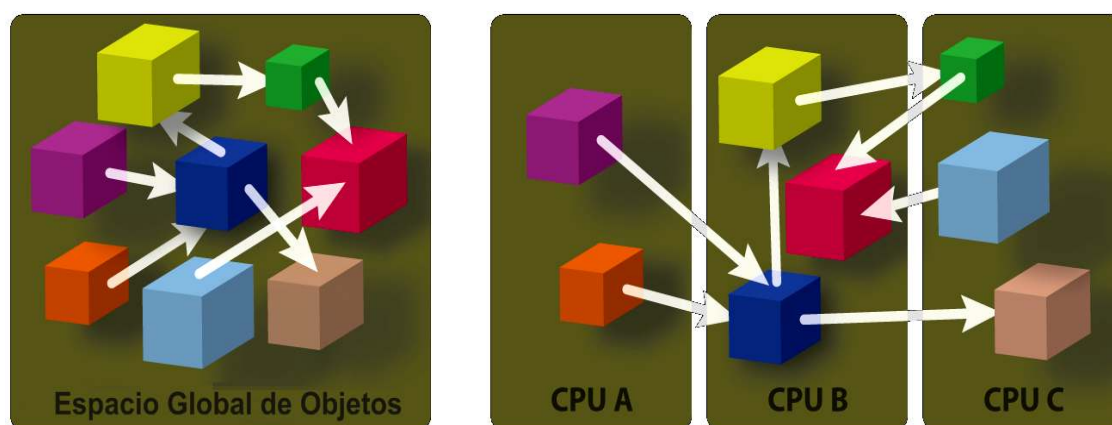


Fig. 4.1 Representación de los objetos en un cluster y en cada procesador

### 4.2.1 Objetos Chare

Cada objeto *chare* en una aplicación Charm++ puede formarse de varias entidades independientes. Cada objeto tiene su propio estado o información. Los objetos *chare* en la aplicación no tienen acceso directo al estado de otro objeto, pero pueden comunicarse directamente con él, a través de un mensaje. El conjunto general de objetos es referido como

**Espacio Global de Objetos.** Cualquier objeto *chare* en el espacio puede pedir información a otro objeto mediante el uso de mensajes.

Una clase *chare* define un tipo de objeto *chare* y un objeto *chare* es una instancia específica de esta clase. Un arreglo de *chares* es un arreglo de objetos *chare* (similar a un arreglo de enteros o caracteres en C++).

Existe un objeto especial llamado *main chare object* al cual nos referiremos solamente como *main*. El objeto *main* actúa como el punto de entrada o de inicio de una aplicación Charm++, de la misma forma que la función *main* en una aplicación en C++.

## 4.2.2 Mensajes

Los objetos *chare* se comunican unos con otros mediante el envío de mensajes. En Charm++ este proceso es referido también como la *invocación de un método remoto* tanto el envío de llamadas simples como la recepción de métodos de entrada. Desde la perspectiva del programador el código básicamente se presenta como una llamada a una función o método miembro de algún objeto *chare*.

## 4.2.3 Métodos de entrada

Los métodos de entrada son funciones especiales de una clase *chare*. La diferencia entre una función C++ y un método de entrada Charm++ radica en la recepción de mensajes (en otras palabras son métodos que pueden ser invocados de manera remota). Cuando un objeto realiza una invocación remota a un método, los datos a enviar son *empacados* dentro de un mensaje y enviados para que los reciba el otro objeto. Una vez recibido, el método que invocado es ejecutado y los datos del mensaje son *desempacados* por ese método de entrada.

Hay dos aspectos que hacen diferente un método de entrada a una función de C++: primero los métodos de entrada no devuelven información o datos; segundo, desde la perspectiva del objeto con el que se comunica, el método de entrada regresa inmediatamente el control al punto del código donde fue invocado sin esperar alguna respuesta, permitiendo que continúe con su ejecución. Esto no significa que el método haya sido ejecutado, solamente que el mensaje fue entregado y en algún momento posterior será ejecutado.

## 4.2.4 Proxies

Debido a que los objetos del espacio en Charm++ pueden estar dispersos a través de varios procesadores no siempre es posible para 2 objetos comunicarse directamente. Para estos casos si un objeto desea enviar un mensaje a otro primero tendrá que hacer referencia a lo que llamamos un *Proxy*. El *Proxy* es un objeto que oculta detalles de la comunicación al programador haciendo transparente este proceso. Los *métodos de entrada* son llamados sobre

un objeto **Proxy** como si se tratara de otro objeto “*físicamente*” en el mismo procesador y es una referencia a un objeto **chare** similar a usar un puntero, la diferencia radica en que esta referencia es válida sobre todos los procesadores.

## 4.2.5 Colecciones de Chare

En las aplicaciones Charm++ es más común ver colecciones de objetos que objetos individuales. Esta colección es esparcida a través de todos los procesadores y todos los miembros actúan como si fueran operaciones individuales similares aplicadas sobre elementos de un conjunto de datos.

Cada colección dentro de un programa es representada por un único identificador (*handle*). Este identificador es visible para todos los procesadores e incluso puede enviarse como a otro procesador como información, además puede ser utilizado para tener acceso a las funciones de los objetos **chare** de esa colección. Existen varios tipos de colecciones de **chares**.

### 4.2.5.1 Arreglos de Chares

Un arreglo es simplemente un conjunto de objetos **chare**. Los arreglos de **chares** pueden ser indexados de forma similar a un arreglo en C++ y estos pueden ser uni o multidimensionales. También pueden ser indexados usando esquemas más complejos tales como un vector o utilizar índices definidos por el propio usuario.

Inicialmente, los elementos del arreglo **chare** son esparcidos de acuerdo a un esquema de mapeo definido por el Sistema de Ejecución de Charm++. Sin embargo, debido a que el arreglo de objetos puede ser migrado, estos pueden moverse entre procesadores o sobre uno solo. El Sistema de Ejecución Charm++ puede realizar esta migración y se encarga de equilibrar la carga de procesamiento sobre todos los nodos. El programador no se tiene que preocupar de conocer donde se encuentra un elemento particular del arreglo aunque este sea migrado de un procesador a otro, siempre se hará referencia con el mismo nombre y el sistema se hace cargo de redireccionar esta referencia hacia el procesador apropiado.

### 4.2.5.2 Grupos de Chares

Un grupo de **chares** es una colección donde físicamente cada procesador disponible para una aplicación tiene un representante de un grupo de objeto **chare** que esta localizado en este.

### 4.2.5.3 Grupos nodo

Un grupo nodo de **chares** es similar a los grupos de **chares**, con la única diferencia que se tiene un representante en cada nodo en lugar de uno en cada procesador. Existen computadoras que tienen físicamente más de un procesador en este caso si un nodo tiene 4

procesadores habrá solo un representante del grupo nodo en cualquiera de esos procesadores, no existe una regla que defina en cual se encontrará.

### 4.3 Sistema de ejecución de Charm++

Cuando un programador escribe una aplicación Charm++ lo hace en términos de objetos que se comunican unos con otros a través de la invocación de métodos (paso de mensajes). Los detalles tales como el tipo y número de procesadores, el tipo de interconexión y demás no son considerados a la hora de programar, el programador simplemente escribe la aplicación como una colección de objetos que van a interactuar entre ellos. Alternativamente cuando una aplicación es compilada existe una plataforma específica que incluye el tipo de procesador, la interconexión y las demás características que son requeridas.

Adicionalmente, cuando una aplicación es ejecutada, existe un conjunto de *recursos físicos* que deben estar disponibles para la aplicación tales como el número de procesadores, la interconexión y otros puntos similares. El propósito del Sistema de Ejecución de Charm++ es manejar los detalles referentes a los recursos físicos en lugar de la aplicación y por consiguiente en lugar del programador mismo. Entre las decisiones que toma esta administración se encuentran:

- Mapeo de Objetos **Chare** hacia los procesadores físicos: A través de varios métodos el Sistema de Ejecución puede asignar objetos a cada procesador.
- Balance de carga de objetos **chare**: El Sistema puede dinámicamente migrar objetos **chare** físicamente entre procesadores permitiendo a la aplicación utilizar de manera más eficiente los procesadores disponibles.
- Redirección de mensajes: Los objetos **chare** son designados físicamente en un procesador, pero además pueden ser migrados a otro, entonces el Sistema maneja un enlace o una liga a los objetos que se encuentren activos. De esta forma los mensajes que son enviados a un objeto que fue migrado, dinámicamente son redirigidos al procesador físico donde se encuentre en ese momento.
- Tolerancia a fallos: Si un procesador físicamente tiene problemas o se cae, el Sistema dinámicamente vuelve a crear los objetos **chare** del procesador caído y los distribuye entre los que estén disponibles.
- Re-ubicación dinámica de recursos físicos: En un cluster puede haber un procesador que reciba bastante trabajo. El sistema de ejecución puede migrar objetos hacia otros procesadores físicos permitiendo a la aplicación disminuir la carga de ese procesador.

Cada procesador tiene su propio Sistema de Ejecución corriendo, así cada instancia tiene su propio responsable local y puede comunicarse con otro para realizar una o varias tareas en conjunto.

El Sistema De Ejecución consta de un conjunto de componentes los cuales mencionaremos a continuación:

### 4.3.1 La capa de Máquina

La capa mas baja del sistema es la capa de máquina, la cual contiene código específico del Sistema de Ejecución y es utilizado para realizar tareas básicas como el envío de mensajes pero de forma más transparente. Cuando el Sistema es instalado en la computadora el usuario necesita especificar el tipo de máquina que será utilizado.

### 4.3.2 Converse

La siguiente capa es llamada *Converse*. Converse abstrae las capacidades de la capa máquina hacia la capa mas alta del sistema de ejecución, también provee de varias funcionalidades no soportadas por algunas máquinas en específico. Adicionalmente Converse contiene el administrador Charm++ y la cola de mensajes.

### 4.3.3 Cola de mensajes

La cola de mensajes actúa como una colección de puntos de inicio hacia un procesador particular, para los mensajes que llegan. La cola de mensajes es compartida por todas las *chares* que son localizadas sobre un elemento procesado. Para mensajes básicos, la cola de mensajes puede ser visualizada como una simple *cola de procesos* esperando su ejecución.

Existen otros tipos de mensajes soportados por Charm++: mensajes expeditos, mensajes inmediatos y mensajes con prioridades definidas por el usuario. Cuando estos tipos son utilizados por una aplicación la cola de mensajes cambia su funcionamiento y de forma que puede dejar de actuar como una simple cola.

### 4.3.4 Planificador

El planificador es un componente de Converse que tiene a su cargo el cuidado de seleccionar un mensaje de la cola y enviarlo a su ejecución. La entrada del mensaje en la cola representa una tripleta (mensaje, objeto *chare* y método de entrada a ejecutar). Una vez que el planificador ha elegido un mensaje para procesar, este inicia su ejecución que esta asociada a un método de entrada. Este método de entrada continúa su ejecución hasta que es completada y devuelve el control al Sistema de Ejecución. Con lo cual se garantiza que solamente un método de entrada será ejecutado a la vez sobre un objeto. Una ventaja de este

enfoque es que una aplicación evita tener que colocar candados a una variable por el acceso de múltiples métodos de entrada.

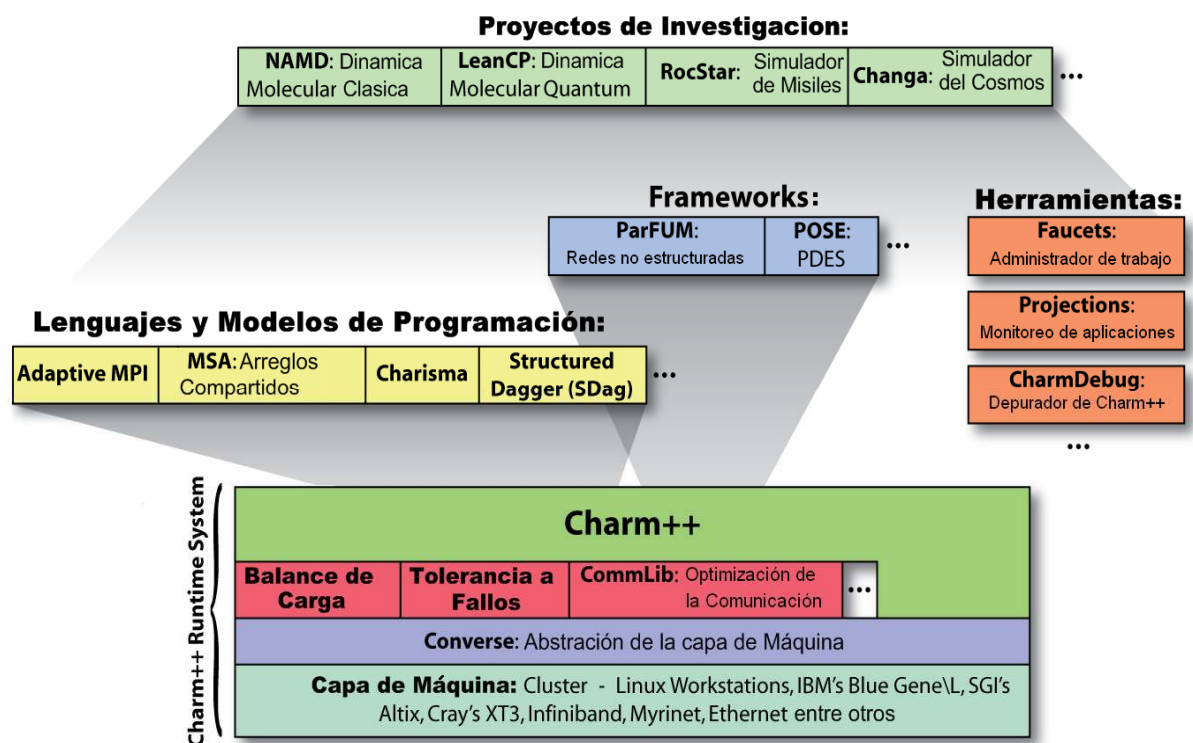


Fig. 4.2 Diagrama General de la composición de Charm++

## 4.4 Proceso de compilación y ejecución de una aplicación

Cada clase en un programa C++ necesita dos cosas, una definición de clase y un cuerpo de función para sus funciones miembro. La definición de la clase es donde son declaradas las funciones y las variables a utilizar. El cuerpo de una función contiene el código que será ejecutado cuando sea invocada. En una aplicación estos dos puntos son tomados en archivos separados, la definición de clases se encuentra en un archivo de cabeceras (con extensión “.h”) y las funciones en el código fuente (con extensión “.C” o “.cpp”).

Las clases *chare* necesitan una pieza adicional de información, la definición de la interfaz. Esta definición especifica cual función puede ser invocada remotamente por otro objeto *chare* en el espacio global de objetos. Esta información está contenida en el archivo de interfaces (con extensión “.ci”).

### 4.4.1 Charmc

*Charmc* es la herramienta de Charm++ que envuelve al *compilador nativo*. Por compilador nativo queremos decir cualquier compilador que sea instalado para un tipo de máquina específica donde las aplicaciones Charm++ vayan a ser compiladas. El propósito de *charm* es permitir que los archivos *makefiles* sean utilizados como un compilador portable de C++. *Charmc* esta disponible sobre todas las plataformas que el Sistema de ejecución Charm++ permite (más adelante revisaremos el listado de plataformas disponibles).

### 4.4.2 Charmxi y el archivo interfaz

El archivo interfaz es procesado por una herramienta de Charm++ llamada *charmxi*, dando como resultado el conjunto de archivos con extensión "decl.h" y "def.h". Estos archivos contienen código autogenerado que liga el código de la aplicación al Sistema de ejecución Charm++. Este código es después incluido o asociado al archivo fuente al momento de ser compilado agregando las línea *#include* mas los archivos generados, y es compilado como una clase en C++. El resultado es un archivo objeto que contiene las clases *chare*.

Después las clases *chare* para una aplicación tienen que ser compiladas, *charm* es una vez más utilizado para ligar todos los archivos. Es durante el proceso de ligado que el *Sistema de Ejecución Charm++* es ligado a la aplicación.

### 4.4.3 Ejecutando un programa Charm++

Cuando *charm* crea el archivo ejecutable, otro programa llamado *charmrun* será creado en el mismo directorio como ejecutable. Las aplicaciones Charm++ pueden ser ejecutadas de dos formas. Primero simplemente se corre el ejecutable, con lo cual el programa solamente es ejecutado en la máquina local, como si de un único procesador se tratara. La segunda es ejecutarlo usando el programa *charmrun*.

### 4.4.4 Charmrun

Cuando un programa es ejecutado usando el programa *charmrun*, el número de procesadores a usar debe ser especificado como un argumento en la línea de comandos *+p* junto con el número\_de\_procesadores. Por ejemplo se puede usar el comando como:

```
charmrun +p4 ./miprograma
```

para ejecutar *miprograma* en cuatro procesadores. Los procesadores disponibles son especificados en un archivo llamado *nodelist*.

#### 4.4.5 El archivo Nodelist

El archivo *nodelist* especifica qué procesadores están disponibles para ejecutar un programa usando *charmrun*. Un ejemplo de un archivo *nodelist* es el siguiente:

```
group main ++shell ssh
host alpha
host bravo
host gama
host delta
host alpha
host bravo
```

las líneas empiezan con la palabra **host**, seguido se especifica el nombre del procesador disponible para ser usado por *charmrun*, y serán utilizados en el orden en que aparecen en la lista, por ejemplo si se ejecuta un programa con dos procesadores, los que serán tomados son los procesadores *alpha* y *bravo* si son 4 se tomarán *alpha*, *bravo*, *gama* y *delta*. Si una sola computadora tiene más de un procesador solamente se tendrá que repetir en la lista tantas veces como procesadores tenga pero se continúa respetando el orden. Continuando con el mismo ejemplo si *alpha* y *bravo* tienen dos procesadores y se necesitarán cinco procesadores entonces *alpha* tendrá ejecutando dos tareas mientras los demás solo uno. Si los procesadores que requiere o que se indiquen en al ejecutar *charmrun* son más de los que en la lista de nodos aparecen se volverá a iniciar el proceso en el mismo orden hasta completar los necesarios.

En línea de comando el argumento **++nodelist** puede utilizarse para especificar el archivo o la lista de nodos que se utilizará para ejecutar una aplicación, si no es especificado *charmrun* tomará por defecto */home/user/.nodelist*.

### 4.5 Ejemplo de un programa Charm++

En este ejemplo el objeto **Main** simplemente imprime el mensaje "Hola Mundo" en la pantalla pero lo utilizaremos para describir varias características de una aplicación programada. Los archivos que son utilizados para crear una aplicación Charm++ son:

#### 4.5.1 El archivo de cabecera (main.h)

Este archivo básicamente es lo mismo que un archivo de clases en C++. Existen varias características que serán explicadas a continuación:

Archivo de cabeceras (main.h)

```
#ifndef __MAIN_H__
#define __MAIN_H__
```

---

```

class Main : public CBase_Main {                                (1)

    public:
        Main(CkArgMsg* msg);                                    (2)
        Main(CkMigrateMessage* msg);                            (3)

};

#endif // __MAIN_H__

```

1. Las clases **Chare** son heredadas de una clase padre la cual es autogenerada durante la compilación. No es necesario que la *Main Chare* se llame *Main*, solamente debe ser especificada en el archivo de interfaces (.ci) usando la palabra clave **mainchare**, pero puede tener el nombre que el programador desee.
2. Este es el inicio o el punto de entrada hacia la aplicación. Este constructor es automáticamente llamada por el Sistema de Ejecución de Charm++, cuando la aplicación arranca. **CkArgMsg** es un mensaje por el cual pasan hacia la aplicación los argumentos por la línea de comandos que especifique el usuario. En Charm++ los argumentos son sustituidos con mensajes.
3. Este es un constructor que es definido para la migración de objetos entre procesadores.

#### 4.5.2 El archivo Fuente (main.C)

Es prácticamente lo mismo que el código fuente para una aplicación en C++.

##### Archivo Fuente (main.C)

```

#include "main.decl.h"                                          (6)
#include "main.h"

// Inicio de una aplicación en Charm++
Main::Main(CkArgMsg* msg) {

    // Imprime "Hola Mundo" en la pantalla del procesador 0
    CkPrintf("Hola Mundo!\n");                                (4)

    // Finaliza la ejecución de la aplicación                  (5)
    CkExit();

}

// Este es el constructor para la migración de objetos
// Este constructor no es necesario que aparezca en el archivo
// interfaz (.ci)

Main::Main(CkMigrateMessage* msg) { }
#include "main.def.h"                                          (6)

```

4. *CkPrintf()* es el equivalente para Charm++ de la función *printf()* de C/C++. En el ejemplo *CkPrintf()* es utilizado para redireccionar las salidas de todos los procesadores hacia un solo punto (el procesador 0) para poder ser visualizado por el usuario.
5. *CkExit()* equivalente para Charm++ de *exit()* en C/C++. En una aplicación Charm++ esta función debe ser llamada para finalizar la ejecución. *CkExit()* causa la finalización de todos los procesos en todos los procesadores donde existan elementos o tareas pendientes por ejecutar.
6. Los archivos *main.decl.h* y *main.def.h* son autogenerados por las herramientas de Charm++ durante el proceso de compilación. Estos contienen código que es utilizado para ligar el código al Sistema de Ejecución De Charm++. El archivo con extensión “*decl.h*” deberá ser incluido al principio del código fuente y el archivo con extensión “*def.h*” al final del mismo.

### 4.5.3 El Archivo Interfaz (main.ci)

```
mainmodule main {                                     (7)

// la Main chare se especifica aquí
  mainchare Main {                                     (8)
    entry Main(CkArgMsg* msg);                         (9)
  };
}
```

7. Esto declara el módulo de interfaz y los archivos que se autogeneran. En este ejemplo la palabra *main* es utilizada para nombrar los archivos generados *main.decl.h* y *main.def.h*.
8. En esta línea se declara la clase *mainchare*. Para este ejemplo la clase *Main* y es declarada como la principal, donde inicia la ejecución la aplicación. El sistema de Ejecución Charm++ creará una instancia de la clase *Main*. Cuando se inicie la ejecución de la aplicación, está empieza por el constructor de la clase *Main*.
9. Esta línea declara una de las funciones (en este caso, un constructor) de la clase *chare* principal como un método de entrada. Esto quiere decir que otros objetos *chare* y el sistema de tiempo de ejecución Charm++ pueden enviar a este objeto *chare* mensajes destinados para la clase *Main* (Los otros objetos *chare* en el espacio global de objetos pueden invocar este método de entrada sobre el objeto *chare* principal).

### 4.5.4 El archivo Makefile

```
CHARMDIR = [Aquí va el directorio raíz Charm++]      (10)
CHARMC = $(CHARMDIR)/bin/charmc $(OPTS)              (11)
```

```

default: all
all: hello

hello : main.o
      $(CHARMC) -language charm++ -o hello main.o          (12)

main.o : main.C main.h main.decl.h main.def.h
      $(CHARMC) -o main.o main.C                          (13)

main.decl.h main.def.h : main.ci
      $(CHARMC) main.ci                                    (14)

clean:
      rm -f main.decl.h main.def.h main.o hello charmrun

```

10. Aquí indicamos la ruta completa donde se encuentra los ejecutables de Charm++, como *charmrun* y *charmcc*.
11. La herramienta *charmcc* funciona como máscara para el *ligador* y el *compilador* nativo del sistema donde se ejecute. Esto permite las instrucciones descritas en un archivo *Makefile* use a *charmcc* como compilador y reutilice *charmcc* sobre varias plataformas haciendo la aplicación portable para varios sistemas.
12. En este caso *charmcc* es utilizado como un *ligador*. Las librerías estándar de C/C++ y las librerías del Sistema de ejecución Charm++ son ligadas en el punto final de la aplicación. Adicionalmente *charmrun* será creado en el mismo directorio el cual permitirá al usuario ejecutar su aplicación sobre múltiples procesadores al mismo tiempo, si se utiliza solamente *./hello* la aplicación se ejecutará solo en la máquina local.
13. Aquí *charmcc* es usado como un compilador, será básicamente solo una llamada al *compilador nativo* crear un objeto el cual será el archivo fuente que corresponde al tipo de máquina donde se ejecutará.
14. En este punto *charmcc* es utilizado para auto generar los archivos *main.decl.h* y *main.def.h*. Estos archivos contienen las ligas necesarias para permitir interactuar el código de la aplicación junto con el Sistema de Ejecución de Charm++.

## 4.6 Salida en pantalla del programa "Hola Mundo"

A continuación presentamos como es la salida en pantalla del programa "Hola Mundo". La primera línea es usada para ejecutar el programa. Debido a que es un solo objeto no tiene caso utilizar varios procesadores. La segunda línea solo indica que el programa ha sido ejecutado en un solo procesador, es decir no se utilizó *charmrun*. La tercer línea es la salida de la sentencia *CkPrintf()* en *Main::Main(CkArgMsg\*)*. Y la cuarta línea es impresa por el Sistema de Ejecución Charm++ para indicar que se ha terminado de ejecutar.

```
$ ./hello
Charm++: standalone mode (not using charmrun)
Hello World!
Program finished.
```

## 4.7 Elementos de una aplicación en Charm++

En la secciones anteriores describimos de forma general las entidades que conforman este lenguaje de programación, ahora presentaremos mas detalles acerca de cada elemento, junto con la sintaxis y en algunos casos un pequeño ejemplo de cómo utilizar dicha entidad.

### 4.7.1 Objetos Chare

Las *chares* son la entidad principal de Charm++ ya que son los encargados de procesar una tarea y de esta forma nos referiremos a estas tareas u objetos, en general son objetos concurrentes con métodos los cuales pueden ser invocados de manera remota, es decir, un objeto *chare* puede pedir a otro que ejecute una de sus funciones sin importar que estén o no en el mismo procesador. Estas funciones son conocidas como métodos de entrada (en ingles *entry method*) y deben ser especificados en el archivo interfaz (con extensión *.ci*) de la siguiente manera:

```
Chare TipoDeChare
{
    Entry TipoDeChare(parametro1);
    Entry void nombreMétodo2(parametro2);
};
```

La definición correspondiente a una *chare* en el archivo de cabeceras (con extensión *.h*) se realiza de la siguiente manera:

```
class TipoDeChare : public CBase TipoDeChare {
    // Aquí van las variables y las funciones como en C++
    // La definición de un métodos de entrada es como sigue
public:
    TipoDeChare (parametro1)
    { // Código en C++ }
    void EntryMethodName2(parametro2)
    { // Código en C++ }
};
```

Las *chares* son objetos concurrentes que encapsulan unidades de trabajo de grano medio, es decir, que tienen un equilibrio entre el procesamiento y la comunicación. Los objetos *chare* pueden ser creados dinámicamente sobre cualquier procesador, y puede haber miles de ellas en un solo procesador, no hay limitaciones más que las del equipo de cómputo. La ubicación de un objeto *chare* es determinada por el sistema de balance de carga dinámico, sin embargo una vez que un *chare* comienza su ejecución no puede ser migrado a otro

procesador. Por defecto un *chare* no tiene un *control de hilos* los métodos de entrada en la ejecución de un *chare* utilizan un manejador de mensajes para recibir un mensaje.

Por definición un método de entrada específico es ejecutado sin interrupción, cuando un mensaje es recibido y programado para su ejecución. Solo un mensaje por *chare* es procesado a la vez. Los métodos de entrada son definidos como una función en C++, con la diferencia que deben regresar un valor vacío o void y solo deben tener un argumento que es un apuntador al mensaje enviado.

Cada instancia tiene un identificador o *handle* que en esencia es un apuntador global y único en todos los procesadores. El identificador de una *chare* tiene un tipo *CKChareID*. La variable *thishandle* identifica al *chare* del cual se esta ejecutando algún método o función, esta variable es muy útil ya que se puede enviar como un argumento en un mensaje, así un *chare* puede enviar su identificador a otra para que pueda conocer donde se encuentra.

#### 4.7.1.1 Creación de una Chare

Primero, una *chare* necesita ser declarada en los archivos .ci y .h. A continuación se muestra un ejemplo de cómo declarar una *chare* llamada tarea, donde *mensaje1* y *mensaje2* son mensajes definidos por el usuario y *entrada* es un método de entrada. En el archivo programa.ci tendremos:

```
module programa {
    chare tarea {
        entry tarea(parametros);
        entry void entrada(parametros);
    };
}
```

y en el archivo programa.h:

```
#include "programa.decl.h"
class tarea : public Chare {
public:
    tarea(parametros);
    void entrada(parametros);
};
```

La variable *parametros* debe corresponder con los parámetros para el constructor del método de entrada. Si el constructor no tiene parámetros se queda vacío.

Una vez declaradas las *chares* se puede usar la clase *CProxy\_tipodeChare* para crear una nueva instancia a una *chare*. Teniendo en cuenta lo anterior, el *Proxy* debe ser del tipo *Cproxy\_tarea* y la creación de un *Proxy* se realiza de la siguiente forma:

```
CProxy_tipodeChare::ckNew(param, CkChareID *apuntVirt,int procDest);
```

Cada argumento es opcional, donde:

- ***tipodeChare*** es el nombre del tipo de ***chare*** que será creado.
- ***apuntVirt*** es un puntero a una identificador ***chare*** del tipo de ***CkChareID***, Que es creado por el método ***ckNew***. Este argumento opcional se puede utilizar si el usuario desea tener un identificador ***virtual*** a la instancia del ***chare*** que será creado.
- ***procDest*** cuando un ***chare*** va a ser creado se puede especificar en que procesador. Lo más conveniente es dejar a ***Charm++*** decidir donde crearlo, pero si el usuario lo desea de esta manera puede ser creado en un procesador específico.

El método de creación de una tarea ***chare*** deposita una ***semilla*** en una cola y regresa inmediatamente a su ejecución. La ***chare*** será creada posteriormente en algún procesador, que será determinado por el sistema de balance de carga. Cuando un ***chare*** es creado, es inicializado por la llamada a su constructor mediante un método de entrada y un mensaje como parámetro. El método no regresa valor alguno pero apunta al identificador virtual de la ***chare*** que será creada.

A continuación se presentan algunos ejemplos de cómo usar este método para la creación de ***chares***.

1. Esto creará un nuevo ***chare*** de “***tipo C***” sobre cualquier procesador:

```
CProxy_C chareProxy = CProxy_C::ckNew(parametros);
```

2. Esto creará un nuevo ***chare*** del “***tipo C***” sobre el procesador ***procesadorDestino*** el cual debe ser un entero positivo:

```
CProxy_C chareProxy = CProxy_C::ckNew(parámetros,
procesadorDestino);
```

3. Lo siguiente crea primero un identificador ***CkChareID*** con el nombre: ***cid***, después crea una nueva tarea ***chare*** del ***tipo C*** sobre el procesador ***procesadorDestino***:

```
CkChareID cid;
CProxy_C::ckNew(parámetros, &cid, procesadorDestino);
CProxy_C chareProxy(cid);
```

#### 4.7.1.2 Invocación de métodos de una chare

Un mensaje debe ser enviado a una ***chare*** usando la sintaxis:

```
Nombre_Del_Chare_Proxy.Método_De_Entrada_1(parámetros);
```

Esto invoca el método de entrada ***Método\_De\_Entrada\_1*** sobre el ***chare*** al que hace referencia ***Nombre\_Del\_Chare\_Proxy***, ésta llamada es asíncrona y no bloqueante y regresa a su ejecución inmediatamente después de enviar el mensaje.

### 4.7.1.3 Acceso local

Se puede tener acceso a un *chare* local usando el método de Proxy *ckLocal*, el cual devuelve un puntero de C++ a una *chare* local si esta existe en el procesador local, y NULL en caso contrario o si se encuentra en otro procesador.

```
C *tarea=chareProxy.ckLocal();
    if (tarea==NULL) {
//...si no se encuentre en este procesador, enviar mensaje.
    } else {
//...si es local utilizar directamente los métodos de "tarea".
    }
```

### 4.7.2 Envío de Parámetros

En Charm++, las *chare*, grupos y grupos nodo se comunican invocando a alguno de sus métodos o funciones. Los métodos pueden tener varios parámetros o un objeto mensaje que describiremos más adelante. En sentido literal llamaremos mensaje a un objeto mensaje o a un conjunto de parámetros enviados a través de la red.

Por ejemplo, una *chare* pudiera tener éste método de entrada en la declaración del archivo interfaz (.ci):

```
entry void tarea(int i, int k);
```

Entonces la invocación de tarea(2,3) sobre el Proxy de una *chare* invocará tarea(2,3) en esa *chare remota*. Como Charm++ se puede ejecutar sobre máquinas con memoria distribuida, no es posible enviar un arreglo mediante un puntero como en C++, en su lugar debemos especificar el tamaño del arreglo en el archivo de interfaz (.ci) también :

```
entry void tarea(int i, double arr[n]);
```

Como C++ no reconoce esta sintaxis, los datos del arreglo deben enviarse a un Proxy *chare* como un puntero. El arreglo será copiado y enviado al procesador destino, donde la *chare* recibirá una copia mediante este puntero. La copia remota de los datos permanecerá disponible hasta que el método remoto regrese a donde fue invocado, que será justo cuando sea liberada la memoria. Esto significa que cualquier modificación hecha a los datos de manera local después de la invocación no será visualizada por la *chare* remota y los datos se perderán al regresar a donde fue invocada. Esto significa también que los datos deben ser copiados al procesador destino. En arreglos grandes, esto es muy poco eficiente respecto al método de paso de mensajes.

Los arreglos pueden ir como parámetro de varias formas.

```
entry void dibujaLinea(float datos[n], int n);
entry void dibujaPlano(float datos[n*n], int n);
entry void dibujaEspacio(int n, int m, int o, float datos[n*m*o]);
entry void dibujaTodo(int nd, int dims[nd], float datos[product(dims, nd)]);
```

La expresión entre corchetes puede ser cualquier expresión válida en C++, incluyendo constantes, y puede depender de algún modo de los parámetros pasados o incluso de variables o datos globales.

### 4.7.3 Mensajes

En Charm++ hablar de un mensaje no es lo mismo que hablar del paradigma de paso de mensajes, es decir, un mensaje no es el medio para comunicarse sino que un mensaje encapsula los parámetros que serán enviados a un método de entrada. Los datos que son encapsulados en un mensaje, regularmente es más eficiente que el paso de parámetros, además que los mensajes son más fáciles de colocar en una cola y almacenar en el procesador que los recibe, por el Sistema de Ejecución de Charm++. Los métodos de entrada permiten pasar a los mensajes como un parámetro. En el lado donde es recibido, el método de entrada debe almacenar el mensaje o borrarlo si no va a ejecutar tarea alguna, en el caso de no realizar alguna tarea y no borra el mensaje este nunca será destruido y estará utilizando espacio en memoria.

Hay muchos tipos de mensajes disponibles, regularmente los mensajes en Charm++ son de tamaño fijo. Se pueden tener mensajes que contengan punteros o arreglos grandes (especificando el tamaño del mismo) y aun esos punteros serán válidos cuando los mensajes sean enviados a los demás procesadores. También existe un mecanismo para asignar una prioridad a cualquier tipo de mensajes.

Al igual que las demás entidades que se utilizan en una invocación asíncrona (*chares*, *groupchare*, array *chares*), los mensajes deben ser declarados en el archivo de interfaz (.ci). Un mensaje es declarado de la siguiente forma:

```
message TipoDeMensaje;
```

Un mensaje que contiene arreglos es declaradas como:

```
message TipoDeMensaje {
    tipo1 nombre_var1[];
    tipo2 nombre_var2[];
    tipo3 nombre_var3[];
}
```

donde tipo es el tipo de variable a declarar puede ser *int*, *float* o *double* por ejemplo.

Si el nombre de la clase *message* es *TipoDeMensaje*, la clase debe heredar una clase pública con nombre *Cmessage\_TipoDeMensaje*. Esta clase es generada por el traductor Charm++. Entonces la definición debe quedar de la siguiente manera:

```
class TipoDeMensaje : public Cmessage_TipoDeMensaje {
    // Lista de datos y funciones como en C++
}
```

### 4.7.3.1 Creación y borrado de mensajes.

Los mensajes reservan espacio en memoria usando el operador *new* de C++:

```
TipoDeMensaje *msgptr = new[(int tam1, int tam2,...,int priobits=0)]
TipoDeMensaje[(argumentos para constructor)];
```

Los argumentos opcionales para el operador *new* son utilizados cuando se va a reservar espacio de mensajes que contengan arreglos muy grandes o dar prioridad al mensaje. Las variables *tam1*, *tam2*,... son el tamaño de los bloques de memoria que serán reservadas y asignadas a los punteros que esos mensajes contengan, el argumento “*priobits*” denota el tamaño de un campo de bit (número de bits) que serán usados para almacenar la prioridad del mensaje.

Por ejemplo, para reservar espacio de un mensaje cuya declaración de clase es:

```
class Message : public Cmessage_Message {
    //.. datos y métodos para el mensaje
};
```

solo hay que escribir lo siguiente:

```
Message *msg = new Message;
```

Ahora para reservar espacio en memoria de un mensaje cuya clase es de la siguiente forma:

```
class VarsizeMessage : public Cmessage_VarsizeMessage {
public :
    int *primer_arreglo;
    double *segundo_arreglo;
};
```

realizamos lo siguiente:

```
VarsizeMessage *msg = new(10, 20) VarsizeMessage;
```

Esto quiere decir que se creará *VarsizeMessage* con la variable *primer\_arreglo* será de 10 enteros y que *segundo\_arreglo* será de 20 elementos de tipo flotante.

Para agregar el campo bit de prioridad a este mensaje se usa:

```
VarsizeMessage *msg = new(10, 20, sizeof(int)*8) VarsizeMessage;
```

más adelante veremos como utilizar las prioridades.

Cuando un mensaje es enviado a una *chare*, el programador pierde el control sobre este, en tal caso el espacio de memoria reservado es liberado por el sistema automáticamente. Ahora si un mensaje es recibido en un punto de entrada este no es liberado por el Sistema de

Ejecución de Charm++ (*Runtime System*) y puede ser reutilizado o eliminado por el programador. Los mensajes se eliminan utilizando el operador de C++ *delete*.

#### 4.7.3.2 Mensajes con arreglos de tamaño variable

Un mensaje en Charm++ tiene un tamaño establecido que es reservado internamente. Pero en muchas ocasiones el tamaño de los datos no puede ser conocido si no hasta la ejecución del programa. En estos casos se pueden utilizar mensajes con variables empaquetadas lo cual puede resolver algunos problemas puntuales, sin embargo, esto requiere reservar múltiples espacios de memoria. Esta situación puede evitarse utilizando mensajes del tipo variable o “*varsize*”. Para estos mensajes, el espacio requerido para el tamaño de los arreglos que es variable, es reservado con un mensaje hasta que este es creado en la ejecución.

La declaración en el archivo interfase (.ci) es la siguiente:

```
message mtype{
    tipo1 nombre_var1[];
    tipo2 nombre_var2[];
    tipo3 nombre_var3[];
}
```

La clase *mtype* hereda los métodos y funciones de *CMessage\_mtypee*. Además, ha de contener las variables correspondientes apuntando a los tipos correctos. Si alguna de estas variables deben ser “*private*” o “*protected*”, esto se debe declarar en la clase *CMessage\_mtype* como una clase “*friend*” (amigo). Así la declaración de la clase *mtype* debe ser de la siguiente forma:

```
class mtype : public CMessage_mtype {
    private:
        tipo1 *nombre_var1;
        tipo2 *nombre_var2;
        tipo3 *nombre_var3;
        friend class CMessage_mtype;
}
```

Y en el archivo de interfaz debe ser declarado como:

```
message VarsizeMessage {
    int prime_arreglo[];
    double segundo_arreglo[];
};
```

Especificamos los tipos y nombres actuales de los campos que contienen los arreglos grandes. Las dimensiones de estos arreglos no son especificadas en el archivo interfaz, estos serán especificados en el constructor del mensaje cuando sea creado. En el archivo cabecera o en el código fuente (con extensión .h o .C), esta clase es declarada así:

---

```

class VarsizeMessage : public Cmessage_VarsizeMessage {
public:
    int tamanol1;
    int tamanol2;
    int* primer_arreglo;
    double* segundo_arreglo;
    // ... si hay más funciones se colocan después
}

```

El traductor de interfaz genera la clase *CMessage\_VarsizeMessage*, la cual contiene el código correcto para empaquetar y desempacar el mensaje *VarsizeMessage*.

Para reservar el espacio que necesitan los mensajes de la clase *VarsizeMessage* se utilizan los constructores para inicializar y con sus métodos utilizar los datos, siguiendo la sintaxis anterior podemos tener lo siguiente:

```

//el arreglo primer_arreglo tendrá 4 elementos
//el arreglo segundo arreglo tendrá 5
VarsizeMessage* p = new(4, 5, 0) VarsizeMessage;
p->primer_arreglo[2] = 13;
p->segundo_arreglo[4] = 6.7;

```

Otra forma de reservar espacio para un mensaje “*varsize*” es pasar el tamaño en un arreglo en lugar de una lista de parámetros, por ejemplo:

```

int sizes[2]; // arreglo con 2 elementos
sizes[0] = 4 ; // primer elemento 4
sizes[1] = 5; // segundo elemento 5
// se crea el objeto con los tamaños en el arreglo sizes
VarsizeMessage* p = new(sizes,0) VarsizeMessage;
p->primer_arreglo[2] = 13;
p->segundo_arreglo[4] = 6.7;

```

#### 4.7.3.3 Empaquetado de mensajes de tamaño variable

El traductor de interfaz de Charm++ genera una implementación de 3 métodos estáticos para la clase *CMessage\_mtype*. Los métodos tienen las siguientes sintaxis:

```

static void* alloc(int msgnum, size_t size, int* array, int priobits);
static void pack(mtype*);
static mtype* unpack(void*);

```

El método ***alloc*** será llamado cuando se reserva espacio en memoria para el mensaje con el operador ***new*** y el programador no necesita llamarlo explícitamente. Para todos los elementos de un mensaje es reservado un espacio de memoria en el momento que son creados con ***new***, y además no es necesario realizar un llamado a ***new*** para cada el elemento de un mensaje. Esto es diferente de empaquetar un mensaje donde par cada elemento hay que reservar espacio en memoria. Si se desea, no es necesario usar los métodos generados por el traductor de Charm++ y crear los propios. Ahora el método ***alloc*** debe reservar el espacio de memoria usando ***CkAllocMsg*** el cual se utiliza de la siguiente forma:

```
void *CkAllocMsg(int msgnum, int size, int priobits);
```

En el caso de los mensajes variables (*varsize*), los métodos *alloc*, *pack* y *unpack* son generados por el traductor, por ejemplo para una clase *VarsizeMessage* tendremos algo similar a lo siguiente:

```
// se reserva memoria para varmessage método alloc
static void* alloc(int msgnum, size_t size, int* array, int priobits)
{
    int totalsize, first_start, second_start;

    first_start = ALIGN8(size);
    second_start = ALIGN8(first_start + array[0]*sizeof(int));
    totalsize = second_start + array[1]*sizeof(double);
    VarsizeMessage* newMsg = (VarsizeMessage*) CkAllocMsg(msgnum, totalsize,
    priobits);

    // Aqui hacemos que firstArray apunte al final de newMsg en memoria

    newMsg->firstArray = (int*) ((char*)newMsg + first_start);

    // hacemos que secondArray apunte despues del final de firstArray en memoria

    newMsg->secondArray = (double*) ((char*)newMsg + second_start);
    return (void*) newMsg;
}
```

- *array* es un puntero al mensaje de tamaño variable
- *CkAllocMsg* reserva espacio en memoria para un mensaje después de obtener el tamaño de este.
- *size* es el tamaño de memoria que se necesita al momento de compilar.
- Dependiendo de sus valores hay ocasiones en que las direcciones no tienen el límite de 8 bits esto se puede arreglar usando el comando *ALIGN8*.

```
// mandamos el puntero a donde esta el mensaje empaquetado, método pack
static void* pack(VarsizeMessage* in)
{
    // definimos para firstArray un offset desde el principio de in
    in->firstArray = (int*) ((char*)in->firstArray - (char*)in);

    // también para definimos un offset para secondArray
    in->secondArray = (double*) ((char*)in->secondArray - (char*)in);
    return in;
}

// restauramos el mensaje desde el bloque de memoria "sucio"
static VarsizeMessage* VarsizeMessage::unpack(void* inbuf)
{
    VarsizeMessage* me = (VarsizeMessage*)inbuf;

    // enviamos el mensaje a una dirección absoluta de la memoria
    me->firstArray = (int*) ((size_t)me->firstArray + (char*)me);

    // de igual forma para secondArray
    me->secondArray = (double*) ((size_t)me->secondArray + (char*)me);
    return me;
}
```

Los punteros a un mensaje de tamaño variable (*varsize*) pueden tener 2 estados. Al momento de crearlos *Alloc* son punteros de C++ válidos al inicio de un arreglo. Después en *Pack*, se convierten en una dirección de desplazamiento en la memoria (*offset*) desde las direcciones de los punteros variables al inicio de los datos. Y por último *Unpack* restaura los punteros para que apunten a los arreglos variables.

#### 4.7.3.4 Creación de Prioridades

Por defecto Charm++ procesa los mensajes de manera estricta como una cola (FIFO), el primero que llega es el primero en salir. Para la mayoría de los programas esto no tiene inconvenientes. Sin embargo algunos programas necesitan un control más específico sobre el orden en el cual los mensajes deben ser procesados, para lo cual Charm++ permite controlar el encolamiento de cada mensaje.

La llamada más simple disponible para cambiar el orden en el cual es procesado un mensaje es *CkSetQueueing*.

```
Void CkSetQueueing(MsgType message, int queueingtype);
```

donde *queueingtype* es alguna de las siguientes constantes:

```
CK_QUEUEING_FIFO
CK_QUEUEING_LIFO
CK_QUEUEING_IFIFO
CK_QUEUEING_ILIFO
CK_QUEUEING_BFIFO
CK_QUEUEING_BLIFO
```

Las *CK\_QUEUEING\_FIFO* y *CK\_QUEUEING\_LIFO* se utilizan de la siguiente manera:

```
MsgType *msg1 = new MsgType ;
CkSetQueueing(msg1, CK_QUEUEING_FIFO);
MsgType *msg2 = new MsgType ;
CkSetQueueing(msg2, CK_QUEUEING_LIFO);
```

Cuando el mensaje *msg1* llegue al destino, será puesto al final de la cola de mensajes (siguiendo el orden preestablecido). Sin embargo cuando llegue *msg2* será colocado al principio de la cola (por que será manejado como pila).

Las demás opciones son para establecer directamente prioridades. Para dar prioridad a un mensaje es necesario reservar el espacio en el *buffer* del mensaje, mientras se asigna espacio al mismo. Para lograr esto, el tamaño del campo prioridad debe ser especificado como un argumento en el operador *new*. Aunque el tamaño de la prioridad se especifique, siempre es reforzado por un número “s”.

Un puntero a la parte de la prioridad de un buffer se obtiene con la siguiente llamada:

```
unsigned int *CkPriorityPtr(MsgType msg)
```

Hay 32 clases de prioridades que pueden ser dadas a un mensaje: las prioridades de un número entero y *bitvector*. Las prioridades de enteros son sencillas. Uno asigna un mensaje dejando suficiente espacio (en bits) en el mensaje para tomar la prioridad, el cual es un entero. Se guarda la prioridad en el mensaje y por último se informa al sistema que el mensaje contiene una prioridad entera usando *CkSetQueueing*.

```
MsgType *msg = new (8*sizeof(int)) MsgType;
*CkPriorityPtr(msg) = prio;
CkSetQueueing(msg, CK_QUEUEING_FIFO);
```

La constante predefinida *CK\_QUEUEING\_FIFO* indica que este mensaje tiene una prioridad entera y que si hay otros mensajes con la misma prioridad, deben ir en orden *FIFO* (en relación a los demás). De forma similar, un *CK\_QUEUEING\_LILO* esta disponible, pero usando el orden *LIFO*. Hay que tener en cuenta que *MAXINT* es la prioridad más baja y que *NEGATIVE\_MAXINT* tiene la prioridad más alta.

Las prioridades *bitvector* son un poco más complicadas, ya que se compone de una cadena de bits arbitrario la cual representa un número de punto fijo entre 0 y 1. Por ejemplo la cadena “001001” representa el número “.001001” en binario. Al igual que las prioridad entera los número altos representan prioridad baja y los números menores prioridad alta, pero a diferencia de esta, un *bitvector* puede tener una longitud arbitraria por lo tanto la prioridad que representan puede ser de precisión arbitraria.

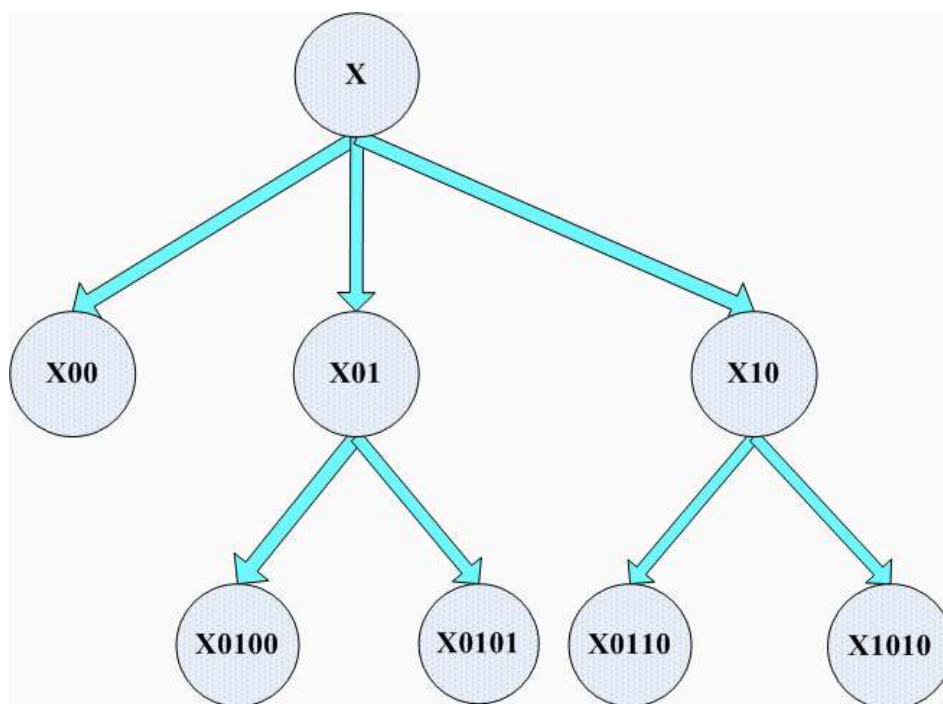


**Fig. 4.3 Prioridades de tipo bitvector en un árbol**

Este tipo de prioridades es útil en las aplicaciones en Inteligencia Artificial para la búsqueda en árboles, digamos tenemos el caso donde un árbol con nodos *N1* y *N2* ahora lo

que deseamos es que la búsqueda tome como prioridad alta a la rama del nodo *N1* con lo cual sus descendientes deben tener también prioridad alta sobre las de la rama *N2* entonces se tendrían que asignar prioridades a cada uno de los descendientes lo cual se logra con los *bitvectores*.

#### 4.7.4 Métodos de entrada

En Charm++ las tareas *chares*, grupos y grupos nodo se comunican entre si usando la invocación de métodos remotos. Estas "*entradas remotas*" pueden tomar sus parámetros de objetos especiales llamados "mensajes". Los mensajes al ser objetos de bajo nivel, más eficientes, más flexibles y también más difíciles de usar que el envío de parámetros.

Un método de entrada siempre es parte de una *chare*, es decir, no existe un método de entrada global en Charm++ como función predeterminada. Los métodos de entrada deben ser declarados en el archivo interfase (.ci) de la siguiente manera:

```
entry void Entrada1(parámetros);
```

Los parámetros pueden ser una lista de variables (por ejemplo *int var1, double var2*) o un mensaje (por ejemplo *MiMensaje \*msg*) que es un puntero. Los métodos de entrada no devuelven datos, por tal motivo son declarados como "*void*" y el constructor de un método de entrada no debe tener algún tipo de dato que devolver. Por último existen métodos de entrada con atributos, uno de ellos es "*sync*" que será descrito mas adelante, con este atributo un método de entrada tiene la posibilidad de regresar un mensaje cuando sea invocado.

##### 4.7.4.1 Atributos de los métodos de entrada.

Charm++ posee una serie de atributos especiales para definir a los métodos de entrada. Para designar uno o más atributos para un método de entrada, se especifica con una palabra clave (*keyword* de una lista que veremos mas adelante) para el atributo deseado. En las declaraciones del archivo de interfase (.ci) la sintaxis para designar un atributo es la siguiente:

```
entry [atributo1,...,atributoN] void Método_De_Entrada(parámetros);
```

A continuación presentamos los tipos de atributos que puede tomar un método de entrada:

- *Threaded*: Este tipo de método de entrada solamente se ejecutan dentro de su propio hilo "*nonpreemptible*", es decir, que no es interrumpido por el sistema. Para hacer un método "threaded" simplemente se añade la palabra clave "threaded" a la lista de atributos de ese método.
- *Sync*: Con éste atributo las llamadas hacia un método "*sync*" son bloqueantes, es decir, no continúa o no regresa control al método o función que lo llamó hasta que

éste no haya finalizado su ejecución totalmente. Los métodos *"sync"* pueden devolver valores, sin embargo estos solo pueden ser mensajes. Para hacer un método *"sync"* se añade la palabra clave *"sync"* a la lista de atributos de ese método.

- *Exclusive*: Este atributo solo esta disponible para los "grupos nodo", son métodos de entrada que no se ejecutan mientras otro método del mismo tipo este ejecutándose sobre el mismo nodo. Si un método *"exclusive"* esta ejecutándose sobre el nodo 0, y otro método esta programado para ejecutarse sobre el mismo nodo, el segundo tendrá que esperar a que el primero finalice totalmente para que pueda iniciar su propia ejecución. Para hacer un método *"Exclusive"* se añade la palabra clave *"Exclusive"* a la lista de atributos de ese método.
- *Nokeep*: Este atributo indica a Charm++ que los mensajes pasados a ese método no serán preservados, es decir, los mensajes que lleguen como parámetros pueden ser eliminados. El sistema de ejecución de Charm++ tiene la capacidad de optimizar y reutilizar la memoria de un mensaje que no será guardado.
- *Notrace*: Este atributo indica a Charm++ que no haga una traza de la ejecución de un método con *trace* de *"PROJECTIONS"* o en el modo *"summary"* resumen.
- *Immediate*: Este tipo de atributo hace que los métodos de entrada sean un mensaje corto que puede ser ejecutado de manera *inmediata* cuando son recibidos. Tales mensajes pueden ser útiles para la implementación de mensajes múltiples y reducciones, también en la búsqueda de datos, o en el caso de un procesamiento de un mensaje que sea crítico y no deba ser retrasado (en el administrador de colas), por otra entrada que tomará más tiempo en ejecutarse. Este tipo de atributo solo esta disponible para los métodos de entrada de un *nodegroup* y además implícitamente lo hace *Exclusive* en cada nodo, por lo cual una vez que se ejecuta un método con el atributo *Immediate* no puede ser interrumpido por otro hasta finalizar. La función *CmiProbellImmediateMsg()* puede ser llamada en el código para probar y procesar un mensaje con atributo *Immediate* periódicamente.
- *Expedite*: Estos son métodos de entrada que pasan por alto la prioridad de Charm++ en la cola de mensajes. Esto es útil para mensajes que requieren ser procesados rápidamente cuando el atributo *"Immediate"* no puede ser aplicado.
- *Inline*: son funciones en las cuales los mensajes son entregados inmediatamente, solo si corresponde a un método que esta en el mismo procesador, en caso contrario se invocará de manera normal. Por lo tanto, estos métodos necesitan ser re-entrantes, como si fueran a ser llamadas múltiples veces de manera recursiva.
- *Local*: este atributo solo esta disponible para grupos y arreglos. Es equivalente a llamar una función que se ejecutará inmediatamente y además debe estar dentro del mismo procesador desde donde se esta invocando. El usuario debe asegurarse que el receptor se encuentra en el mismo procesador, ya que si existe algún error puede finalizar la ejecución de la aplicación.

- *Python*: estos métodos son habilitados para poder ser llamados desde scripts python.

### 4.7.5 Arreglos de Chares

Los arreglos son colecciones de *chares* de tamaño arbitrario. Un arreglo completo tiene un identificador global único de tipo **CkArrayID**, y cada elemento tiene un índice único de tipo **CkArrayIndex** que puede ser un número entero (i.e. un arreglo de una dimensión), varios enteros (i.e. arreglo multidimensional), o una cadena de bytes arbitraria (por ejemplo un índice de árboles).

Los elementos de los arreglos pueden ser creados dinámicamente y destruidos sobre cualquier procesador, y los mensajes para sus elementos llegarán de manera correcta. Los elementos de los arreglos pueden ser migrados en cualquier momento con una balance de carga eficiente y también pueden recibir una llamada a todo el arreglo para realizar una reducción con *contribute*.

#### 4.7.5.1 Declarando un Arreglo Unidimensional (1D)

Un arreglo unidimensional se declara de la siguiente forma:

```
// en el archivo interface (.ci)

array [1D] Arreglo {
    entry Arreglo(parametros1);
    entry void OtraEntrada(parametros2);
}
```

Así como cada *chare* hereda el método “*thishandle*” de *CBase\_nombreClase*, un arreglo hereda “*thisArrayID*”, “*CkArrayID*” para su arreglo y “*thisIndex*” para indexar los elementos del arreglo. También como una *chare* puede heredar de la clase **Chare**, los arreglos pueden heredar métodos y funciones de la clase *ArrayElement1D* si el arreglo es de una dimensión, de *ArrayElement2D* si fuera de 2 dimensiones, es posible tener arreglos de hasta 6 dimensiones.

```
class Arreglo : public Cbase_Arreglo {
public:
    Arreglo(parametros1);
    Arreglo(CkMigrateMessage *);
    void OtraEntrada(parametros2);
};
```

El constructor para la migración de Arreglo, regularmente debe estar vacía:

```
// En el archivo .ci
Arreglo::Arreglo(void)
{
    //... código para el constructor
}
Arreglo::Arreglo(CkMigrateMessage *m){}
```

### 4.7.5.2 Crear un Arreglo

Se puede crear un arreglo usando la rutina `CProxy_Array::ckNew`, esto devuelve un objeto Proxy, el cual se puede usar, copiar o enviar dentro de un mensaje. Un ejemplo para crear un arreglo de una dimensión que contenga índices (0,1,..., númeroElementos-1) se realiza de la siguiente manera:

```
CProxy_Arreglo1 = Cproxy_Arreglo1::ckNew(parametros, numElementos);
```

### 4.7.5.3 Mensajes en arreglos

El Proxy de un arreglo responde a una llamada siempre que el índice sea el apropiado, por ejemplo para arreglos de una dimensión usaremos índices como  $[i]$  o  $(i)$  para dos dimensiones serán de la forma  $(x, y)$ , para tres dimensiones usaremos  $(x, y, z)$  y de la misma manera hasta llegar a las seis dimensiones, en el caso de tipos definidos por el usuario se usa el índice como  $[f]$  o  $(f)$  donde  $f$  es un arreglo.

```
a1[i].hacerAlgo(parametros);
a3(x, y, z).hacerAlgo(parametros);
aF[ckArrayIndexFoo(. . .)].hacerAlgo(parametros);
```

Se puede invocar métodos sobre los elementos de un arreglo que no han sido creados aún, el sistema los mantendrá en un buffer el mensaje hasta que el elemento sea creado, ahora, no existe un orden predefinido para la entrega de los mensajes, por ejemplo, si invocamos al método  $A$  y después al método  $B$ , existe la posibilidad que se ejecute  $B$  antes que  $A$ . Se invocan los métodos o funciones de los arreglos como en las funciones de C++:

```
a1[i].A();
a1[i].B();
```

Los mensajes enviados para migrar elementos serán entregados después de que los elementos lleguen a su destino. Esto puede causar un error al enviar un mensaje a un elemento del arreglo que ha sido migrado, ya que es eliminado en el procesador origen.

### 4.7.5.4 Broadcast

Para transmitir un mensaje a todos los elementos concurrentes de un arreglo, solamente hay que omitir el índice, al llamar a algún método o función del mismo:

```
// invoca a hacerAlgo sobre cada elemento del un arreglo de chares
a1.hacerAlgo(parametros);
```

El mensaje será entregado a cada elemento del arreglo solamente una vez. La transmisión trabaja correctamente aún si hay migración, inserciones y borrados de elementos, debido a que los índices guardan la ubicación para el sistema de donde están los elementos del arreglo.

#### 4.7.5.5 Reducciones sobre arreglos

Una reducción es una operación que recoge los resultados de una operación como suma o resta de varios elementos en un grupo de procesadores en uno solo. Los datos a reducir vienen de cada elemento del arreglo, los cuales hacen el llamado al método “contribute”.

```
ArrayElement::contribute(int nBytes, const void *datos,
CkReduction::reducerType type);
```

#### 4.7.5.6 Borrado de arreglos

Para eliminar un elemento de un arreglo, se llama al destructor y se libera la memoria, la instrucción es la siguiente:

```
a1[i].ckDestroy();
```

Se debe tener la seguridad que ningún mensaje sea enviado al elemento borrado, y puede insertar un nuevo elemento en ese índice.

#### 4.7.5.7 Declarando índices de arreglos multidimensionales o definidos por el usuario

Charm++ tiene la capacidad de utilizar índices para arreglos multidimensionales y definidos por el usuario. Estos arreglos son declarados en el archivo interfaz (.ci) de la siguiente manera:

```
array [1D] A1 {entry A1(); entry void e(parametros);}
array [2D] A2 {entry A2(); entry void e(parametros);}
array [3D] A3 {entry A3(); entry void e(parametros);}
array [4D] A4 {entry A4(); entry void e(parametros);}
array [5D] A5 {entry A5(); entry void e(parametros);}
array [6D] A6 {entry A6(); entry void e(parametros);}
array [foo] AF {entry AF(); entry void e(parametros);}
```

La última declaración es el índice de un arreglo de tipo *CkArrayIndexFoo* que veremos en la siguiente sección como crearlo, el cual debe ser declarado en el archivo con extensión .decl.h. Ahora se declara en el archivo de cabeceras (.h) de la siguiente forma:

```
class A1 : public CBase_A1 { public: A1(){} ...};
class A2 : public CBase_A2 { public: A2(){} ...};
class A3 : public CBase_A3 { public: A3(){} ...};
```

```

class A4 : public CBase_A4 { public: A4(){} ...};
class A5 : public CBase_A5 { public: A5(){} ...};
class A6 : public CBase_A6 { public: A6(){} ...};
class AF : public CBase_AF { public: AF(){} ...};

```

Se puede acceder a un elemento de un arreglo 1D mediante el método heredado *thisIndex*, para un arreglo 2D con *thisIndex.x* y *thisIndex.y*, para uno 3D con *thisIndex.x*, *thisIndex.y* y *thisIndex.z*. Para el caso de los arreglos 4D, 5D y 6D se utilizan los campos {w,x,y,z}, {v,w,x,y,z} y {x1,y1,z1,x2,y2,z2} respectivamente.

Ahora basándonos en las declaraciones anteriores creamos arreglos de la siguiente forma:

```

CProxy_A1 arreglo1 = CProxy_A1::ckNew(parametros, elementos);
CProxy_A2 arreglo2 = CProxy_A2::ckNew(parametros, reng, colum);
CProxy_A3 arreglo3 = CProxy_A3::ckNew(parametros, reng, colum, prof);

```

#### 4.7.5.8 Índice definido por usuario

Para definir un nuevo índice, se crea una clase que herede las funciones y métodos de *CkArrayIndex* y se define la variable “*nInts*” para la longitud en entero para los índices del arreglo. Por ejemplo, si tenemos una estructura o una clase “*Foo*” el índice del arreglo lo podemos definir de la siguiente forma:

```

#include <charm++.h>
class CkArrayIndexFoo:public CkArrayIndex {
Foo f;
public:
    CkArrayIndexFoo(const Foo &in) {
        f=in;
        nInts=sizeof(f)/sizeof(int);
    }
    operator Foo &() { return f; }
    operator const Foo &() const { return f; }
};

```

El tamaño de *Foo* debe ser un número entero, también no debe contener punteros si no funciones virtuales. Finalmente, existe una configuración de tiempo en Charm++ llamada *CK\_ARRAYINDEX\_MAXLEN*, el cual es el número máximo de enteros que acepta un índice de arreglos, por defecto es 3; pero se puede sobrescribir usando la instrucción *DCK\_ARRAYINDEX\_MAXLEN=n*.

Para declarar un arreglo indexado de *Foo* se realiza lo siguiente:

```

//en el archivo interfaz (.ci):
array [Foo] AF { entry AF(); ... }

//en el archivo cabecera (.h):

```

---

```

class AF : public CBase_AF
{ public: AF() {} ... }

//en el código fuente:
    Foo f;
    CProxy_AF a=CProxy_AF::ckNew();
    a[CkArrayIndexFoo(f)].insert();
    ...

```

El constructor *CkArrayIndexFoo* al no estar declarado explícitamente, la última línea también puede escribirse:

```
a[f].insert();
```

Cuando el arreglo, hereda las funciones de *CBase\_ClassName* como se muestra a continuación, también cada elemento hereda sus funciones de *ArrayElementT*. Así el índice de arreglos es accesible mediante la función “*thisIndex*”. Por ejemplo:

```

//en el código fuente tenemos
AF::AF()
{
    Foo myF=thisIndex;
    functionTakingFoo(myF);
}

```

#### 4.7.5.9 Inserción Dinámica de Elementos

Charm++ tiene la capacidad de crear elementos nuevos sobre arreglos ya creados durante la ejecución de la aplicación. Para insertar nuevos elementos se llama al *insert*, opcionalmente puede tener algunos parámetros los cuales pasan al constructor, y el número de procesador donde será creado. Los elementos de un arreglo pueden insertarse en cualquier orden y en cualquier momento. después de usar *insert* debe llamarse al método *Cproxy\_Array::doneInserting*.

Ahora presentamos cómo inserta elementos en arreglos 1D, 2D, 3D y en los definidos por el usuario:

```

//En el código fuente usamos:
    int x,y,z;
    //Crea un nuevo arreglo 1D vacío
    CProxy_A1 a1=CProxy_A1::ckNew();

    for (x=...) {
        //usando corchetes
        a1[x].insert(parametros);
        // o usando paréntesis que es equivalente
        a1(x+1).insert(parametros);
    }
    a1.doneInserting();

//Crea un arreglo 2D

```

```

CProxy_A2 a2=CProxy_A2::ckNew();

for (x=...) for (y=...)
    /Aquí solo utilizamos paréntesis
    a2(x,y).insert(parameters);
    a2.doneInserting();

//Creamos un arreglo 3D
CProxy_A3 a3=CProxy_A3::ckNew();
for (x=...) for (y=...) for (z=...)
    a3(x,y,z).insert(parameters);
a3.doneInserting();

//Creando una arreglo con índices definidos
CProxy_AF aF=CProxy_AF::ckNew();
for (...) {
    // Podemos usar corchetes
    aF[CkArrayIndexFoo(...)].insert(parameters);

    // o paréntesis
    aF(CkArrayIndexFoo(...)).insert(parameters);
}
aF.doneInserting();

```

#### 4.7.6 Grupos de chares

Un grupo es una colección de *chares* de tal forma que existe un elemento *chare* sobre cada procesador (llamado en versiones anteriores “*branch*”, una rama o réplica). Cada réplica tiene sus propios métodos y datos. Los grupos tienen una sintaxis similar a un objeto *chare* simple, y heredan funciones de la clase definida *CBase\_ClassName*. Los grupos son declarados en el archivo interfaz con la siguiente sintaxis:

```

group GroupType {
    // especificaciones de interfaz como una chare
};

```

en el archivo cabecera (.h) definimos *GroupType* como:

```

class GroupType : public Cbase_GroupType {

    //Se colocan los datos y funciones como en C++
    // métodos de entrada como en una chare
};

```

Un grupo tiene un identificador global, que es del tipo *CkGroupID*, éste identificador es común para todos los elementos del grupo y se obtiene con la variable “*thisgroup*”. Para identificar un elemento del grupo se utiliza la variable “*thishandle*” como en una *chare* simple. Así los grupos pueden implementar operaciones paralelas más fácilmente, ya que además de poder enviar mensajes a una réplica o elemento en particular de un grupo, también

es posible transmitir un mensaje único a todas las réplicas de un grupo. Es posible la existencia de varias instancias que correspondan a un tipo de grupo y cada instancia tiene diferentes identificadores de grupo, además de su propio conjunto de réplicas.

#### 4.7.6.1 Creación de un grupo

Para la creación de un grupo primero hay que declararlo en el archivo interfaz de la siguiente manera:

```
group G{
    entry G(parametros1);
    entry void algunaEntrada(parametros2);
};
```

y en el archivo .h como:

```
class G : public CBase_G {
public:
    G(parametros1);
    void algunaEntrada(parametros2);
};
```

después ya podemos crear un grupo, es similar a la creación de una *chare* normal

```
CProxy_G groupProxy = Cproxy_G::ckNew(parametros1);
```

otra sintaxis también válida es la siguiente:

```
CkGroupID groupId = CProxy_G::ckNew(parametros1);
CProxy_G groupProxy(groupId);
```

#### 4.7.6.2 Invocación de métodos y funciones de los Grupos

Antes de enviar un mensaje a un grupo a través de un método de entrada, necesitamos obtener el Proxy para ese grupo (el identificador del grupo) . Obteniendo el Proxy del grupo un mensaje puede ser enviado a una réplica en particular de un grupo usando la siguiente sintaxis:

```
groupProxy[ProcNum].EntryMethod(parametros);
```

Esto llama al método de entrada *EntryMethod*, el cual deberá ser válido en el grupo, con sus parámetros a una réplica del grupo que es identificada mediante *groupProxy*, el cual se encuentra en el procesador número *ProcNum*. Esta llamada es asíncrona y no bloqueante, así que regresa inmediatamente a su ejecución después de enviar el mensaje.

Un único mensaje puede transmitirse a todas las réplicas de un grupo (i.e. a todos los procesadores) con la misma sentencia anterior, quitando el parámetro [*ProcNum*], para tener la invocación de la siguiente manera:

```
groupProxy.EntryMethod(parametros);
```

Los objetos secuenciales, *chare* y otros grupos de *chare* pueden tener acceso a grupo local usando:

```
GroupType *g=groupProxy.ckLocalBranch();
```

Esta llamada regresa un puntero al objeto actual (no al Proxy) a la cual hace referencia **groupProxy**. Una vez que un Proxy de un grupo local es obtenido, se puede acceder a las funciones y métodos de esa réplica como si se tratara de un objeto de C++. Sus métodos públicos pueden regresar valores, y sus datos públicos pueden ser leídos.

Así una *chare* creada dinámicamente puede llamar a un método público de un grupo cualquiera sin la necesidad de conocer en cual procesador esta actualmente, y el método se ejecutará en una réplica local.

Una función bastante útil de los Grupos es reducir el número de mensajes enviados entre procesadores para la recolección de datos de todas las *chares* sobre un solo procesador antes de enviar los datos a la “*mainchare*”. Para hacer esto, se crea una *chare* que divida un problema en partes, después se crea el grupo que se va a encargar de procesar todas las partes. Cuando una *chare* en particular ha terminado su trabajo, éste reporta a la réplica local, después cuando todas las *chares* de un procesador han terminado su trabajo la réplica local reporta a la “*mainchare*”. Con esto reducimos el número de mensajes entre *chares* en los procesadores y la *chare* principal con lo cual no se genera un cuello de botella al finalizar todas las operaciones.

#### 4.7.7 Objetos Grupos Nodo (NodeGroup)

Los grupos de nodos son muy similares a los objetos grupo, los grupos nodo tienen una réplica sobre cada nodo y además una réplica en cada procesador, es decir si una máquina tiene más de un procesador habrá una réplica para todos los procesadores que tenga la misma y una réplica para cada uno de los procesadores que este tenga. Cuando un método de entrada de un grupo nodo es ejecutado, este corre solamente sobre el procesador dentro de ese nodo.

La sintaxis de los grupos nodo es similar a los grupos. Además hereda las funciones y métodos de la clase *NodeGroup* de la misma manera que los grupos heredan de la clase *Group*. Ahora se muestra la sintaxis para crear grupos nodo, primero va la declaración en el archivo interfaz:

```
nodegroup NodeGroupType{
    // Interfaces de un chare
};
```

después en el archivo cabecera (.h), definimos a un *NodeGroupType* de la siguiente forma:

```
class NodeGrouptype : public NodeGroup [, otras superclases]{
```

```

//Funciones y datos de la clase como en C++
// métodos de entrada como de una chare normal
};

```

Al igual que los grupos, los nodos tienen un identificador único y global de tipo *CKGroupID* y de igual manera que en los grupos, este es el mismo para todas las réplicas. El identificador para los grupos nodo es obtenido mediante la variable “*thisgroup*”, nuevamente como en los grupos para una réplica en particular utilizamos la variable “*thishandle*”.

Los grupos nodo poseen métodos exclusivos, esto significa que, un método de este tipo no se ejecutará mientras otro método también exclusivo este ejecutándose sobre el mismo nodo.

#### 4.7.7.1 Invocación de métodos en un Grupo Nodo

Uno o varios métodos o funciones pueden ser invocados sobre una réplica en particular de un grupo nodo especificando el número de nodo como un parámetro. Si este parámetro es omitido se manejará como una invocación general, de tal manera que se ejecutará el método o la función invocada sobre todos los nodos. Cuando un método es invocado en una réplica en particular, este puede ser ejecutado por cualquier procesador de ese mismo nodo. Así dos invocaciones de un método en específico en un nodo pueden ser ejecutadas al mismo tiempo en dos procesadores distintos de un mismo nodo. Si ese método contiene código que debe ser ejecutado por un solo procesador, deberá ser marcado con una bandera de exclusión: *exclusive* en el archivo de interfaz (.ci). Por ejemplo si el método *M* de un nodo *NG* es marcado como *exclusive*, entonces mientras *M* esté en ejecución ningún otro método exclusivo puede ser ejecutado en el mismo nodo y sus procesadores, pero sin embargo si es posible ejecutar métodos *no exclusivos* en los demás procesadores del mismo nodo.

Se puede acceder a la réplica local de un grupo nodo usando la función *CkLocalNodeBranch()*. Así los datos pueden ser leídos y/o modificados, o bien puede ejecutar sus métodos o funciones sobre una réplica de un nodo usando esta función. La función *CkLocalNodeBranch()* regresa un puntero genérico (*void \**), similar a *CkLocalBranch*, si se hace el llamado sobre una clase Proxy correcta se podrá obtener también el tipo de puntero correcto. Ahora tales accesos son no seguros por defecto, de tal forma que la invocación concurrente de un método en un grupo nodo por procesadores diferentes dentro de un nodo puede tener resultados imprevisibles. Una forma de evitar esto es usar candados que utiliza *Converse* que es otra herramienta separada de Charm++ pero no hablaremos de ella.

#### 4.7.8 Balance de carga.

Charm++ tiene soporte para el balance de carga para un gran número de tareas *chare* o para elementos de un arreglo de *chares* que estén disponibles para ser migrados a otro procesador existente.

Varias aplicaciones paralelas, especialmente las simulaciones son iterativas por naturaleza, estas pueden contener una serie de pasos o iteraciones que convergen a una solución. Para este tipo de cómputos se utiliza típicamente la heurística llamada “*principio de persistencia*” en el cual la carga computacional y los patrones de comunicación entre objetos tienden a ser persistentes o previsibles a través del tiempo incluso en aplicaciones dinámicas. En esos casos se utiliza una entidad que prediga el comportamiento en un futuro cercano de estos eventos.

Para aplicaciones tales que no tengan una estructura iterativa o que dicha estructura en no tenga el principio de persistencia no aplica (es decir no sea previsible su comportamiento), Charm++ tiene un administrador de semillas, el cual mueve las semillas de nuevas *chare* entre procesadores para lograr así un balance en la carga entre ellos. Esta estrategia esta disponible para una *chare* simple o para arreglos.

#### 4.7.8.1 Estrategia de migración de objetos basado en la medición

En Charm++ los objetos pueden migrar de un procesador a otro en tiempo de ejecución. La migración de los objetos mejora el rendimiento de programa paralelo al migrar objetos de un procesador sobre cargado a otro que no lo esté.

Charm++ implementa una estrategia balance de carga basado en la medición del espacio de trabajo al cual automáticamente se recaban la carga computacional y la estructura de comunicación, guardándolo en una base de datos de balance de carga, obteniendo así varias estrategias de balance de carga con el trabajo de mapear los objetos a los procesadores. Esta estrategia es eficiente cuando podemos asumir cierta relación entre el cómputo y los patrones de comunicación, es decir, se puede predecir parte del cómputo en base a la base que se guarda.

#### 4.7.8.2 Estrategia de Balance de carga disponible

El balance de la carga puede ser mejorado de un modo centralizado o distribuido. En un enfoque centralizado, la carga completa de la máquina y la estructura de comunicación son acumuladas un solo punto, regularmente el procesador 0, y después determinar una nueva distribución de los objetos. La carga centralizada requiere sincronización el cual puede incurrir en una sobrecarga y un retraso. Sin embargo debido al alto grado de conocimiento de la máquina las decisiones tienden a ser más exactas.

Desde el punto de vista distribuido, la carga se intercambia entre procesadores vecinos, no requiere sincronización global. Sin embargo no provee de un balance de carga inmediato, se realizan varias iteraciones hasta realizar el balance completo.

A continuación se nombran los balanceadores de carga centralizados y una descripción:

1. **RefineLB:** Mueve los objetos de un procesador sobrecargado para obtener mejor rendimiento, se tiene un límite de objetos migrados.

2. **RefineCommLB:** igual que el anterior pero tomando en cuenta la comunicación.
3. **RandCentLB:** Asignación de objetos de manera aleatoria.
4. **RecBistectBtLB:** Partición recursiva con enumeración de partes
5. **MetisLB:** usa Metis(tm) para dividir la gráfica de comunicación de objetos.
6. **GreedyLB:** Usa el Algoritmo voraz tomando el objeto más pesado y lo envía al procesador menos cargado.
7. **GreedyCommLB:** Igual que GreedyLB pero tomando en cuenta la comunicación.
8. **ComboCentLB:** Un balanceador de carga especial que puede ser usado para combinar varias opciones de las antes mencionadas al mismo tiempo.

Para los balanceadores de carga distribuidos:

- **NeighborLB:** En una vecindad de balanceadores para mejorar el rendimiento se pueden cambiar trabajo solamente entre vecinos.
- **WSLB:** Un balanceador de carga de un cluster puede detectar cambios de carga del procesador y puede ajustar la carga de trabajo sin interferir con la ejecución de los procesadores.

#### 4.7.8.3 Balance de carga en arreglos chare

El entorno de balance de carga esta integrado dentro de una implementación del arreglo *chare*, cuando un arreglo es creado automáticamente se registran sus elementos para el balance de carga, el tiempo de cómputo y los patrones de comunicación. Para poder usar el balance de carga se tendrá que hacer el arreglo migrable y elegir una estrategia de balance de carga.

Existen 3 métodos para la usar el balance de carga con arreglos:

- **Modo de Balance de carga periódico.** En este método se debe especificar que cada que tiempo se realiza el balance de carga, este puede ser en cualquier momento excepto a la mitad de un método de entrada.
- **Modo automático con sincronización:** El usuario indica específicamente cuanto realizar el balance de carga dentro del código fuente. Usando *AtSync()* los elementos a procesar pueden ser migrados en la ejecución del programa.
- **Modo manual.** El balanceador de carga puede ser programado para iniciar manualmente. Para cambiar a modo manual se debe hacer el llamado a la función *TurnManualLBOn()* en cada procesador para prevenir que inicie automáticamente y

este debe ser llamado en el programa tan pronto como inicie. Y la función *StartLB()* inicia el balanceador de carga y esté solo se debe llama una vez y en solo un procesador.

Otras funciones útiles para el balance de carga son los siguientes:

- **LBTurnInstrumentOn()** y **LBTurnInstrumentOff()** Son funciones para iniciar o detener la recolección de los datos, estadísticas e instrumentar el balance de la carga.
- **setMigratable(CmiBool migratable)** es una función para los arreglos y funciona para indicar al balanceador de carga que un elemento es migrable o no.
- **Lbdb->SetLBPeriod(double s)** esta función se utiliza para regular cada cuanto tiempo se debe hacer un balance de la carga y puede ser utilizado en cualquier momento de la ejecución.

```
// si utilizamos dentro un arreglo se llama de esta manera
LBDatabase *ldb = getLBDB();
ldb->SetLBPeriod(5);
```

```
// y fuera de un arreglo debemos llamar después de que sea
inicializado el balanceador de carga
```

```
LBDatabase *ldb = LBDatabase::Object();
ldb->SetLBPeriod(5);
```

Se puede como forma alternativa *+LBPeriod {segundos}* en línea de comandos.

#### 4.7.8.4 Opciones para usar el módulo de balance de carga en el compilador y en tiempo de ejecución

Las estrategias de balance de carga están implementadas en una librería de Charm++ esto permite al programador fácilmente experimentar con las mismas simplemente al ligar o al ejecutar agregándolos vía la línea de comandos.

Opciones al tiempo de ligar.

- *-module NeighborLB – module GreedyCommLB ...*  
De esta manera se cargan los módulos para utilizar las estrategias NeighborLB y GreedyCommLB pero quedan inactivas hasta que en tiempo de ejecución es llamado.
- *-module EveryLB*  
Con esta orden se cargan los módulos de las estrategias de balance de carga
- *-balancer GreedyCommLB*

Con esta orden se cargan los módulos y se activan para ser invocado en tiempo de ejecución, también se pueden utilizar varias estrategias por ejemplo:

- *-balancer GreedyCommLB -balancer RefineLB*  
Con lo cual en primera instancia para el balance se usa *GreedyCommLB*, en segunda se utiliza *RefineLB* al siguiente nuevamente *RefineLB* y así sucesivamente.
- *-balancer ComboCentLB:GreedyLB,RefineLB*  
De esta manera se puede elegir usar un combo con lo cual cada paso de balance de acuerdo al ejemplo se aplica primero: *GreedyLB* e inmediatamente *RefineLB*.

Opciones para el momento de ejecutar:

- *+balance helper*  
Visualiza todos los módulos que se cargan al momento de ligar.
- *+balancer GreedyCommLB*  
Se activan múltiples balanceadores de carga en tiempo de ejecución para trabajar en secuencia.
- *+balancer ComboCentLB:GreedyLB,RefineLB*  
Lo mismo que en tiempo de compilación pero para trabajar en la ejecución del mismo

La opción *+balancer* funcionará solamente si se activó el módulo en tiempo de compilación, si se utilizó la opción *-balancer "A"* para compilar no será necesario utilizar *+balancer "A"* al momento de ejecutar, pero se puede usar *+balancer "B"* para sobrescribir las opciones y activar el módulo "B" en lugar del "A".

#### 4.7.8.5 Otras opciones útiles

Existen varias opciones más que pueden llegar a ser útiles dependiendo del tipo de problema que deseemos resolver:

- *+LBDebug{nivel de detalle}*  
{nivel de detalle} puede ser cualquier número entero positivo, 0 esta desactivado, mientras más alto el número más detallada será la salida.
- *+LBPeriod{segundos}*  
{segundos} puede ser cualquier número flotante, sirve para que se defina cada cuanto se debe balancear o equilibrar la carga, por defecto esta en 1.
- *+LBSameCpus*

Esta opción le indica al equilibrador de carga que todos los procesadores tiene la misma velocidad y así evitar que en tiempo de ejecución éste tenga que hacer esa medición.

- *+LBObjOnly*  
Esta orden le indica al equilibrador de carga que ignore la carga del procesador cuando decide migrar objetos
- *+LBSyncResume*  
Regularmente un procesador empieza a realizar los cálculos una vez que han recibido todos los objetos, no importando que otro procesador aún este migrando objetos, en ciertos casos resulta un problema, así que con esta orden se hace un barrido completo para asegurar que empiecen los cálculos una vez que se tienen migrados todos los objetos.
- *+LBOff* y *+LBCommOff*  
*+LBOff* deja desactivada la instrumentación para el balance de carga sobre todos los datos, uso del procesador y de comunicación, *+LBCommOff* deja desactivada el monitoreo para la comunicación, pero para el procesador aún continua activo.

#### 4.7.9 Quiescence

Quiescence del inglés inactividad, esta definido como el estado donde el procesador no esta atendiendo algún punto de entrada (*entry point*) y cuando no esta a la espera de algún mensaje para procesar.

Charm++ provee de 2 formas para detectar la inactividad:

**CKStartQD**: registra en el sistema una llamada de respuesta que puede ser hecha a la próxima vez que se detecte inactividad.

**CKWaitQD** tiene 2 variantes las cuales llevan los siguientes argumentos:

Un índice correspondiente a la función de entrada que será llamada, y un handle para la tarea (*chare*) sobre el cual será llamada la función de entrada. La sintaxis es como sigue:

```
CKStartQD(int Index, const CKChareID* chare ID);
```

Para recuperar el índice correspondiente a un punto de entrada particular, se deberá obtener un índice estático dentro del objeto *CKIndex* usando la siguiente sintaxis:

```
myIdx = CKIndex_ChareName::EntryMethod(parametros);
```

donde *ChareName* es el nombre de la tarea que contiene el método de entrada deseado, *EntryMethod* es el nombre del método, y los parámetros son utilizados para encontrar el método apropiado, sino son ignorados.

Un objeto *CkCallback*. La sintaxis para hacer el llamado es como sigue:

```
CkStartQD(const CkCallback& cb);
```

Una vez localizada la falta de actividad es regresada la llamada sin parámetros.

*CkWaitQD*, por el contrario, no envía de regreso una llamada. Es más, *CkWaitQD* bloquea y no regresa hasta no encontrar una inactividad. Su sintaxis es la siguiente:

```
CkWaitQD();
```

Hay que tener en cuenta que *CkWaitQD* solo puede ser llamado desde un método de entrada *threaded* debido a que suspende la ejecución del thread actual, si es llamado fuera de un método *threaded* este puede suspender el thread principal en el procesador donde fue llamada *CkWaitQD*, lo cual puede suspender lo que se realiza en ese procesador.

## 4.7.10 Reducciones

Esta es una útil herramienta paralela para los casos donde se tiene que combinar varios resultados procedentes de cada uno de los procesadores disponibles para los cálculos y es necesario recolectarlos en un solo procesador.

Las reducciones transforman los datos aislados a través de un arreglo en un solo valor, como una suma o encontrar un máximo. Cada una de las tareas en un arreglo contribuye con su resultado local, de tal forma que si obtenemos un error local, al combinarlo en un entorno global este se transformará en un error global.

### 4.7.10.1 ¿Por qué usar reducciones?

Con un simple paso de mensaje será suficiente para recoger valores de cada tarea *chare* y combinarlos. Se puede escribir un programa donde cada *chare* envíe su valor local a la “*chare 0*”, el cual examina los valores y produce el resultado deseado. Así que, ¿Por qué utilizar reducciones?

La primera razón es la eficiencia. Si cada *chare* en el arreglo envía su valor directamente al “*chare 0*”, entonces este tendrá una cantidad grande de mensajes para el procesador donde resida y los otros procesadores que no tienen algo que procesar, quedarán ociosos. Por lo tanto “*chare 0*” se convierte en un cuello de botella para el procesador donde reside. Un mejor enfoque es construir un árbol con procesadores, donde cada *chare* toma algunos mensajes de los “*chares hijos*”, los combina, y pasa el resultado a un próximo nivel. Esto elimina el cuello de botella en el “*chare 0*” y mejora el rendimiento, pero es más complicado implementarlo. Si se usa una reducción, esta complejidad queda oculta por el sistema en tiempo de ejecución, pero se obtiene un programa más eficiente.

La segunda razón en usar reducciones es la claridad. Hay muchas operaciones de reducción que son muy comunes en programas paralelos: encontrar los mínimos globales, construir una lista en la que cada elemento viene de una *chare* diferente, evaluar si una condición lógica es verdadera sobre cualquier *chare* y más tareas similares. Charm++ tiene varios tipos de reducciones que realizan éstas y muchas otras operaciones. ¿Cuál es la ventaja? Alguien que lea su código podrá ver inmediatamente que se está usando la operación `CkReduction::max_int`, con lo cual puede deducir que esta buscando el entero máximo sobre un arreglo de *chares*. Si se hiciera el cálculo a mano, tendrá que revisar más cuidadosa y en algunos casos completamente para determinar qué es lo que hace el código.

#### 4.7.10.2 ¿Cómo usar reducciones?

Para poder usar reducciones, necesitamos dos cosas: una de qué manera las *chares* van a enviar los datos para la reducción, y una forma de manipular esos datos cuando han sido reducidos. Afortunadamente, todas las tareas *chare* pueden utilizar una función llamada *contribute* que es usada para llevar los datos locales a una reducción general. La sintaxis de la función es la siguiente:

```
void contribute(int nBytes, const void *data, CkReduction::reducerType type);
```

En esta función el primer argumento es un entero que especifica el tamaño en bytes de los datos que se están aportando a la reducción, después tenemos un puntero a los datos, y por último un objeto especial de tipo `CkReduction::reducerType` que especifica qué clase de reducción se está llevado a cabo (por ejemplo. una suma de enteros), más adelante daremos a conocer una tabla con los tipos de reducción que existen. Por ejemplo, si está sumando una variable local que representa la fuerza sobre cada *chare*, la llamada de *contribute* sería:

```
// Primero obtenemos el dato
double force = get_force();

// después lo enviamos a la reducción
contribute(sizeof(double), &force, CkReduction::sum_double);
```

Ahora sabemos como conjuntar datos locales para una reducción, pero ahora lo que necesitamos es una forma de manejar la salida de la reducción. En Charm++, se maneja el resultado de una reducción usando un objeto *callback* conocido como un *cliente de reducción*. Esta función únicamente es llamada una vez que la reducción ha sido completada para tener acceso a su resultado.

Existen dos formas para especificar el cliente asociado con una reducción. Primero se puede establecer un arreglo de *chares* por defecto mediante la función *ckSetReductionClient*. Entonces si no hay cliente especificado cuando se llama a *contribute*, por defecto esta será utilizada.

```
myArray.setReductionClient(entrymethod, (void *) NULL);
```

La segunda forma es para especificar la función callback como un parámetro opcional para la función contribuye. Si pasa un objeto *CkCallback* como el último parámetro de la llamada a contribuye sobre cada *chare*, esa callback será usada, si diferentes *chares* especifican diferentes clientes, el comportamiento será indefinido.

Charm++ tiene un sistema de callback extenso, pero no necesitaremos todas las funcionalidades para realizar un callback simple. Solo construiremos un callback que, cuando sea invocado llamará a un método de entrada que nosotros especifiquemos. Para hacer esto creamos un objeto *CkCallback* con el método que será invocado y un “*proxy chare*” indicando la clase a la que pertenece. Por ejemplo, si tenemos un arreglo de tipo *myType*, el cual tiene un Proxy llamado *thisProxy* y una función *myReductionFunction* se puede construir un callback de la siguiente manera:

```
CkCallback* cb = new CkCallback(CkIndex_myType::myReductionFunction(NULL),thisProxy);
myArray.ckSetReductionClient(cb);
```

Hay una restricción importante en las funciones callback, deben aceptar un solo argumento de tipo *CkReductionMessage\**. Este mensaje contiene los datos generados por la reducción. Este tiene un método *getSize()*, el cual devuelve el tamaño de los datos en bytes y *getData()* el cual devuelve un puntero al dato.

#### 4.7.10.3 Tipos De Reducciones

Charm++ incluye varios tipos de reducción para usar directamente y puedan ser aplicadas fácilmente para realizar sumas, multiplicaciones, y otras operaciones simples. Los primeros 4 tipos utilizan tipos enteros (*int*), punto flotante (*float*) y dobles (*double*), Las reducciones lógicas solo trabajan con enteros.

- *CkReduction::sum\_int, sum\_float, sum\_double*: con este tipo podemos realizar sumas de números.
- *CkReduction::product\_int, product\_float, product\_double* : con este otro tipo obtenemos el producto de números.
- *CkReduction::max\_int, max\_float, max\_double*: este tipo de reducción es para encontrar el número más grande o mayor de una lista.
- *CkReduction::min\_int, min\_float, min\_double*: este otro tipo de reducción es para encontrar el número más pequeño o menor de una lista.
- *CkReduction::logical\_and*: este tipo de reducción realiza la operación lógica AND y obtenemos un 0 si es falso, cualquier otro en caso de ser verdadero.

- *CkReduction::logical\_or*: equivalente a la reducción anterior para la operación lógica OR.
- *CkReduction::bitvector\_and*: igual que la operación lógica *AND*, pero para un bitvector (una cadena de bits, es decir, una cadena de 0 y 1).
- *CkReduction::bitvector\_or*: equivalente al *bitvector\_and* para la operación OR.
- *CkReduction::set*: este tipo de reducción realizará una concatenación de los datos a reunir, estos irán separados en registros *CkReduction::setElement*. Los datos pueden ser de cualquier longitud.
- *CkReduction::concat*: Esta reducción realiza la concatenación bit a bit de los datos recolectados, en este caso no hay una separación de datos, como en *CkReduction::set*.
- *CkReduction::set* regresa un conjunto de objetos del tipo *CkReduction::setElement*, uno por cada dato reunido, la definición de su clase es siguiente:

```
class CkReduction::setElement {
public:
    int dataSize; // El tamaño del arreglo
    char data[]; // El dato que irá en el arreglo
    CkReduction::setElement *next(void);
};
```

Para obtener los datos de cada elemento del arreglo creado se puede utilizar la siguiente rutina:

```
//Despues de realizar la reducción en los datos
// enviamos la información al puntero cur
CkReduction::setElement *cur=(CkReduction::setElement *)data;

while (cur!=NULL) { // mientras no sea un elemento vacio
//usamos cur->dataSize y cur->data, tamaño y dato
// respectivamente, despues avanzamos al siguiente dato
    cur=cur->next();
}
```

cabe aclarar que una reducción no lleva un orden, de tal forma que va a tomar los datos como los tenga disponibles.

#### 4.7.11 Callbacks

Las Callbacks o devolución de llamadas son funciones utilizadas por el sistema de Ejecución Charm++, Frameworks, librerías y demás aplicaciones para notificar al código de

una aplicación que un evento ha ocurrido o finalizado. Por ejemplo una reducción notifica a una parte del código que se ha completado para que pueda utilizar el resultado. La idea básica es que una aplicación pueda especificar qué acción realizar cuando una llamada callback indica que un evento ha ocurrido o finalizado. Esta acción puede realizar una variedad de cosas incluyendo llamadas a otras funciones o incluso finalizar la misma aplicación que esté ejecutándose.

#### 4.7.11.1 Interfaz del Cliente

Existen varias formas de invocar un objeto `CkCallback` dependiendo de qué se necesite realizar después de haber llamado o invocado a este objeto y esto se realiza mediante un mensaje.

#### 4.7.11.2 Método de Entrada Callback

Para crear un callback que provoque que un método de entrada sea llamado sobre un objeto *chare* se utiliza la siguiente sintaxis:

```
CkCallback cb(int entryMethodIndex, CkChareID &id)
```

Cada método de entrada para una clase *chare* tiene asociado un índice a un método de entrada. El índice a este método es usado para identificar cual es el método de entrada para un objeto *chare* específico (indicado por el parámetro *id*) el cual deberá ejecutarse.

Para obtener el índice de cualquier método de entrada se utiliza la siguiente instrucción:

```
int myIndex = CkIndex_MyChareClass::myEntryMethod(parametros);
```

Donde *MyChareClass* es el nombre de la clase *chare* y *myEntryMethod* es el nombre del método de entrada. Los parámetros pasados a esta función no son utilizados cuando se recupera el índice del método, solo son usados cuando se identifica cuál es el método de entrada que está siendo invocado, en el caso de que varios métodos estén compartiendo el mismo nombre (es decir, sobrecarga). Por ejemplo, para obtener el índice para un método de entrada tenemos el siguiente prototipo:

```
void MyChareClass::myEntryMethod(int* x);
```

y lo podríamos usar de la siguiente manera:

```
int myEntryMethodIndex = CkIndex_MyChareClass::myEntryMethod((int*)NULL);
```

Habría que recordar que esto no está llamando a *myEntryMethod*, así que es correcto tener un puntero vacío o null. Solo el tipo de parámetros es importante cuando una función está sobrecargada para poder encontrar la diferencia de cuál vamos a usar.

A continuación presentamos una lista de las formas en que se pueden invocar los callbacks, todas son similares solamente cambia a qué se va a aplicar: una *chare*, arreglos o grupos:

```
1. CkCallback cb(CkCallbackFunc cFunc, void* param);
```

Esta forma es para realizar un callback que ejecute una función de C. Aquí el tipo el *CkCallbackFunc* representa una función de la siguiente forma:

```
2. void myCFunction(void* param, void* msg);
```

El parámetro puntero es usado cuando el objeto callback fue creado y será pasado como una función de C en forma de un parámetro *param*. El parámetro *msg* será un puntero al mensaje creado por cualquier entidad (p.e. una librería).

```
3. CkCallback cb(CkCallback::ignore);
```

Este callback no realiza alguna acción cuando es enviada o cuando un evento ha ocurrido.

```
4. CkCallback cb(CkCallback::ckExit);
```

Este objeto lo que realiza es finalizar la ejecución de la aplicación.

```
5. CkCallback(int ep, const CkChareID &id);
```

Cuando se invoca este callback se envía un mensaje a un método de entrada de una *chare* especificada por su *CkChareID*, aunque también puede utilizarse un proxy en lugar del ID de la siguiente manera:

```
CkCallback myCB(CkIndex_myChare::myEntry(NULL), myChareProxy);
```

```
6. CkCallback(int ep, const CkArrayID &id);
```

Realiza lo mismo que la función anterior, pero ahora se envía un mensaje sobre un arreglo de *chares*, y de igual manera también se pueden utilizar proxies.

```
7. CkCallback(int ep, const CkArrayIndex &idx, const CkArrayID &id);
```

Ahora esta forma es para enviar un mensaje a un elemento del arreglo, el callback anterior enviaba un mensaje a todos los elementos

```
8. CkCallback(int ep, const CkGroupID &id);
```

Ahora en lugar de un arreglo se orienta a un grupo de *chares*. Se envía un mensaje para ejecutar alguna función a todo el grupo.

```
9. CkCallback(int ep, int onPE, const CkGroupID &id);
```

Similar a los grupos, ésta función es para enviar un mensaje a un elemento del grupo.

### 4.7.12 Empaquetamiento y desempaquetamiento de datos PUP

El concepto de PUP es una forma genérica para transformar datos en un objeto y usar esa forma para cualquier tarea que requiera serializar datos. El sistema Charm++ puede usar este tipo de transformación para empaquetar un objeto dentro de un mensaje, y desempacar el mensaje dentro de un nuevo objeto en otro procesador. Las siglas *PUP* es la contracción de la frase *Pack and UnPack*, empaquetar y desempaquetar; y las usaremos para referirnos a esta acción en adelante.

El concepto de PUP es más fácil de usar que describir, supongamos que tenemos:

```
class paquete : public mySuperclass {
private:
    double a;
    int x;
    char y;
    unsigned long z;
    float arr[3];

public:

    ...otros métodos...

    //rutina empaqueta/desempaqueta: Se definen los campos
    // que contendrá o usara

    void pup(PUP::er &p) { mySuperclass::pup(p);
        p|a;
        p|x; p|y; p|z;
        PUParray(p,arr,3);
    }
};
```

Esta rutina define los campos que un método *paquete* de Charm++ tendrá para realizar el empaquetamiento y desempaquetamiento de datos. Esto permite a Charm++ enviar parámetros del tipo *paquete* a través de los procesadores, trasladar objetos *paquete* a través de otros tipos de arquitecturas, leer y escribir objetos *paquete* directamente en archivos, inspeccionar y modificar objetos *paquete*; o reiniciar cálculos que sean parte de *paquete*.

#### 4.7.12.1 El uso de PUP

El objetivo de la rutina PUP es guardar y restaurar los datos en y de un objeto. Como definimos anteriormente, se puede guardar y restaurar el contenido de una clase con la rutina “*pup*” la cual pasa todas las partes de la clase a un objeto del tipo *PUP::er*, el cual guarda o restaura los datos según sea requerido.

La rutina PUP para objetos complicados normalmente es usada solo para sus partes más simples. Debido a que los objetos dependen de su superclase inmediata, en la primer

línea de cada rutina *pup* debe llamarse a su superclase (de la misma rutina). Si una superclase no tiene una rutina “*pup*”, se deberá guardar los valores en la superclase misma.

#### 4.7.12.2 Operador PUP

La forma de usar “*pup*” para empaquetar cualquier objeto “*a*” es la siguiente “*p | a*”. En esta sintaxis el operador “|” se aplica al objeto “*p*” del tipo *PUP::er* y usa como variable a empaquetar a “*a*”.

La sintaxis “*p | a*” funciona siempre que:

- “*a*” sea un tipo de variable sencilla, como caracteres, enteros cortos, largos, flotantes o dobles. En este caso, “*p | a*” copia los datos dentro del objeto que podría equivaler a pasar una variable directamente a *PUP::er* usando *p(a)*.
- “*a*” sea cualquier objeto con una rutina “*pup*”. En este caso “*p | a*”, llama a la rutina *pup* del objeto “*a*” sobre “*p*”, que equivale a la sentencia *a.pup(p)*.
- Un puntero a objeto que pueda *empaquetar* o que pueda usarse en la rutina *pup*. En este caso “*p | a*” reserva espacio en memoria y copia la subclase que sea apropiada.
- Un objeto con una declaración *PUPbytes(miClase)* en la cabecera. En este caso “*p | a*” copia el objeto como bytes similar a la instrucción de C++: *memcpy*.
- Un objeto con un operador “|” definido por el usuario. En este caso “*p | a*”, llama al operador que se ha definido para “|”.

Para guardar los datos dentro de un objeto contenedor simplemente se deberá ejecutar “*pup*” para cada elemento a guardar. Para arreglos se puede utilizar la rutina “*PUParray*”, la cual toma los datos, un puntero base del arreglo y su longitud. Esta utilidad la podemos definir con el siguiente ejemplo, supongamos que usamos una clase del tipo “*T*” entonces haremos lo siguiente:

```
template<class T>
inline void PUParray(PUP::er &p, T *array, int length) {
    for (int i=0; i<length; i++) p|array[i];
}
```

Si un objeto no tiene una rutina PUP, no se puede agregar una o usar la rutina *PUPbytes*, se puede definir un operador “|” para guardar datos en el objeto. Por ejemplo, si la clase *myClass* contiene 2 campos *a* y *b*, podemos usar el operador “|” de la siguiente manera:

```
inline void operator|(PUP::er &p, myClass &c) {
    p|c.a;
```

```
    p|c.b;
}
```

#### 4.7.12.3 PUP como bytes

Para clases y estructuras con varios campos, puede llegar a ser tedioso y propensos a cometerse errores al listar todos los campos en la rutina *pup*. Esto puede evitarse de dos formas, tan grande como sea el objeto puede ser copiado como un bloque de *bytes puros* (*raw*), esto normalmente en el caso de ser estructuras y clases simples, que no incluyan punteros.

Una es con el macro *PUPbytes(miClase)* en el archivo cabecera. Esto permite usar la sintaxis *p|\*myPtr;* para empaquetar la clase entera con un tamaño de *sizeof(miClase)* de bytes puros.

Con *p((void \*) myPtr, sizeof(miClase));* en la rutina *pup*, esto es una llamada directa para que *pup* guarde un conjunto de bytes.

Y con *p((char \*) myCharArray, arraysize);* en la rutina *pup* realizamos lo mismo que lo anterior, pero para un arreglo.

Esta rutina hace una función similar a “*memcpy*”: copia los datos por completo y no hace cambios sobre ellos. Por ejemplo, si la clase contiene algún puntero, hay seguridad que no hará alguna asignación de memoria ni empaquetara ninguna referencia hacia si mismo.

Empaquetar como *bytes* previene que la rutina *pup* tenga que ser habilitada para arquitecturas diferentes cada vez que se utilice. Pero el empaquetamiento como *bytes* actualmente ya no se está usando.

#### 4.7.12.4 Sobre carga PUP

Para la sobrecarga de una clase PUP el cambio en código es muy pequeño, se usa una llamada a una función virtual para cada elemento o arreglo para empaquetar o desempaquetar. Esto hace una copia binaria de memoria a memoria.

Para arreglos del tipo “*int*” y “*double*”, o arreglos con la declaración “*PUPbytes*”, se debe usar *PUParray* ya que realiza transferencias de bloques de forma rápida, usando una función virtual por arreglo.

#### 4.7.12.5 Modos que se utilizan en la rutina PUP

Charm++ usa la rutina *pup* para empaquetar y desempacar, pasando por diferentes tipos de rutinas para ello, por ejemplo, la rutina *p.isUnpacking()* devuelve verdadero si el objeto en cuestión ya ha sido desempacado, esto es, que los objetos han sido restaurados de

manera correcta. La rutina *pup* debe realizar varios pasos: calcular el tamaño correcto de los datos, empaquetar y desempacar, y verificar que al salvar y restaurar los campos esten exactamente en el orden.

Existen 3 modos a usar, con 3 tipos diferentes, *sizing* que solo calcula el tamaño de los datos sin hacer modificación alguna de estos, *packing* el cual lee los datos y los guarda y por último *unpacking* que restaura de manera correcta la información empacada. Es posible conocer cuándo se ha utilizado alguno de los modos usando las rutinas *p.isSizing()*, *p.isPacking()* y *p.isUnpacking()*.

La bandera *p.isDeleting()* indica que el objeto utilizado será borrado después de usar la rutina “*pup*”. Aunque solo son necesarios cuando se realiza el programa en C y Fortran. En la interfaz C++ al finalizar todo el proceso, el objeto tiene su destructor implícito que elimina el objeto utilizado, por esta razón no es requerido.

#### 4.7.12.6 Ciclo de un objeto PUP

Como se muestra en la siguiente figura el ciclo de vida de un objeto con la rutina “*pup*”, como en C++, los objetos son creados, utilizados y después son destruidos. Los objetos pueden ser creados de 2 maneras, se pueden crear con el constructor del usuario o de la rutina *pup*, una vez creados responden a los métodos o funciones que se utilicen y los métodos de entradas remotos y al mismo tiempo se realiza la llamada a los modos “*isSizing*” e “*isPacking*”. Finalmente los objetos son destruidos llamando a su destructor

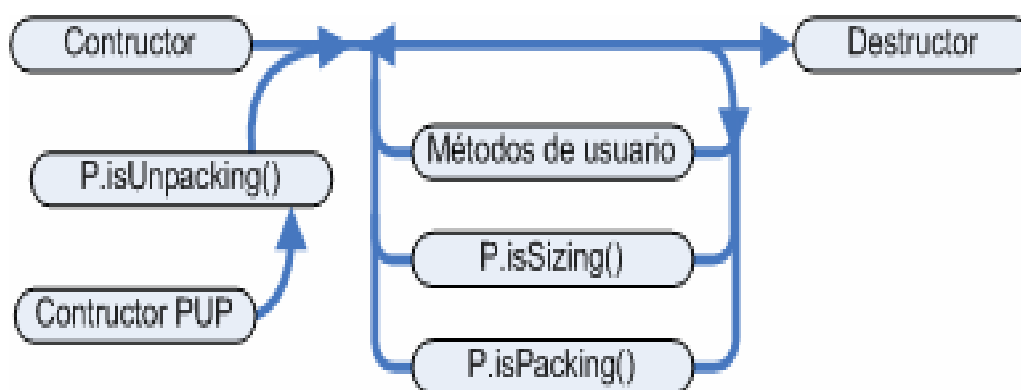


Fig. 4.4 Ciclo de vida de un objeto con la rutina *pup*.

#### 4.7.12.7 Estructuras y clases enviadas a través de la red definidas por el usuario

El sistema de envío usa un entorno *pup* para copiar los datos, esto quiere decir que cada clase que desee enviarse deberá usar un rutina *pup*, una declaración *PUPbytes* o un operador | trabajando.

```
// Declaración
class punto3D {
public:
    double x,y,z;
    void pup(PUP::er &p){
        p | x; p | y; p | z;
    }
};

typedef struct {
    int redCount;
    char data[17];
} refChars;
PUPbytes(fecha);

class fecha {
public:
    char mes, dia;
    int anio;
    // ... las rutinas para manipular los datos van aquí ...
}

inline void operato | (PUP::er &p, fecha, &d) {
    p | d.mes;
    p | d.dia;
    p | d.anio;
}
```

En alguna parte del archivo de interfaz (.ci) debe llevar

```
entry void puntoRefAFecha(punto3D &p, refChars r[d.anio], fecha &d);
```

Cualquier tipo definido por el usuario en la lista de argumentos debe ser declarada antes incluyendo el archivo “*.decl.h*”. Al igual que en *C++* es más eficiente pasar una estructura grande por referencia que por valor.

Por eficiencia, los arreglos (como *refChars*) son copiados por bloques de bytes y se envían mediante punteros. Esto significa que las clases necesitan su rutina *pup* para ser llamada, tal como sucede cuando se reserva espacio para un dato en memoria dinámicamente o con métodos virtuales no se puede pasar del uso de matrices como *CkVec* o vectores STL para pasar listas de clases complicadas definidas por el usuario. Por razones históricas, punteros que acceden a estructuras no pueden enviarse como parámetros, por que son confundidos con mensajes.

El orden para enviar datos es:

- Llamar a *p | a* sobre cada parámetro a enviar con un *sizing PUP::er*.
- Calcular la longitud de cada arreglo
- Llamar a *p | a* sobre cada parámetros a enviar con un *packing PUP::er*.

- Usar *memcpy* sobre cada dato del arreglo

El orden para recibir los datos enviados

- Crear una instancia para cada parámetro que fue enviado usando un constructor por defecto.
- Llamar a *p | a* sobre cada parámetro recibido usando *unpacking PUP::er*.
- Procesar los punteros dentro del mensaje para cada arreglo.

Finalmente, para estructuras demasiado grandes es más eficiente usar mensajes para enviar los datos, debido a que son construcciones a bajo nivel ya definidas dentro del Sistema de Charm++, donde se minimiza el copiado y la sobrecarga, aunque es posible que existan construcciones bastante complicadas donde los mensajes no sean lo más óptimo, si no pasar parámetros mediante el uso del empaquetamiento de variables o datos.

#### 4.7.13 Otros Comandos

Los siguientes comandos proveen información acerca de la computadora sobre la cual se ejecuta un programa paralelo. Donde:

1. *Pe* representa a un solo procesador (*Pes* en plural).
2. *node* representa a una computadora (*nodes* en plural).
3. *Pes* y *nodes* se enumeran comenzando en 0.

De tal forma que, si un programa paralelo se esta ejecutando sobre una máquina de cuatro procesadores y otra que solo tiene dos, deberá haber en total seis *Pes* (0, 1, 2, 3, 4 y 5) pero solo 2 *nodes* (0 y 1). Los *Pes* de un *node* deben ser enumerados de forma secuencial.

*int CkNumPes()*

Devuelve el número total de procesadores de todos los nodos

*int CkMyPe()*

Devuelve el número del procesador desde el cual se realizó la llamada a este comando

*int CkMyRank()*

Devuelve la clasificación del procesador desde el cuál se realizó la llamada a este comando. La clasificación inicia desde cero.

*int CkMyNode()*

Devuelve el número del nodo desde donde se realizó la llamada a este comando

*int CkNumNodes()*

Devuelve el número total de nodos.

***int CkNodeFirst(int node)***

Devuelve el número del primer procesador dentro de los nodos

***int CkNodeSize(int node)***

Devuelve el número de procesadores totales en el *node*.

***int CkNodeOff(int Pe)***

Devuelve el número del nodo al que pertenezca *Pe*.

***int CkRankOf(int Pe)***

Devuelve la clasificación del *Pe* dentro del nodo al que pertenezca.

Los siguientes comandos son los que más comúnmente se utilizan.

***void CkAbort(const char \*message)***

Hace que el programa termine enviando un mensaje de error. Esta rutina no devuelve algún valor.

***void CkExit()***

Este comando informa al kernel de Charm que los cálculos en todos los procesadores deben finalizarse. Una vez ejecutado este método ningún otro mensaje o método será tomado en algún otro procesador.

***void CkExitAfterQuiescence()***

Este comando informa al kernel de Charm que los cálculos en todos los procesadores deben finalizarse tan pronto la computadora se encuentre sin trabajo, después de que todos los métodos de entrada han finalizado.

***double CkCpuTimer()***

Devuelve el valor actual del temporizador del sistema en segundos . El temporizador del sistema indica cuando el programa inicia su ejecución, este mide el tiempo de los procesos (usuario y sistema)

***double CkWallTimer()***

Devuelve el tiempo transcurrido en segundos desde que el programa inicia el temporizador wall timer.

***double CkTimer()***

Este es un alias para cualquier CkWallTimer sobre una computadora dedicada (como la ASCII Red) o CkCpuTimer para máquinas con múltiples procesos por CPU (tales como clusters de estaciones de trabajo)

#### **4.7.13.1 Comandos de entrada y salida**

Al igual que en C y C++, Charm++ provee de métodos similares para la entrada y salida de información.

En lugar del *printf* y *scanf* de C, Charm++ tiene **CkPrintf** y **CKScanf**. Estas funciones tienen la misma interfaz que las de C solo con algunas diferencias.

```
int CkPrintf(format [, arg]*)
```

Esta función es usada para la salida a la pantalla. Es usada de manera similar a *printf* de C, sin embargo, **CkPrintf** tiene una propiedad especial que es más conveniente para la programación paralela. **CkPrintf** dirige las salidas de la terminal a *charmrun*, así una tarea del procesador 3 puede llamar a **CkPrintf**, esa llamada sale como un mensaje y se envía a la computadora “host” para ser desplegado en lugar de la terminal donde está el procesador. El mensaje es enviado de manera asíncrona, es decir una vez enviado, el procesador continúa con la tarea o tareas que realizaba incluso antes que el mensaje sea desplegado en la pantalla.

```
void CkError(format [, arg]*)
```

Similar a CkPrintf, pero es enviado cuando hay mensajes de error en “stderr” para las funciones de entrada y salida.

```
int CkScanf(format [, arg]*)
```

Esta función es utilizada para la entrada de información. Es similar a *scanf* en C. La llamada a **CkScanf**, a diferencia de **CkPrintf**, bloquea la ejecución del procesador desde el cual es llamado, y continúa solamente cuando ha recibido la entrada.

Para las funciones de C++: *cout* y *cerr*, Charm++ tiene sus equivalentes como: *ckout* y *ckerr*, que al igual que sucede con las de C tienen la misma interfase solo que se pueden emplear de modo paralelo.

## 4.8 Projections

Projections es una herramienta de análisis y monitoreo de rendimiento que nos ayuda a visualizar, analizar, y entender que está sucediendo en la ejecución paralela de un programa escrito en Charm++. Es una herramienta basada en el seguimiento post-mortem de la aplicación con ciertas características que permiten un control sobre la cantidad de información generada y una menor perturbación en la aplicación al momento del seguimiento.

El análisis de rendimiento con Projections se desarrolla en 3 fases:

- Instrumentación del código (Realizada automáticamente) y enlaces para la generación de los módulos de seguimiento.
- Genera los datos del seguimiento.
- Visualización de los datos e información generada, para analizar algún posible problema de rendimiento en la aplicación.

El Sistema de Ejecución de Charm++ para realizar el monitoreo de una aplicación es automático, esto significa que el usuario no necesita insertar código adicional para generar datos de seguimiento. Projections provee de una Interfaz de programación de aplicaciones (API) que permite colocar código en la aplicación para un monitoreo más específico. Los usuarios podrán registrar y rastrear eventos específicos en su aplicación a través de este API.

### 4.8.1 Rastreo Selectivo

Charm++ permite iniciar o cancelar el rastreo en un punto definido sobre un procesador local, colocando al momento de compilar *+traceoff* cancela al inicio el rastreo si durante un periodo no se desea monitorear. De igual manera el usuario puede iniciar y volver a cancelar el rastreo en un punto definido de la misma aplicación.

- *void traceBegin()* Habilita el monitoreo de eventos sobre un procesador local donde sea invocado.
- *void traceEnd()* Deshabilita el monitoreo de eventos sobre un procesador local donde sea invocado.

### 4.8.2 Eventos definidos por el usuario

Projections tiene la posibilidad de monitorear eventos específicos de un usuario que son visualizados en una tabla de línea de tiempo. Los pasos a seguir son los siguientes:

- Registrar un evento con un identificador específico o adquirir un identificador global único.
- Usar el identificador del evento para especificar un punto de rastreo en la parte del código donde se necesite monitorear los eventos.

y las funciones necesarias son las siguientes:

```
int traceRegisterUserEvent (char* EventDesc, int EventNum=-1)
```

Esta función registra un evento *EventNum* asociando a *EventDesc*. Si *EventNum* no está especificado, un identificador global único será obtenido en tiempo de ejecución y será el que devuelva la función. *EventDesc* debe ser una constante de tipo cadena (*string*), si contiene una subcadena "\*\*\*" entonces la herramienta de línea de tiempo de Projections dibujará el evento expandiendo a todas las líneas de todos los procesadores.

*EventNum* debe ser el número de procesadores disponibles. Existen 3 formas de llamar a la función:

- Una llamada a *traceRegisterUserEvent* sobre el procesador 0 en *main::main* sin especificar un número de evento, almacenar la respuesta, que será el número de evento, en una variable de solo lectura. Por ejemplo:

```
eventID = traceRegisterUserEvent ("Time Step Begin");
```

- Una llamada a *traceRegisterUserEvent* sobre todos los nodos sin especificar un número de evento. El valor devuelto es un número de evento común de acuerdo al sistema.
- Una llamada a *traceRegisterUserEvent* especificando el número de evento sobre el procesador 0. Haciendo esto sobre otro procesador no debe tener efecto. Más adelante, el número de evento puede ser usado en la siguiente llamada de un evento de usuario. Por ejemplo:

```
- traceRegisterUserEvent ("Time Step Begin", 10);

void tracerUserEvent (int EventNum)
```

Esta función trabaja similar a un marcador, indicando la ocurrencia de un evento en un punto definido por el usuario dado *EventNum*.

```
void tracerUserBracketEvent(int EventNum, double StartTime, double EndTime)
```

Esta función graba un intervalo de eventos o identifica actividad de *EventNum* en el último periodo entre *StartTime* y *EndTime*, estos 2 datos pueden ser obtenidos con la función *CmiWalltimer()* y se obtendrán en segundos.

### 4.8.3 Modos de Rastreo

Existen 2 modos de rastreo disponibles y son especificados al momento de ligar el programa.

- Modo Projections. Este modo genera un archivo detallado de todos los eventos Charm++ que se generan en la ejecución de la aplicación.  
Opción al ligar `-tracemode Projections`
- Modo Resumen. Este modo genera un archivo que muestra los datos importantes de la ejecución en intervalos de tiempos, menos detallado que el modo Projections, que sirve para obtener una idea general del comportamiento de la aplicación.  
Opción al compilar `-tracemode summary`

### 4.8.4 Herramienta de visualización de rendimiento

Charm++ viene con Projections precompilado y necesita del compilador de java para tener la aplicación ejecutable, ingresando a la carpeta `tools/projectios` y ejecutar *make*.

Inicialización : En la carpeta donde se encuentra Charm++ ejecutar

```
tools/projections/bin/projections [Nombre.sts]
```

y se desplegará una ventana como la de la figura 4.2 en donde aparecerán los resultados del monitoreo realizado.

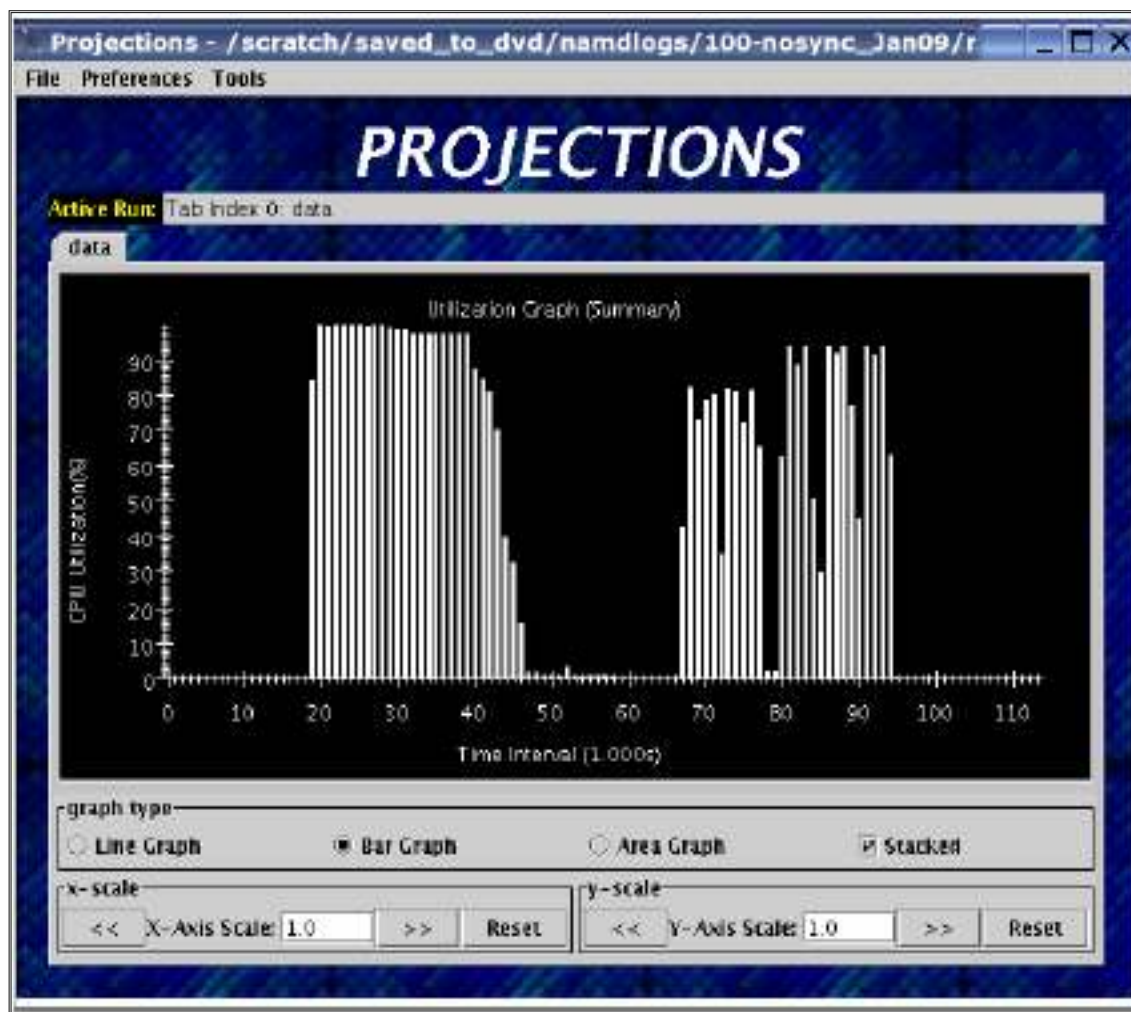


Fig. 4.5 Ventana de uso del procesador o procesadores

## 4.9 Debugger

El sistema de depuración de Charm++ tiene varias características que permiten al programador examinar los cambios durante la ejecución de una aplicación paralela, de igual forma que Projections esta hecho en Java y se ejecuta de forma visual sobre la computadora que actúe como el procesador 0 de tal forma que este funcionará como un servidor CCS (Converse Client-Server). CCS es un protocolo que habilita a un programa hecho con Charm++ actuar como un servidor y responder a peticiones de un cliente. Entre las características que encontramos están las siguientes:

- Se tiene un acceso fácil y visual de las entidades que conforman las aplicaciones, incluyen los arreglos de chares, la cola de mensajes a través de todos los procesadores durante la ejecución de la aplicación.
- Provee una interfaz donde se pueden colocar o quitar puntos de ruptura sobre puntos de entrada remotos (*entry methods*) con lo cual podemos tener un panorama de la ejecución en los demás procesadores.
- Tiene la posibilidad de pausar (*freeze*) el programa y continuar (*unfreeze*) la ejecución de un programa sobre la computadora paralela que utilizemos, para tener consistencia en el monitoreo previniendo que haya cambios por estar ejecutándose en paralelo.
- Permite tomar un subconjunto de procesos durante la ejecución en paralelo y asignarlo a un depurador secuencial como *gdb*, manteniendo un conjunto de ordenes secuenciales en otra ventana por separado.

#### 4.9.1 Generar la herramienta de depuración

En las versiones que existen de Charm++ trae consigo una versión compilada del depurador, este se encuentra en la carpeta:

```
charm_version/java/bin/charmdebug
```

esta versión viene compilada con java versión 1.4.2.

En caso de ser necesario se puede construir nuevamente el depurador con la versión de Java que se utilice con las siguientes órdenes sobre la misma carpeta:

```
\>cd charm/java
\>ant clean
\>ant
```

Como en Projections necesitamos recolectar información de la ejecución del programa así que necesitaremos opciones adicionales al momento de compilar y ligar la aplicación:

- La orden *-g* es necesaria para obtener los nombres de los archivos, de las funciones, los números de las líneas de código.
- Debido que pueden no haber concordancia entre el código generado para el depurador y el generado con optimización, estas opciones deben deshabilitarse.
- Al momento de ligar el programa la opción *-memory charmdebug* será necesario para obtener información necesaria de la memoria.

### 4.9.2 Record Replay

**Record Replay** es una característica independiente del depurador. Este es un mecanismo es utilizado para detectar fallos que pasan una o dos veces a lo mas. El programa al ejecutarse va guardando su trayectoria como con la herramienta **Projections**. La idea principal es etiquetar los mensajes para el emisor, y guardar el orden en que fueron generados por el mismo, Cham++ etiqueta los mensajes usando el procesador emisor y un contador de mensajes de salida de forma secuencial, esto significa que puede reproducirse la ejecución en el mismo orden en como fueron etiquetados los mensajes. Cuando el programa es ejecutado nuevamente en modo **replay** se utiliza la trayectoria anterior así es posible visualizar la ejecución y los procesos en el mismo orden en que se fueron ejecutando, el objetivo es verificar de forma secuencial la ejecución de la aplicación a fin de encontrar algún fallo en el código.

Para habilitar este modo al ligar la aplicación debemos agregar la siguiente opción:

**-tracemode recordreplay**

para guardar la trayectoria de la aplicación vamos a agregar la opción al momento de ejecutar

**+record**

Para reproducir la misma ejecución guardada cambiaremos el comando por el siguiente:

**+replan**

### 4.9.3 Sistema de Depuración Integrado de Charm++

Se ha creado un sistema de depuración con una interfaz gráfica (GUI) hecha en Java que podrá ejecutarse sobre la computadora que utilice el programador la cual actuara como un servidor. El depurador cuenta con los siguientes comandos que se utilizan desde el shell:

- **-pes** Numero de procesadores disponibles.
- **+p** Numero de procesadores a utilizar
- **-host** Nombre del servidor CCS para la aplicación
- **-user** Nombre de usuario para conectarse con el servidor elegido
- **-port** Número de puerto elegido para la conexión al servidor.
- **-sshtunnel** Fuerza la comunicación entre cliente y servidor a través de un túnel con ssh. Esto es útil cuando el servidor se encuentra detrás de un **firewall**.
- **-display** Para obtener una entorno gráfico del depurador

Para ejecutar la aplicación de forma local sobre 4 procesadores escribiremos en la línea de comandos lo siguiente

```
\>charmdebug programa +p4 [parámetros] [opciones]
```

si deseamos ejecutar en una cluster remoto que este detrás de un *firewall* escribiremos los siguiente:

```
\>charmdebug -host servidor -user nombre_usuario -sshtunnel programa  
+p4 [parámetros] [opciones]
```

También es posible visualizar en el depurador una ejecución guardada con **RecordReplay**, cabe aclarar que el depurador funcionará solamente en las construcciones de Charm++ de tipo *net-tipo\_de\_sistema*, para revisar en el depurador solamente agregamos la opción **+replay**

```
\>charmdebug programa +p4 [parámetros] [opciones] +replan
```

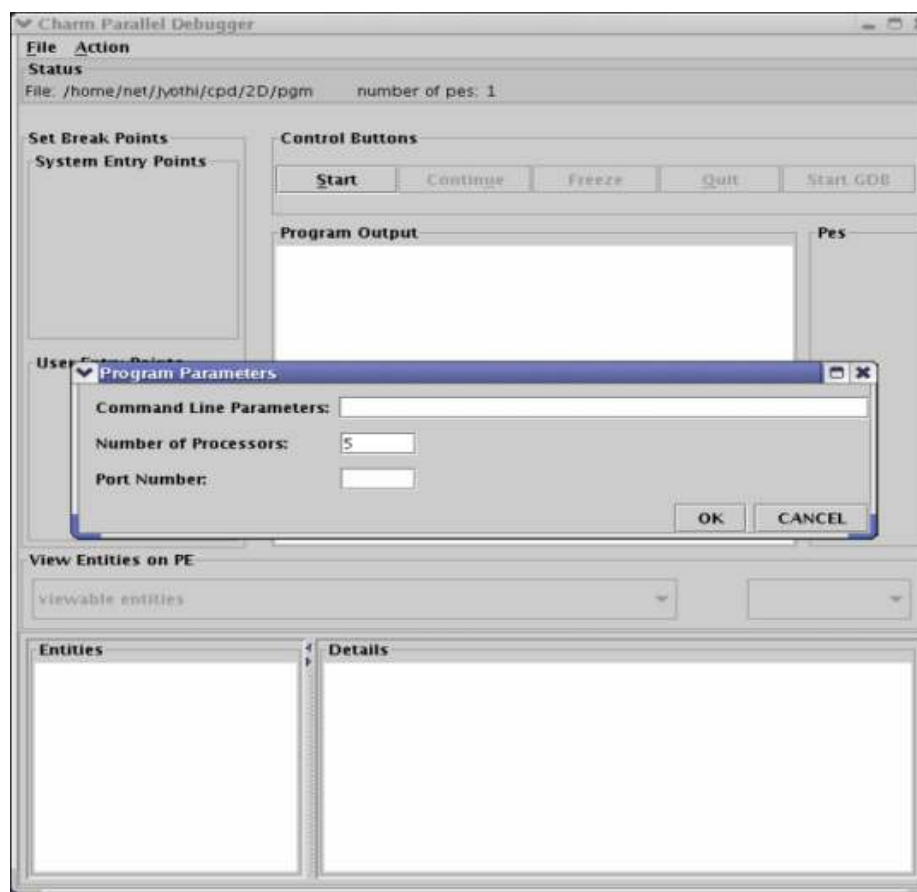


Fig. 4.6 Ventana del depurador de Charm++

## 4.9.4 Opciones del depurador

Cuando el depurador es llamado, automáticamente colocará las opciones a las que sean consideradas como las apropiadas. A continuación mostramos una lista de las opciones disponibles:

- **+cpd** Envía a pausar al inicio la ejecución.
- **++charmdebug** Lanza a charmrn de forma que recolecte la información acerca de la aplicación ejecutable y tener información para usar con ***gdb*** ( otro depurador de aplicaciones)
- **++debugger** para utilizar el depurador
- **++debug** con esta opción podemos utilizar depuradores secuenciales como ***gdb*** o ***dbx*** separando la ejecución en cada procesador para poder analizarla. Esto ejecutara abrirá una Terminal por cada procesador.
- **++debug-no-pause** lo mismo que el anterior comando pero inicia la ejecución inmediatamente
- **++DebugDisplay** Visualización en una terminal xterm
- **++server-port** puerto de comunicación con el servidor CCS
- **++server** Habilita el modo cliente-servidor
- **+record** comando para grabar o guarda la trayectoria para ejecutar la secuencia exacta posteriormente.
- **+replay** Fuerza al sistema a ejecutar la aplicación con la misma secuencia que la grabada o guardada.

## 4.9.5 El Depurador

Una vez que es cargado el depurador, el programador inicia la ejecución de su aplicación con el botón ***Start***, cuando es ejecutado desde la línea de comandos la aplicación inicia de forma automática. El usuario puede elegir activar el despliegue de información como puntos de entrada que por defecto vienen deshabilitados, puede también escoger puntos de ruptura con un clic sobre un punto de entrada y pasar sobre los puntos con el botón ***Continue***.

En la figura 4.7 observamos el momento en que un punto de ruptura es alcanzado, en este punto el programa tiene una pausa hasta oprimir el botón ***Continue***. Los puntos de ruptura que se crean son puntos de entrada que el sistema crea en lugar de los puntos que se define en el código, así puede detener la ejecución en un punto donde debería ejecutarse un método de entrada.

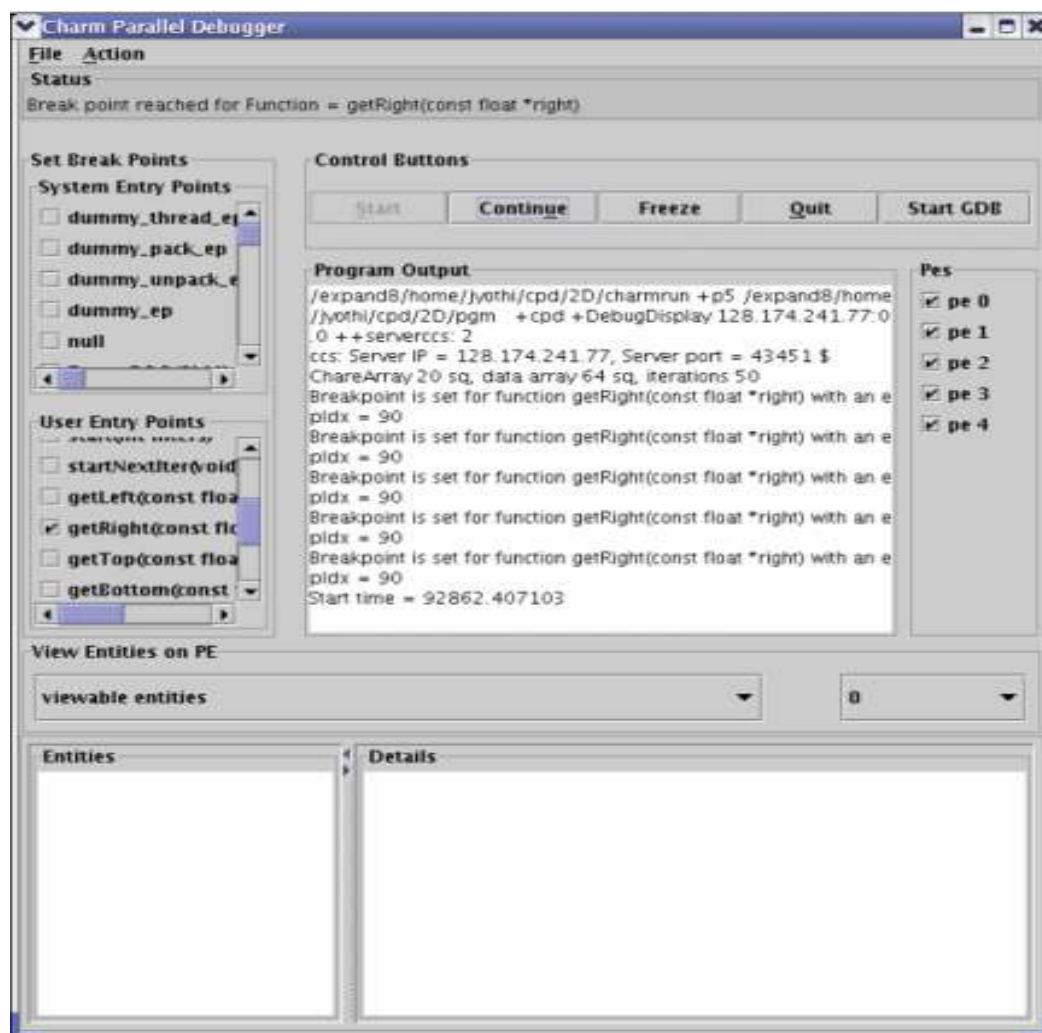


Fig. 4.7 Depurador cuando un punto de ruptura es alcanzado

Al oprimir el botón **Quit** se abortará la continuidad del programa en cualquier punto donde se encuentre la ejecución. Varios elementos como una matriz pueden ser visualizados en cualquier punto de la ejecución como se puede apreciar a continuación.

Los objetos *chare* pueden ser examinados por medio de la vista de entidades **Display selector** sobre un procesador el cual permite elegir los chares, los arreglos, mensajes que estén en la cola, las variables, los puntos de entrada, los *chares*, mensajes de un tipo definido por el usuario y la chare principal la *Mainchare*. Dando un clic sobre la entidad se pueden revisar los detalles de esta. En la figura 4.8 aparece esta información en la parte inferior de la ventana, a la derecha las entidades y a la izquierda sus detalles.

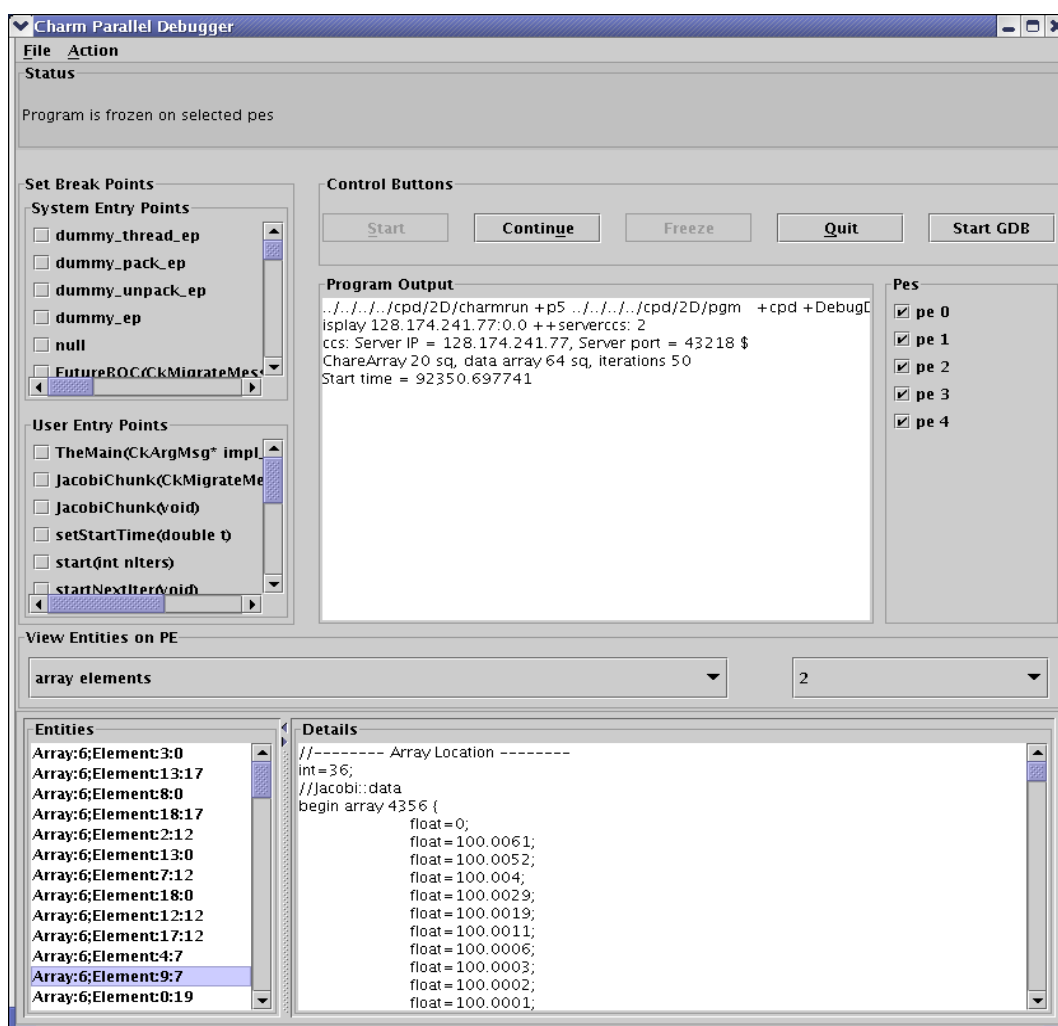


Fig. 4.8 Elementos de una matriz en un punto detenido de la aplicación.

Charm++ nos da la posibilidad de tomar procesos individuales y mostrarlos en el depurador *gdb* como se muestra a continuación.

Se puede elegir a cual procesador se conectará el depurador *gdb* marcando las casillas que se encuentran a la derecha, mientras este detenida la ejecución por medio de *gdb* las opciones de inspección de objetos no funcionarán.

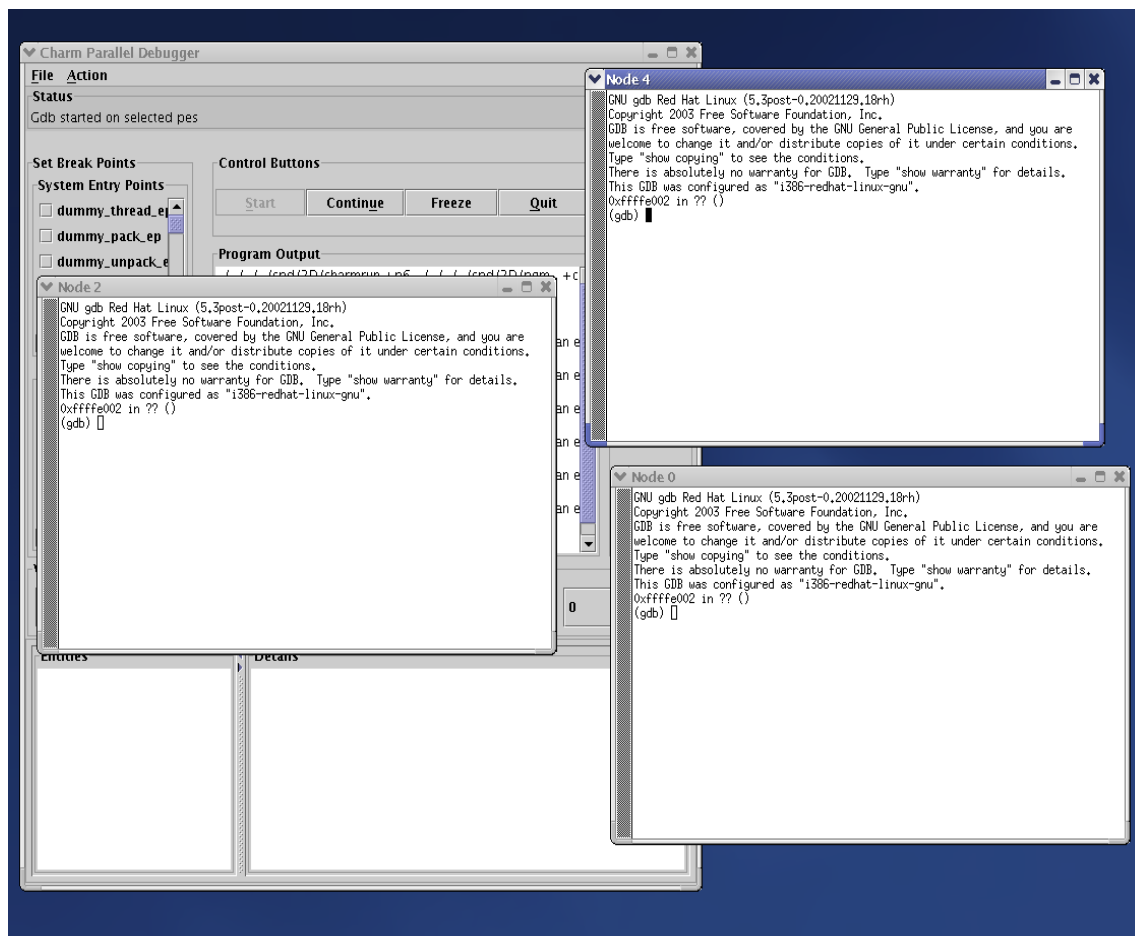


Fig. 4.9 El depurador puede usar instancias de gdb sobre los procesadores

#### 4.9.5.1 Visualización de la Memoria

En el menú Action/Memory podemos visualizar el uso de la memoria sobre algún procesador en específico. A continuación mostramos una imagen y la explicación de lo que significan los colores.

1. **Rojo** memoria asignada al Sistema De Ejecución de CHARM++.
2. **Azul** memoria asignada al código del usuario.
3. **Rosa** memoria utilizada por los mensajes.
4. **Naranja** memoria asignada a un *chare*.
5. **Negro** memoria sin asignar.
6. **Gris** representa una porción grande de memoria virtual no utilizada entre dos zonas diferentes de direcciones de espacio virtual.
7. **Amarillo** espacio de memoria asignado actualmente.

Esta configuración de colores no es posible cambiarla como en *Projections*.

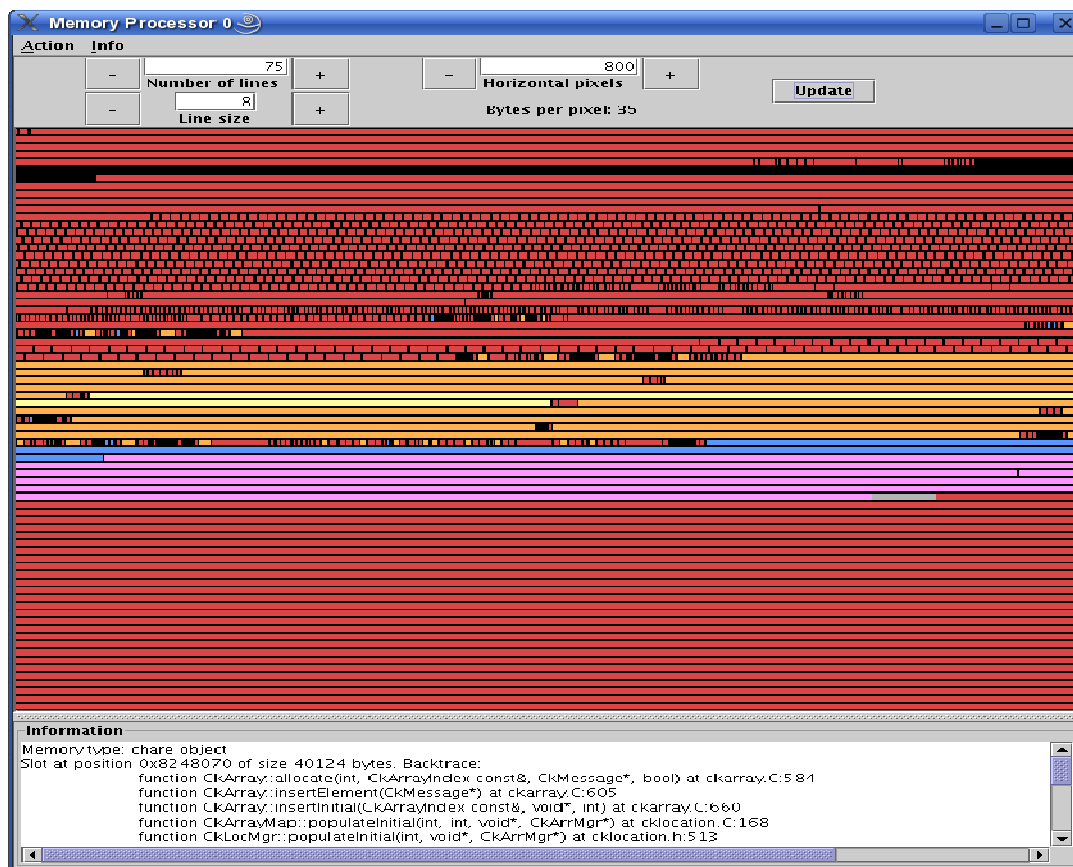


Fig. 4.10 Vista de la memoria principal

En el menú la opción Info /Show Statistics desplegará las siguiente ventana con información de la memoria utilizada y los bloques asignados.

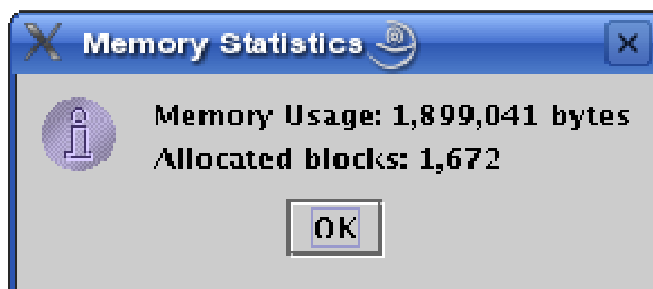


Fig. 4.11 Estadísticas de la memoria

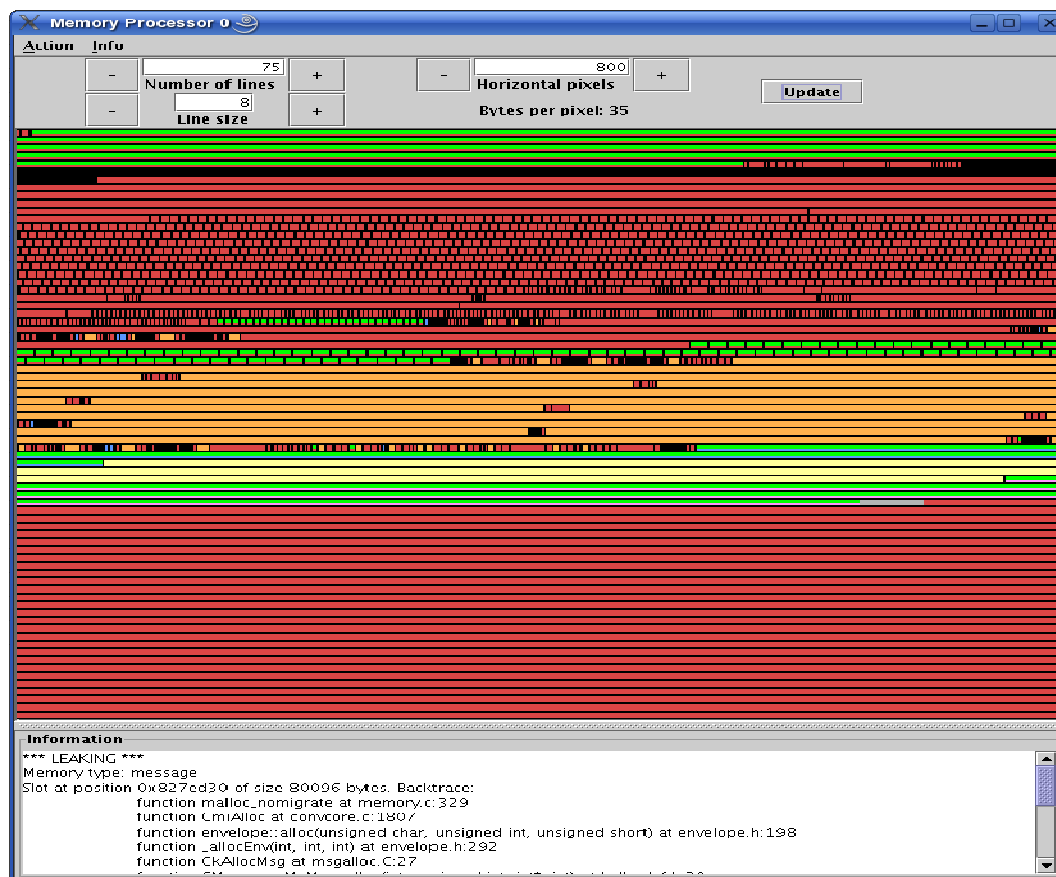


Fig. 4.12 Memoria después de ejecutar la prueba de acceso

También puede verse la memoria que no esta siendo utilizada pero tampoco puede ser asignada, en el menú Action / Search Leaks esto ejecutará una prueba accesibilidad de esta forma podremos visualizar la memoria que no esta siendo utilizada en color verde.

#### 4.9.6 Inspector framework

Sin realizar cambios en la aplicación *CharmDebug* puede cargar parte de la memoria en bruto y analizarla buscando por un nombre a algún *chare*, de esta forma obtendremos un árbol con las tablas de punteros virtuales de objetos, métodos y variables que pertenezcan a este *chare*

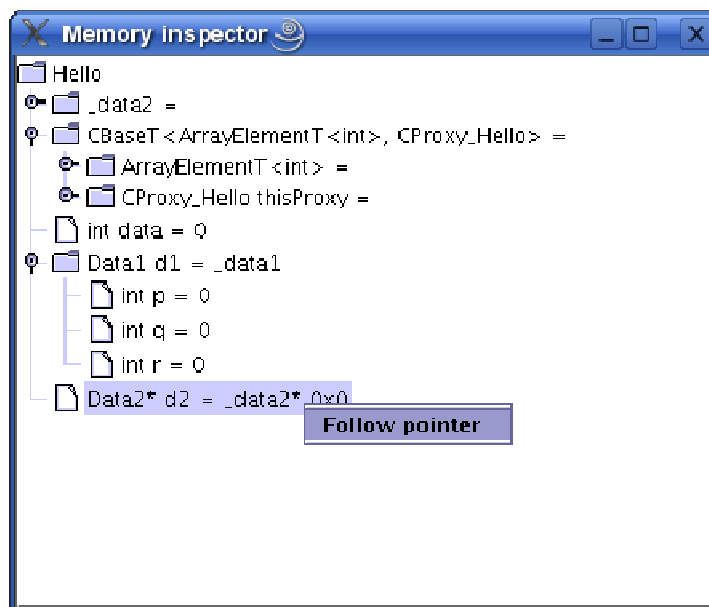


Fig. 4.13 Árbol de la variables y funciones virtuales

Cuando la vista se abre y se muestra un tipo, haciendo clic derecho sobre una rama que contiene un puntero a otra ubicación de memoria, un menú emergente permite al usuario solicitar los datos a los cuales hace referencia. En este caso, CharmDebug también cargará esta información en bruto y se podrá analizar mediante el nombre del puntero, esta información se convertirá en otra rama o nodo del árbol ya creado.

#### 4.9.7 Limitaciones del Depurador

Como mencionamos anteriormente solamente se puede utilizar el depurador en instalaciones de tipo *net-tipo\_de\_sistema* usarlo en otro tipo puede traer resultados inesperados también existe la posibilidad de poder utilizarlo en un futuro en implementaciones *mpi* de *Charm++*.

La característica *RecordReplay* no funciona correctamente en algunos casos de eventos espontáneos como es el balance de carga que en cualquier momento puede migrar trabajo entre procesadores, el balance realiza comprobaciones periódicas a las colas de *chares* en los procesadores aún así la naturaleza del balance de carga genera migraciones espontáneas. Por otra parte, si una aplicación termina inesperadamente, puede perderse registro de la trayectoria o el seguimiento de la misma.

# Capítulo 5

## Aplicaciones De Ejemplo

La Facultad de Ciencias de la Computación cuenta con un cluster de computadoras con las siguientes características:

- 16 nodos en total
- una interconexión fast-ethernet a una velocidad de 1 Gbps

Donde cada nodo tiene:

- 2 Procesadores AMD Opteron de 64 bits
- 2 GB RAM
- 36 GB HD SCSI

un disco compartido sobre todos los nodos de 36 GB SCSI y Sistema Operativo Fedora Core 8. Adicionalmente tiene instalado sobre el OSCAR Cluster una aplicación para administrar clusters que incluye los entornos de programación MPI y PVM.

Sobre este equipo se realizarán los siguientes ejemplos:

1. Calcular el valor de  $\pi$  por el método de Montecarlo.
2. Encontrar los primos que existen en un rango
3. Factorizar un número entero mediante el método de Fermat

Debido a diversas situaciones solo nos fue posible utilizar 10 nodos como máximo para estos ejemplos, es decir, 20 procesadores en total, los ejemplos se ejecutaron tomando desde un solo procesador hasta llegar al máximo de veinte, así veremos si incrementa el rendimiento al tomar más procesadores en cada ejecución y determinar si es o no viable la opción paralela para nuestras aplicaciones.

## 5.1 Calcular el valor de $\pi$ por el método de Montecarlo.

En esta aplicación vamos a realizar el cálculo del número  $\pi$  por el método de Montecarlo. Este método consiste en escoger varios puntos  $(x,y)$  de forma aleatoria en el rango de  $[-1,1]$  y tomar los puntos que caen dentro de una circunferencia de radio 1.

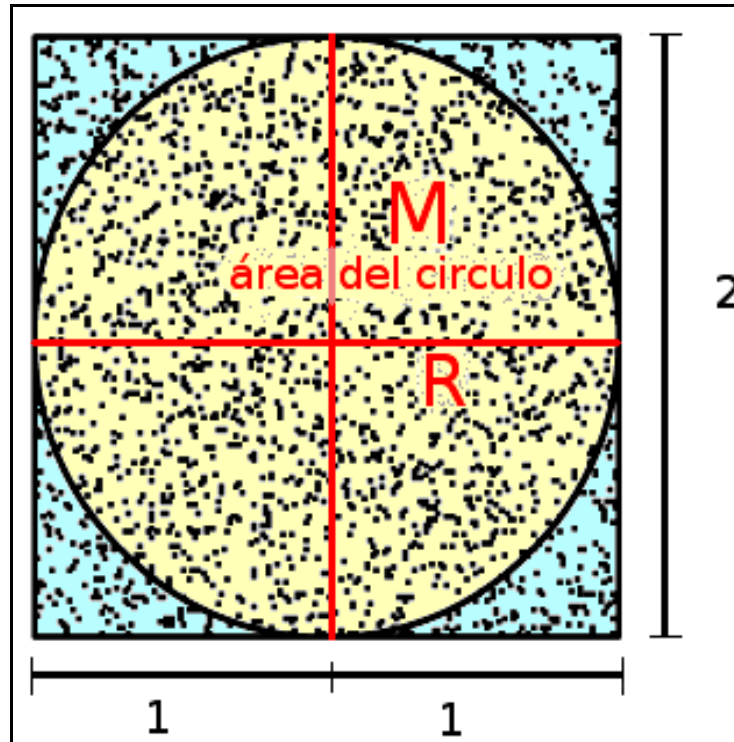


Fig. 5.1 Representación de puntos aleatorios en cuadrado y círculo inscritos.

Se van sumando los puntos dentro de la circunferencia, para validar dichos puntos nos valemos de la fórmula de la circunferencia y obtenemos los puntos que son menores al radio que en este caso es uno.

$$\sqrt{x^2 + y^2} \leq 1$$

Ahora si tenemos un círculo de radio  $R$  inscrito dentro de un cuadrado de lados igual a  $2R$ , el área del círculo es  $\pi R^2$  y el área del cuadrado es  $2R \cdot 2R = 4R^2$ . Así la razón entre el área del círculo y el área del cuadrado es:

$$\frac{\pi R^2}{4R^2} = \frac{\pi}{4}$$

De esta manera si tomamos  $N$  puntos al azar dentro del cuadrado tendremos que  $N\frac{\pi}{4}$  puntos estarán dentro del área del círculo  $M$ .

$$M = N \frac{\pi}{4}$$

Despejando  $\pi$  obtendremos la siguiente formula:

$$\pi = 4 \frac{M}{N}$$

En este ejemplo se van a crear objetos **chare** que calculará de un determinado número de intentos cuantas veces los puntos aleatorios  $(x,y)$ , caen dentro de la circunferencia; una vez finalizado se suman todos los puntos de todos los objetos y la **Mainchare** mostrará en pantalla el resultado de  $\pi$  usando la fórmula anterior.

Primero en el archivo de cabecera (.h) colocaremos las declaraciones de los objetos y variables que utilizaremos. Primero siempre comenzaremos con la línea **#include “nombre\_del\_program.decl.h”** el cual se genera de forma automática por el Sistema de Ejecución Charm++.

```
#include "pi.decl.h"

class main : public Chare
{
```

Se realiza la declaración de las variables

- ni – números de intentos / nc numero de chares
- inicio, final y tiempo para determinar el tiempo que dura la ejecución
- cnt el numero de aciertos dentro de la circunferencia

```
private:
    int procesosFaltantes;
    int ni, nc;
    double inicio, final, tiempo;
    long double cnt, pi;
```

La chare **main** creará los objetos para realizar los cálculos

```
public:

    main(CkMigrateMessage *m) {}
```

```

    main(CkArgMsg *m);
    // La chare resultados para imprimir en pantalla pi
    void resultado(long double c);
};

//arregloPi un arreglo de chares para realizar los cálculos
class arregloPi : public ArrayElement1D
{
    public:

    // constructor para inicializar
    arregloPi(CkMigrateMessage *m) {}
    arregloPi(void);

    // Chare que realizara los cálculos como argumento tendrá
    //el numero de intentos ni
    void calcula(int ni);
};

```

Después tenemos el archivo de interfaces (**.ci**) donde declaramos los métodos de entrada y las variables que corresponden directamente a Charm++ como los handles y los proxies.

```

mainmodule pi {

    readonly CkChareID mainhandle;

```

declaramos la chare principal con la palabra clave **mainchare** cabe que recordar no es necesario que se llama **main**, anteponiendo esta palabra clave el Sistema de Ejecución de Charm++ reconocerá cual es la chare principal que es la que inicia todos los cálculos en una aplicación.

```

mainchare main
{
    entry main();
    entry void resultado(long double cnt);
};

```

También declaramos las demás chares que utilizaremos en este caso el arreglo unidimensional **arregloPi** el cual tiene como métodos de entrada el constructor y la función calcula que buscará en un determinado número de intentos **ni** cuales caen dentro de una circunferencia.

```

array [1D] arregloPi
{
    entry arregloPi(void);
    entry void calcula(int ni);
};

};

```

Ahora tenemos el código fuente que es el archivo con extensión **.C** o **.cpp** donde veremos explícitamente el proceso para calcular  $\pi$ . Iniciamos con las cabeceras donde

colocaremos las librerías que utilizemos, aquí siempre deberemos usar la librería con el nombre del programa. En esta aplicación usaremos proxies así que necesitamos un identificador global que será *mainhandle*.

```
#include "pi.h"

CkChareID mainhandle;
```

La función principal donde primero obtendremos los parámetros *ni* los intentos y *nc* el número de tareas que realizarán estos cálculos. En la aplicación sino son 3 los parámetros obtenidos el programa finalizará su ejecución. El tercer parámetro que es el primero en realidad, es el número de procesadores que se utilizarán, usando *+p* seguido del número de procesadores.

```
main::main(CkArgMsg * m)
{
    if(m->argc != 3)
    {
        CkPrintf("Deben ser 2 parámetros, <total de intentos>
                  <total de chares>\n");
        CkAbort("");
    }

    ni = atoi(m->argv[1]); // total de intentos
    nc = atoi(m->argv[2]); // total de procesos [chares]
    delete m;
```

Inicializaremos el contador *cnt* y el tiempo para saber el tiempo que toma en ejecutarse nuestra aplicación. Crearemos un arreglo de chares que serán las que serán enviadas a los procesadores para su cálculo. Aquí es donde definimos que habrá un cliente de reducciones guardaremos en *mainhandle* el identificador del chare *main* y usaremos la función *calcula* del arreglo para empezar los cálculos.

```
//tomamos el tiempo en que se inicia
inicio = CmiWallTimer();

// se crea un arreglo de tipo Cproxy_arregloPi

CProxy_arregloPi arreglo = CProxy_arregloPi::ckNew(nc);

//definimos al cliente para las reducciones
arreglo.setReductionClient(reductor, (void *)NULL);

arreglo.calcula(ni); //se envían los arreglos para procesar
cnt = 0;
mainhandle = thishandle; // se obtiene el handle del main

}
```

Para enviar los resultados a pantalla tenemos la función resultado de la chare **main** la cual solo va a realizar la operación  $\pi = 4 \frac{M}{N}$  una vez obtenidos el total de puntos que cayo dentro de la circunferencia e imprimirá el resultado con la función **CkPrintf()**. Por ultimo se llama a la función **CkExit()** para que se finalicen los procesos y no se realice algún otro proceso pendiente.

```
void main::resultado(long double cnt)
{
    long double aux2,pi;

    // Total de puntos dentro del circulo
    CkPrintf("Sumas Totales : %Lf \n",cnt);

    final = CmiWallTimer();
    tiempo = (final-inicio);

    // calculo de pi por probabilidades
    aux2 = cnt * 4 /nc;
    pi = aux2 / ni;

    //imprime a pantalla el resultado

    CkPrintf("Finalizado después de %f segundos, valor de
            pi=: %.10Lf \n",tiempo , pi);

    // finaliza la aplicación
    CkExit();
}
```

Esta parte corresponde a las funciones que realizará el arreglo de chares, primero tendremos al constructor que solamente obtendrá un número aleatorio.

```
arregloPi::arregloPi()
{
    CrnSrand((int)(long) this); // constructor para arregloPi
}
```

La función **calcula** que obtendrá puntos aleatorios y determinara cuantos estarán dentro de la circunferencia, posteriormente llamará a la función de reducción **contribute** con tipo **sum\_double** que es el tipo que vamos a usar para sumar todos los puntos que serán en el orden de millones de puntos.

```
void arregloPi::calcula(int ni) // calculo de probabilidades
{
    int i;
    double count=0;

    for (i= 0; i<ni; i++) {
        double x = CrnDrand(); // se obtiene x aleatoria
        double y = CrnDrand(); // se obtiene y aleatoria
```

```

        // se calcula que punto (x,y) cae dentro de la circunferencia
        if ((x*x + y*y) <= 1.0) count++;
    }

    // se utilizan reducciones de suma con tipo double
    contribute(sizeof(double), (void *)&count, CkReduction::sum_double);
}

```

En este caso usaremos reducciones, así que necesitaremos un cliente que haga las reducciones con lo cual usaremos reductor que recibirá todos los datos de las *chare* y el Sistema De Ejecución se encargara de determinar que procesadores realizaran estas tareas, una vez teniendo todos los resultados se enviaran a la función resultado de la *chare* principal para imprimir el resultado final, el valor de  $\pi$ .

```

void reductor(void *param, int tamano, void *sumas)
{
    CProxy_main principal(mainhandle);
    double aux = *(double *)sumas;
    principal.resultado(aux);
}

#include "pi.def.h"

```

Por último el archivo *Makefile y nodelist*. *Makefile* nos va a ayudar en la compilación, ejecución y limpieza de la aplicación. Al inicio vamos a colocar la ubicación del compilador de Charm++

```
CHARMC= ../../charm-6.0/net-linux-amd64/bin/charmcc
```

en esta parte es donde realizamos la compilación del programa, es similar a la de C o C++ solo que usando *charmcc* y los parámetros *-language charm++* para definir el lenguaje a utilizar, *-tracemode projections* para utilizar obtener información acerca de la ejecución usando *projections* y *-tracemode summary* para obtener un seguimiento mas general del comportamiento de aplicación esta opción corresponde a la herramienta *projections*.

```

all: pi

pi: pi.o

$(CHARMC) pi.o -o pi -language charm++ -tracemode
        projections -tracemode summary

```

Con esta orden ejecutamos la aplicación usando *charmrun* que será creada en la misma carpeta donde se llame al compilador después sus argumentos el primero los

procesadores que usaremos representado por **+pX** donde **X** es un número cualquiera, cabe resaltar que se puede colocar un número mas grande que el total de procesadores disponibles, en este caso, el **Sistema de Ejecución de Charm++** reubicara los procesos en los procesadores siguiendo el orden que tienen en el archivo **nodelist**. Los siguientes 2 números son el total de intentos a realizar y el total de procesos a crear y por ultimo se le indica que archivo **nodelist** hay que utilizar, si no se define en esta parte el Sistema tomará el archivo **.nodelist** que se encuentra en el directorio **home** del usuario

```
run: pi
    ./charmrun +p10 pi 1000000 100000 ++nodelist nodelist

pi.o : pi.C pi.h pi.decl.h
      $(CHARMC) -c pi.C

pi.decl.h: pi.ci
      $(CHARMC) pi.ci
```

En esta parte eliminaremos los archivos generados automáticamente al momento de la compilación si realizamos algún cambio y nos genera algún error tener alguno de estos archivos

```
clean:

rm -f pi *.o conv-host *.decl.h *.def.h *.sts *.log *.sum *~ charmrun
```

El archivo **nodelist** tiene la lista de los procesadores a utilizar en este caso definimos 10 entonces quedara de la siguiente forma:

```
group main ++shell ssh
host oscarnode03
host oscarnode03
host oscarnode04
host oscarnode04
host oscarnode06
host oscarnode06
host oscarnode07
host oscarnode07
host oscarnode09
host oscarnode09
```

Se define un grupo **main** y se va a utilizar la comunicación mediante **ssh** con la orden **++shell ssh**.

A continuación presentamos la tabla con los resultados y gráficas que nos dan una idea del rendimiento que van obteniendo al ir aumentando el número de procesadores.

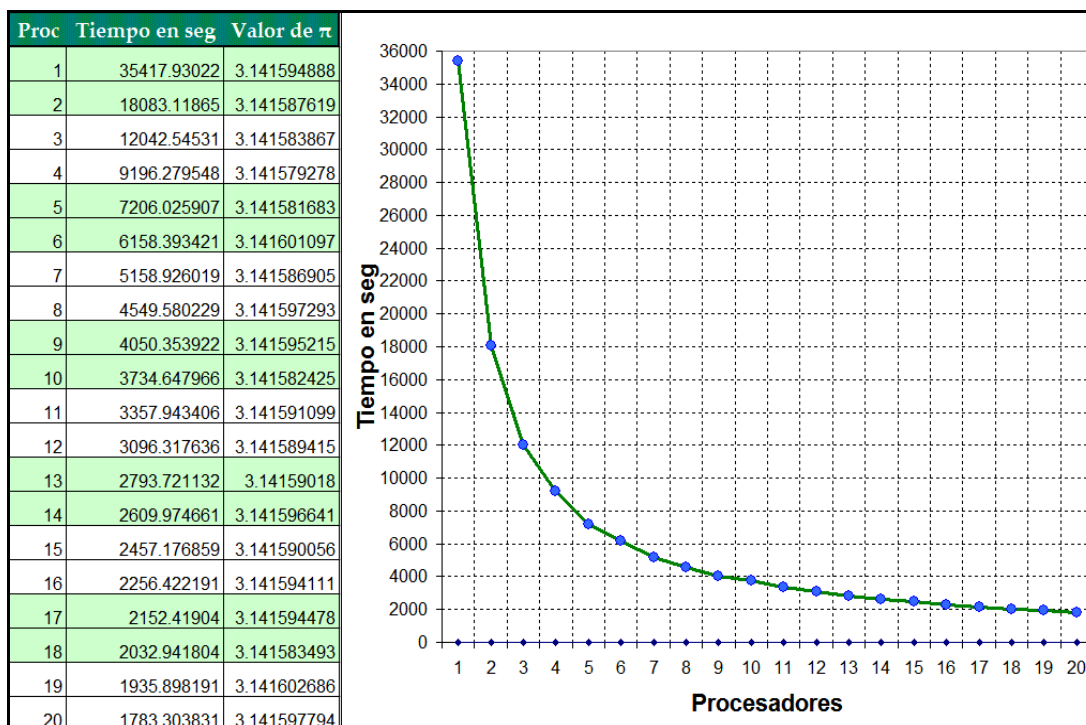


Tabla 2 Tabla de resultados y gráfica de tiempos de ejecución

Y también tenemos el comportamiento de los resultados de  $\pi$

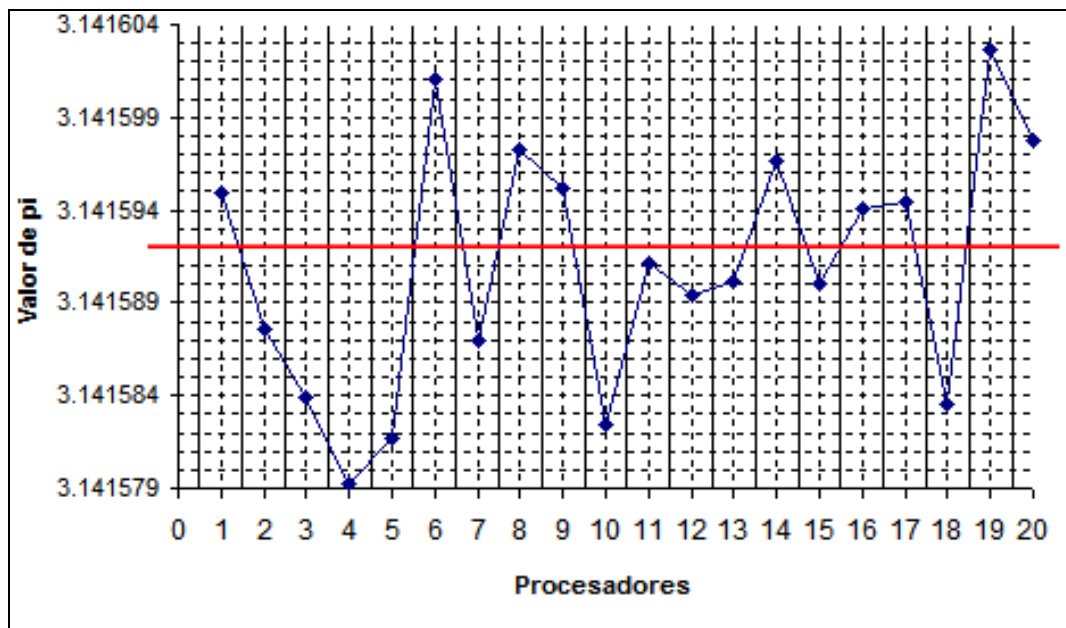


Tabla 3 Gráfica de los valores obtenidos de  $\pi$

Y por ultimo el comportamiento de los procesadores en la ejecución de la aplicación desde 1 procesador hasta el total utilizado: 20. Vemos que en un procesador se mantiene un aprovechamiento equilibrado, esto es, el balance de carga se efectúa correctamente.

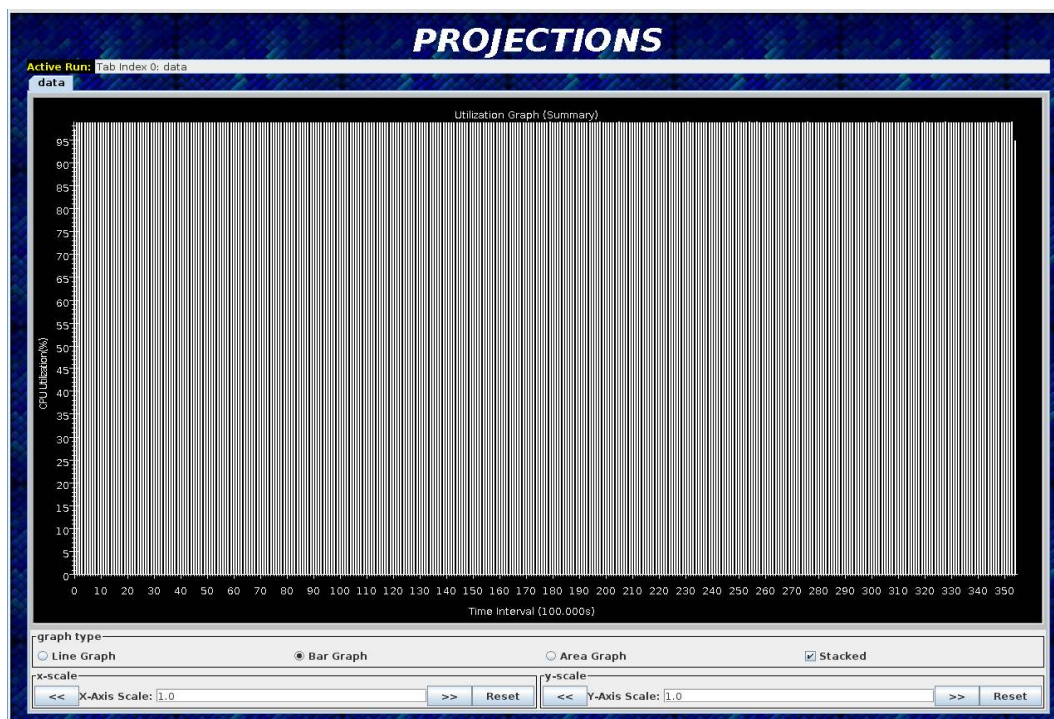


**Fig. 5.2** Gráfica de uso de procesamiento con 1 procesador

En esta parte es donde se comienzan a ver cambios al final se reduce el aprovechamiento a la mitad, esto es debido al proceso de reducción o recolección de los resultados 1 procesador realiza la recolección de la información.

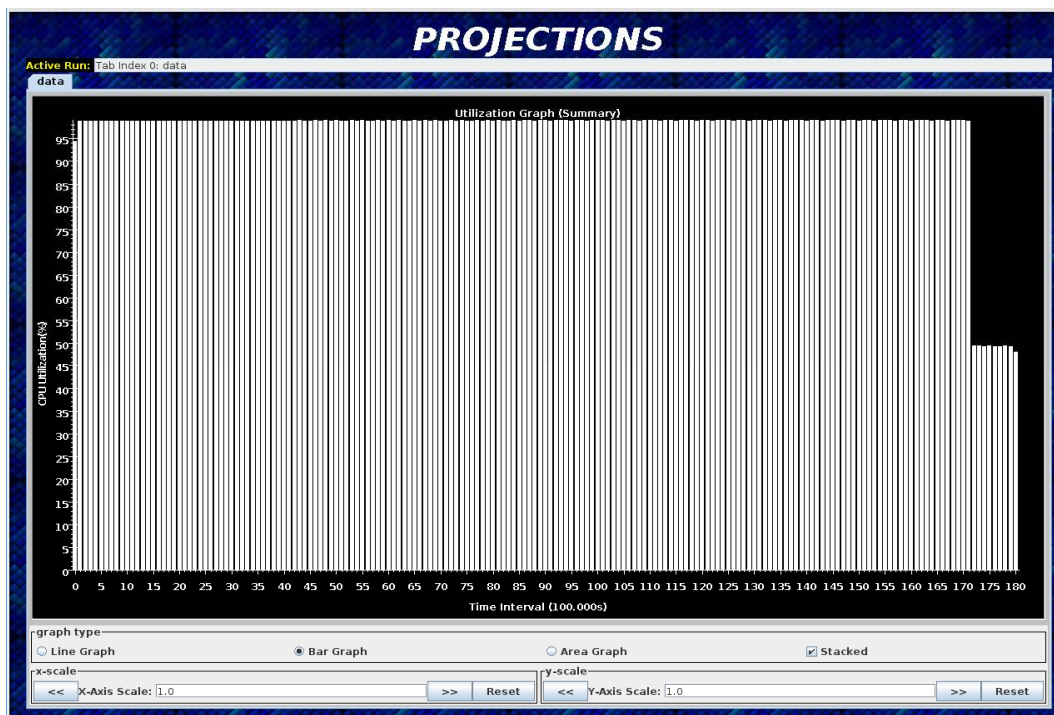


**Fig. 5.3** Gráfica de uso de procesamiento con 2 procesadores

En esta parte vemos como las reducciones estabilizan el trabajo ahora no será un solo procesador el encargado si alguno de los procesadores se libera de carga también ayudará a la recolección de los datos.

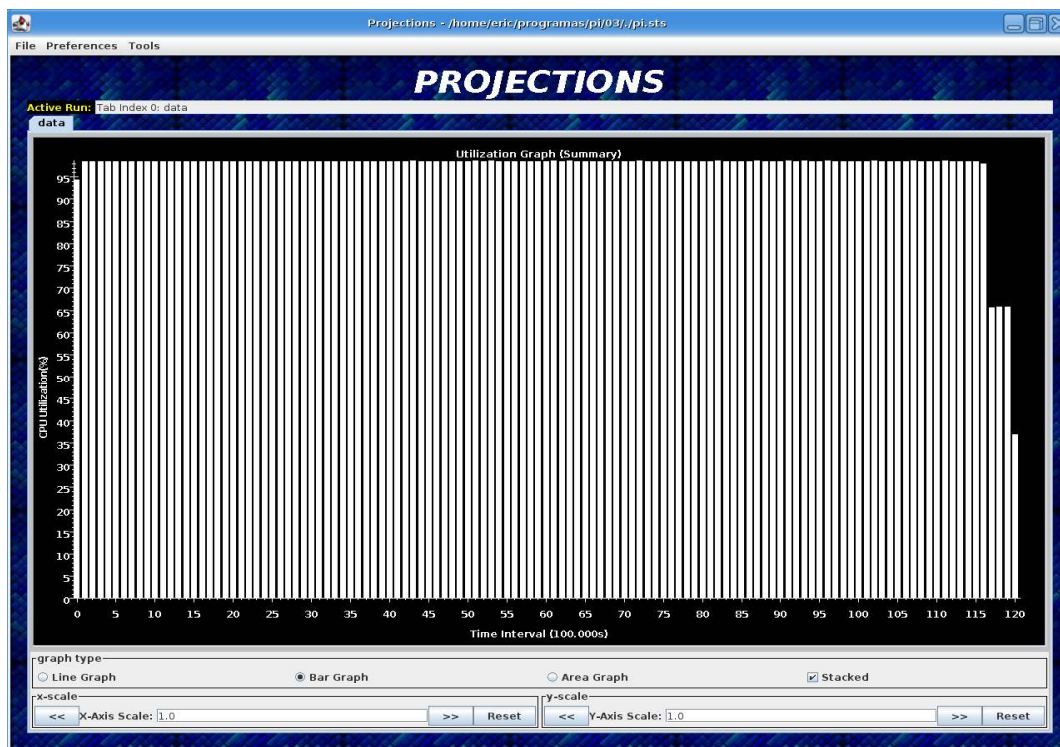


Fig. 5.4 Gráfica de uso de procesamiento con 3 procesadores

El mismo comportamiento anterior con 3 y 4 procesadores se puede visualizar, la carga se equilibra hasta llegar a un punto donde solamente trabaja un procesador

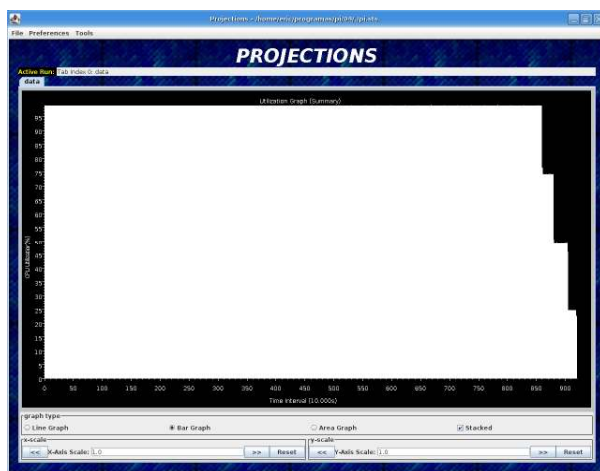


Fig. 5.5 Gráfica con 4 procesadores



Fig. 5.6 Gráfica con 5 procesadores

Lo mismo seguimos viendo con cuando se utilizan más procesadores, tienen una carga similar. En las sumas finales donde se obtienen los resultados se equilibra la carga ya

que el uso de los procesadores no cae inmediatamente, sino poco a poco debido a los procesadores que les sea posible ayudarán en la recolección de los datos, es esta la ventaja de usar reducciones en lugar de regresar los resultados a un solo punto o procesador.

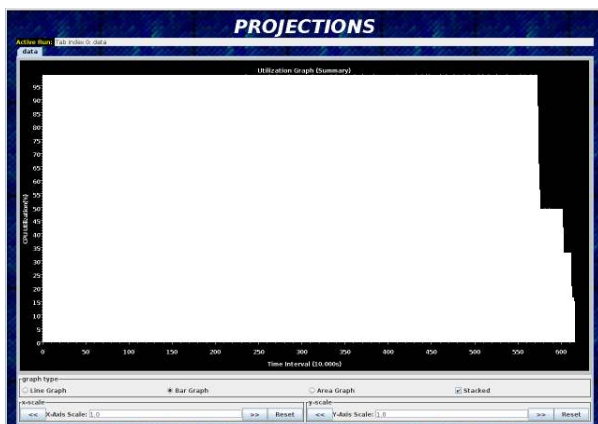


Fig. 5.7 Gráfica con 6 procesadores

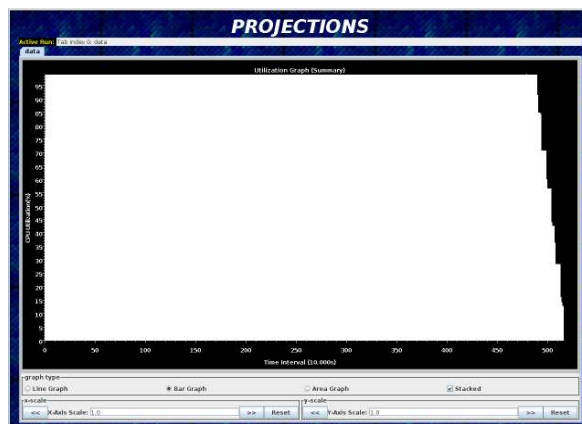


Fig. 5.8 Gráfica con 7 procesadores

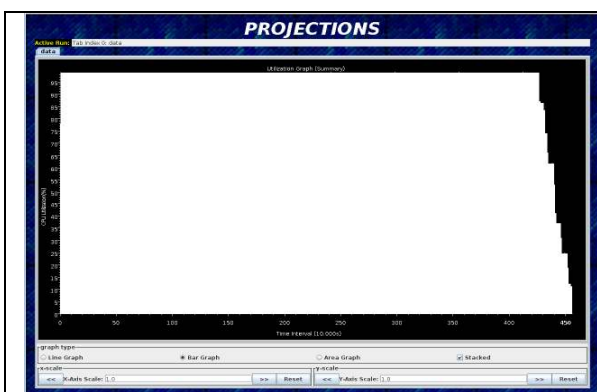


Fig. 5.9 Gráfica con 8 procesadores

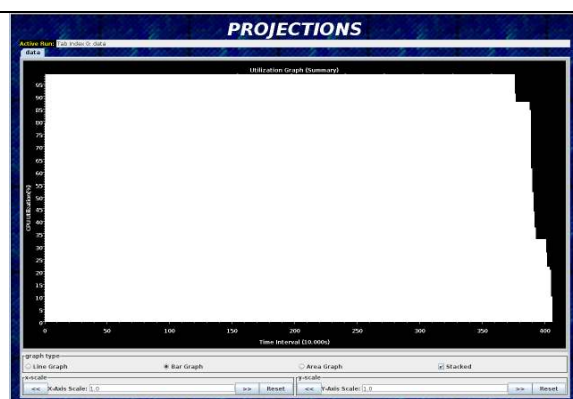


Fig. 5.10 Gráfica con 9 procesadores

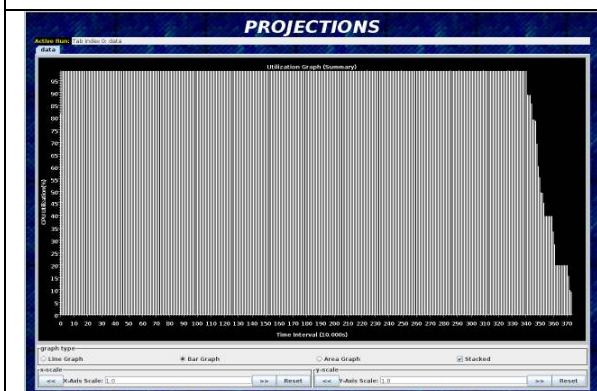


Fig. 5.11 Gráfica con 10 procesadores

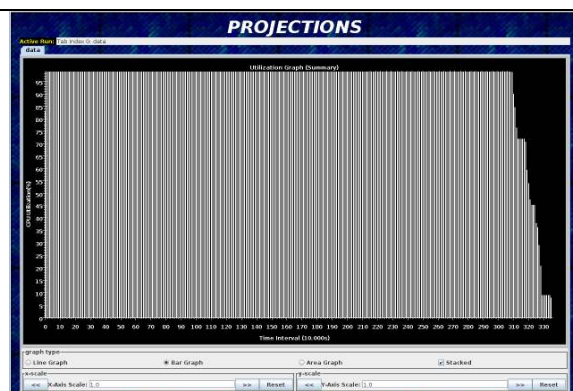


Fig. 5.12 Gráfica con 11 procesadores

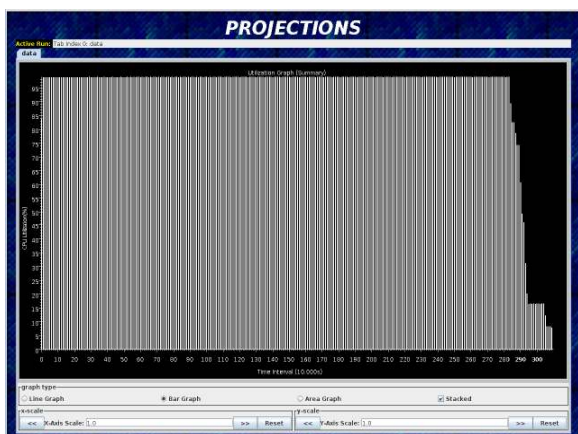


Fig. 5.13 Gráfica con 12 procesadores

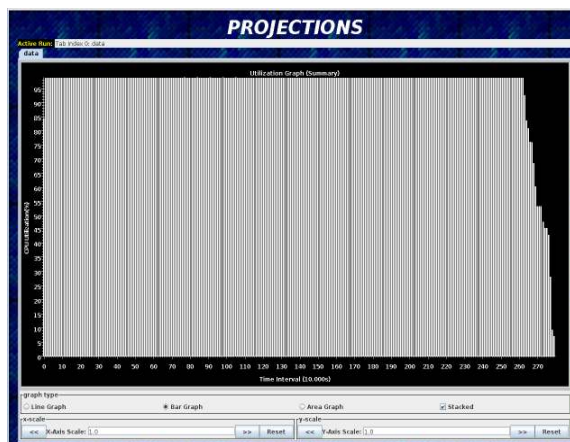


Fig. 5.14 Gráfica con 13 procesadores

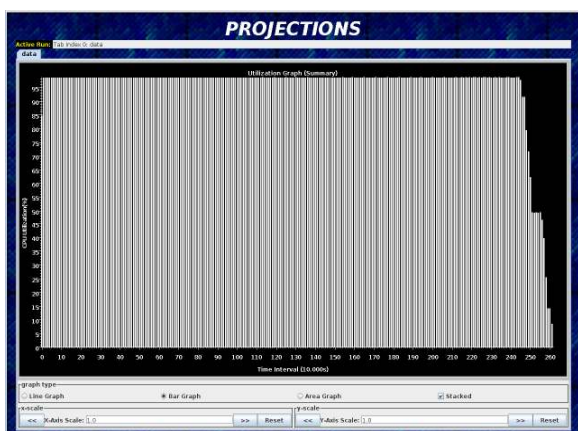


Fig. 5.15 Gráfica con 14 procesadores

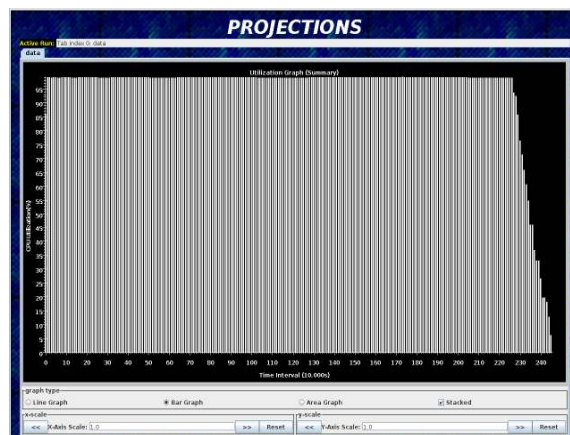


Fig. 5.16 Gráfica con 15 procesadores

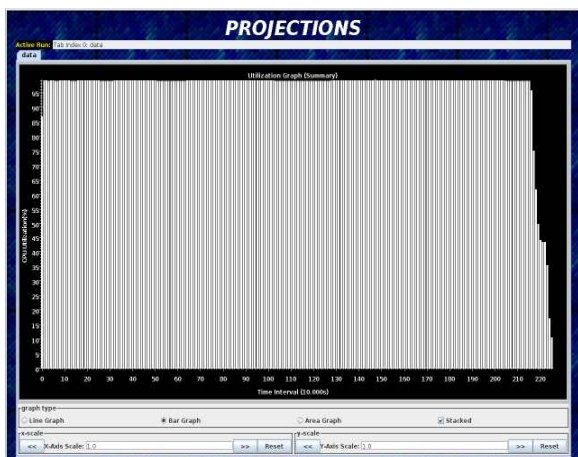


Fig. 5.17 Gráfica con 16 procesadores

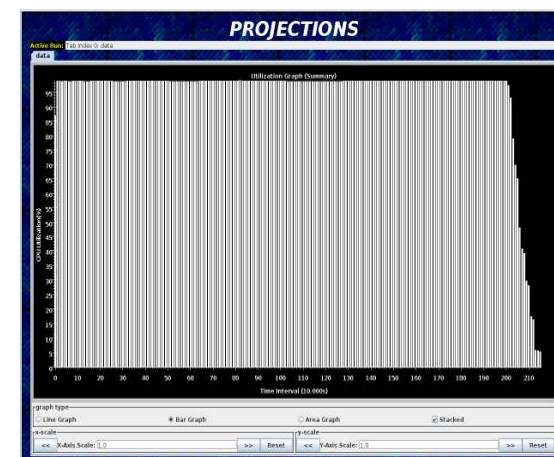


Fig. 5.18 Gráfica con 17 procesadores

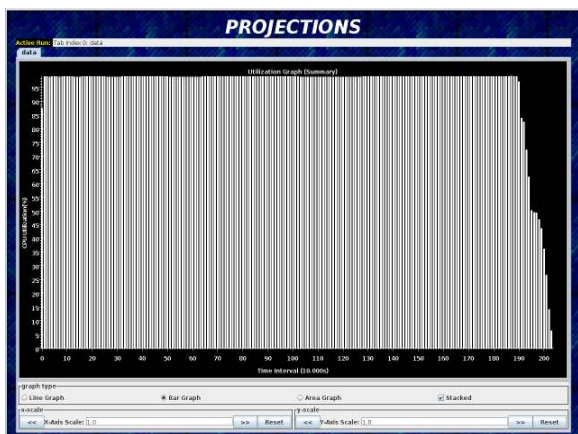


Fig. 5.19 Gráfica con 18 procesadores

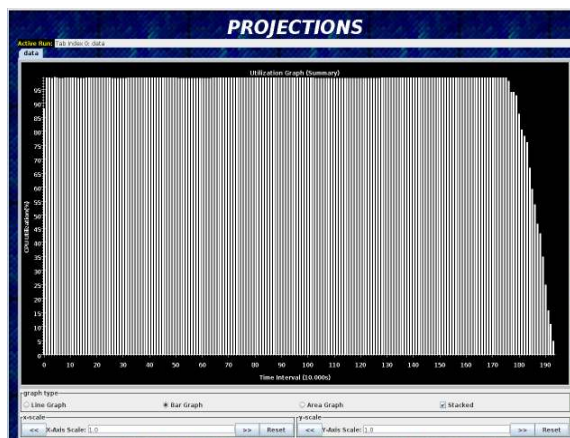


Fig. 5.20 Gráfica con 19 procesadores

En la última gráfica continua el mismo comportamiento no cae el rendimiento inmediatamente ya que en conjunto van realizando todas las sumas, para llegar al resultado de  $\pi$  y solamente un procesador realiza la cuenta final que será el método de entrada *main::resultado*.

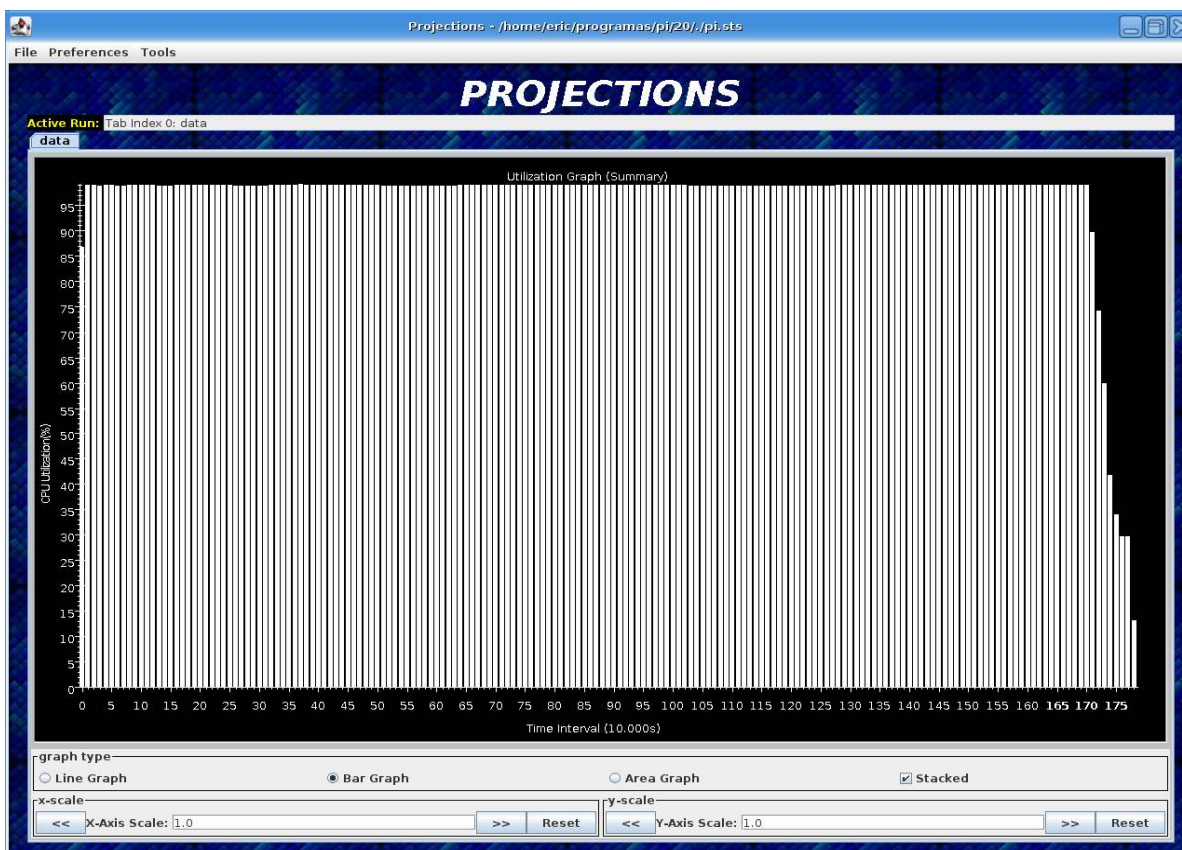


Fig. 5.21 Gráfica con 20 procesadores

También con *Projections* obtenemos una Gráfica de tiempos por cada procesador donde se puede observar como empiezan su ejecución en paralelo y como finalizan todos los procesos, en este caso solo mostraremos el de 20 procesadores.

## 5.2 Encontrar los primos que existen en un rango

El siguiente ejemplo tiene relación con el anterior, vamos a implementar una aplicación para encontrar los primos que se encuentren dentro de un rango definido mediante la Criba de Eratóstenes.

En este método se toma un número el cual se compara con los anteriores para verificar si es o no primo. Se imprimirá el número que sea primo en pantalla

Como en cada aplicación empezamos con el archivo de cabeceras donde declaramos las variables y métodos a utilizar, la clase main tendrá un método llamado finalizar que hará que la aplicación termine una vez que todas las tareas se hayan ejecutado.

```
#include "criba.decl.h"

class main : public Chare
{
    private:
        long double count, inicio, final;
        int faltan, total;

    public:
        main(CkMigrateMessage *m) {}
        main(CkArgMsg *m);
        void finalizar(int cnt);
};
```

En este ejemplo en lugar de utilizar arreglos utilizaremos un grupo que será *primos* con un método buscar que tendrá como parámetros un rango [a,b] para buscar que números son primos.

```
class primos : public CBase_primos
{
    public:
        primos(CkMigrateMessage *m) {}
        primos(void);
        void buscar(long double a, long double b);
};
```

En el archivo de interfaz se declaran los métodos de entrada que se utilizarán en la aplicación así como los identificadores de algún chare.

```
mainmodule criba {

readonly CkChareID mainhandle;
mainchare main{
    entry main();
    entry void finalizar(int cnt);
};
```

Aquí se declara el grupo de chares que se repartirán sobre cada procesador disponible para realizar tareas. Para designarlo se antepone la palabra **group** al nombre del grupo. Tendremos un método **buscar** que es el que se encargara de buscar en el rango [a,b] que número son primos y los enviara a pantalla.

```
group primos
{
    entry primos(void);
    entry void buscar(long double a, long double b);
};

};
```

En el código fuente vamos a definir unas constantes **FALSO** y **VERDADERO** como 0 y 1 respectivamente los cuales utilizaremos para definir cuando hemos encontrado un número primo en el método buscar.

```
// Encontrar los primos en un rango mediante la Criba de Eratóstenes
#include "criba.h"
#include "math.h"

CkChareID mainhandle;

#define VERDADERO 1
#define FALSO 0

main::main(CkArgMsg * m)
{
    int nc,i;
    long double aux1, aux2, proc;
    long double ini, fin;
    nc=CkNumPes();
    faltan = nc;
    total = 0;
```

Tomaremos los argumentos de límite inferior y superior en las variables **ini** y **fin**, una vez mas tomamos el tiempo en que comienzan los cálculos. En la variable **proc** colocaremos el tamaño para cada partición, es decir, los rangos que tendrá cada procesador que utilizar.

```
if(m->argc != 3)
{
```

```

    CkPrintf("Deben ser 2 parámetros, <limite inferior> <limite
              superior>\n");
    CkAbort("");
}
ini = atoi(m->argv[1]); // limite inferior
fin = atoi(m->argv[2]); // limite superior
delete m;

CkPrintf("procesadores a utilizar = %d \n",nc);
inicio = CmiWallTimer(); //tomamos el tiempo en que se inicia

proc = ceil((fin - ini) / nc);

```

La creación del grupo *eratostenes* es similar a la creación de un arreglo con la variante que no tendrá que colocarse el tamaño por que será creado exactamente un chare sobre cada procesador, solamente si el constructor necesitara parámetros adicionales los colocaremos, en este caso no será así.

```

// se crea un grupo eratostenes de tipo Cproxy_primos
// para que cada procesador tenga un objeto chare
CProxy_primos eratostenes = CProxy_primos::ckNew();

aux1 = ini;
aux2 = aux1 + proc;

```

En esta parte se enviarán a ejecutar los procesos que buscarán en el rango los números que primos, la variable *aux2* la ajustaremos al final del rango para no realizar operaciones adicionales.

```

for (i=0; i<nc; i++){

    //se envía el método buscar para cada procesador
    eratostenes[i].buscar(aux1,aux2);
    aux1 += proc;

    if ((i+1) == nc )
    {
        aux2 += fin;
    }
    else
    {
        aux2 += proc;
    }
}

mainhandle = thishandle; // se obtiene el handle del main
}

```

El método **finalizar** estará recibiendo información del método **buscar** para saber cuando debe finalizar todos los cálculos enviando a pantalla la información acerca del tiempo que tomaron todos los cálculos.

```
void main::finalizar(int cnt)
{
    long double tiempo;

    faltan--;
    total+=cnt;
    if (faltan==0){
        final = CmiWallTimer();
        tiempo = (final-inicio);
        CkPrintf("Total de primos = %d encontrados\n", total);
        CkPrintf("Finalización de los cálculos en %Lf segundos\n",tiempo);
        CkExit();
    }
}
```

Como habíamos visto en los ejemplos anteriores es el constructor en este caso del grupo, la sintaxis no varía en lo que respecta a los arreglos.

```
//constructor no se realiza nada a si que se queda en blanco
primos::primos() { //constructor }
```

El método **buscar** es donde en un ciclo desde un límite inferior **a** hasta uno superior **b** se revisarán todos los números de este rango para saber cuál es primo e imprimirlo en pantalla. Utilizaremos algunas variables auxiliares para hacer ciertos cálculos.

```
void primos::buscar(long double a, long double b)
{
    int esp, proc, cont = 0;
    long double i, num,aux, aux1, aux2, raiz_num;

    proc = CkMyPe();
```

Debido a que la función **mod** de C solo está disponible para enteros implementaremos la misma función para reales, con las variables **aux1** y **aux2** vamos a dividir entre 2 el número **num** a evaluar. En **aux1** redondeamos la cantidad al entero anterior y en **aux2** al entero siguiente y en **aux** tendremos una resta entre estas 2 variables si el módulo de **num** es cero la resta, es decir **aux**, será cero.

```
for (num=a; num<b; num++)
{
    aux1 = ceil(num/2);
    aux2 = floor(num/2);
    aux = aux1 - aux2;
```

```

        esp = FALSO;

        if ((num==2) || (num==3))
        {
            esp = VERDADERO;
        }
        else
        {
            if ((aux==0) || (num==1))
            {
                esp = FALSO;
            }
        }
    }
}

```

Primero usaremos una variable **esp** de tipo booleano (anteriormente designamos los valores **FALSO** y **VERDADERO**) que será verdadera hasta que sea encontrado algún factor para **num**. Inicializaremos la búsqueda en 3 por que ya hemos evaluado si es par en el paso anterior al dividir por 2, y obtenemos el entero siguiente a la raíz de **num** para saber cuando terminar de realizar los cálculos.

De la misma forma que dividir entre 2 ahora con las variables **aux1** y **aux2** vamos a dividir entre los demás números anteriores a **num** para evaluar si es divisible o no por alguno de estos.

Al igual que el caso anterior **aux1** será redondeado la entero anterior y en **aux2** al entero siguiente, **aux** será la resta de las 2 variables y nuevamente si el modulo de **num** es cero la resta, será cero entonces ese número será cero, finalizamos el ciclo actual y tomamos el siguiente número.

```

        else
        {

            esp = VERDADERO;
            i = 3;
            raiz_num = floor(sqrt(num));

            do
            {

                aux1 = ceil(num/i);
                aux2 = floor(num/i);
                aux = aux1 - aux2;

                if (aux==0)
                {
                    esp = FALSO;
                    break;
                }
            }
        }
    }
}

```

El ciclo se realizara hasta llegar a la raíz de **num** por que es el número máximo que alcanzaría algún factor de **num**, después de la raíz el numero tendrá que ser mayor a num.

```

        i+=2;
    }while (i <= raiz_num);

    }
}

```

Si el valor de **esp** es verdadero se imprime el número y en que procesador fue encontrado y un contador **cont** para saber cuantos números encontramos.

```

        if (esp==VERDADERO)
        {
            CkPrintf("Primo encontrado = %.0Lf en el procesador
            %d\n",num, proc );
            cont++;
        }

    }

    CkPrintf("Total de primos = %d encontrados en el procesador
    %d\n",cont,proc);

    CProxy_main main(mainhandle);
    main.finalizar(cont);

}

#include "criba.def.h"

```

Igual que en el ejemplo anterior tenemos el **Makefile** para la compilación, ejecución y limpieza de la aplicación. Iniciamos con la ubicación del compilador de Charm++

```
CHARMC= ../../charm-6.0/net-linux-amd64/bin/charmc
```

nuevamente tenemos la compilación del programa usando **charmc** y los parámetros **-language charm++** para definir el lenguaje, **-tracemode projections** para utilizar obtener información acerca de la ejecución usando **projections** y **-tracemode summary** para obtener las gráficas del seguimiento.

```

all: criba
criba: criba.o
    $(CHARMC) criba.o -o criba -language charm++ -tracemode
    projections -tracemode summary

```

La orden **run** para ejecutar con **charmrun** los procesadores que usaremos representado **+p10**. Los siguientes 2 números son el limite inferior y superior donde buscaremos números primos y por ultimo nuevamente tenemos el archivo **odelist**, de este archivo ya no vamos a hablar.

```

run: criba
    ./charmrun +p1 criba 100000000000 100001000000 ++nodelist nodelist

criba.o : criba.C criba.h criba.decl.h
    $(CHARMC) -c criba.C

criba.decl.h: criba.ci
    $(CHARMC) criba.ci

```

En esta parte eliminaremos los archivos generados automáticamente al momento de la compilación si realizamos algún cambio y nos genera algún error tener alguno de estos archivos

```

clean:
    rm -f criba *.o conv-host *.decl.h *.def.h *.sts *.log *.sum ~~ charmrun

```

A continuación presentamos la tabla con los resultados y la gráfica del comportamiento respecto al tiempo de ejecución, notaremos aquí que en el caso de llegar a 20 procesadores el tiempo comienza a aumentar esto es el tipo de cálculos llega a un punto donde utilizar paralelismo no mejora el rendimiento de la aplicación. La causa también puede ser que uno de los procesadores es el que esta encargado del sistema completo del cluster y al estar atento a sus servicios no sea posible utilizarlo al máximo.

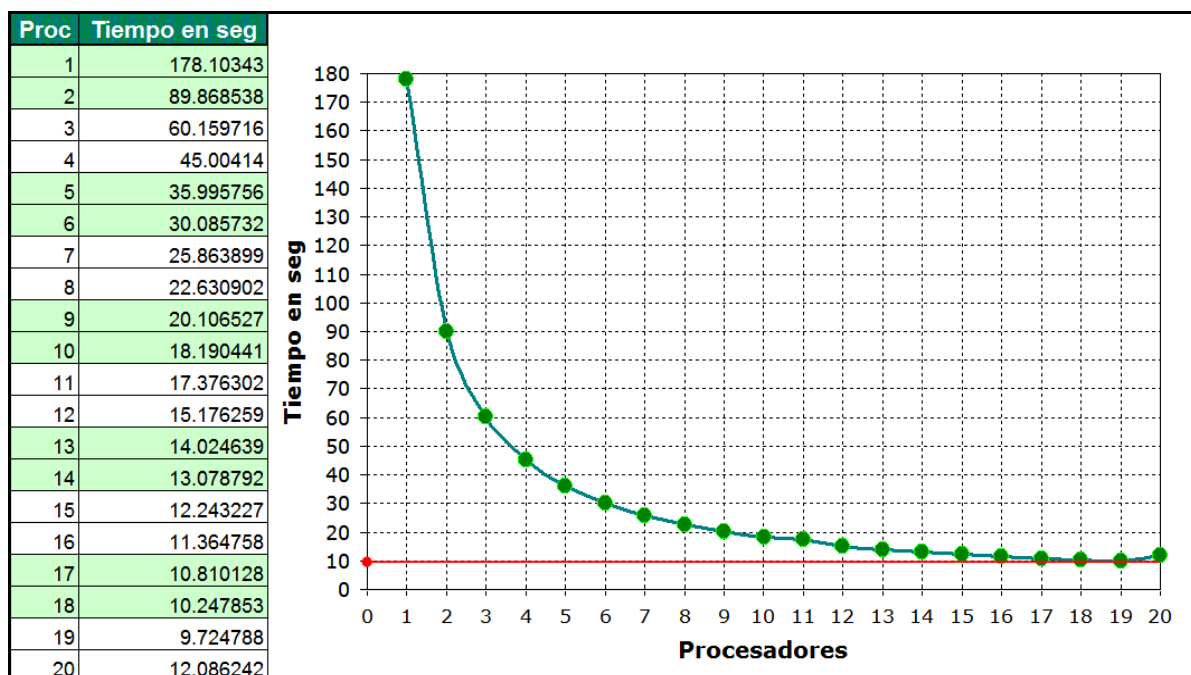


Tabla 4 Tabla de tiempos y gráfica comparativa tiempo vs. Procesadores

Presentamos ahora las gráficas obtenidas por *Projections* también vemos un comportamiento continuo sin variación en el uso del procesador.

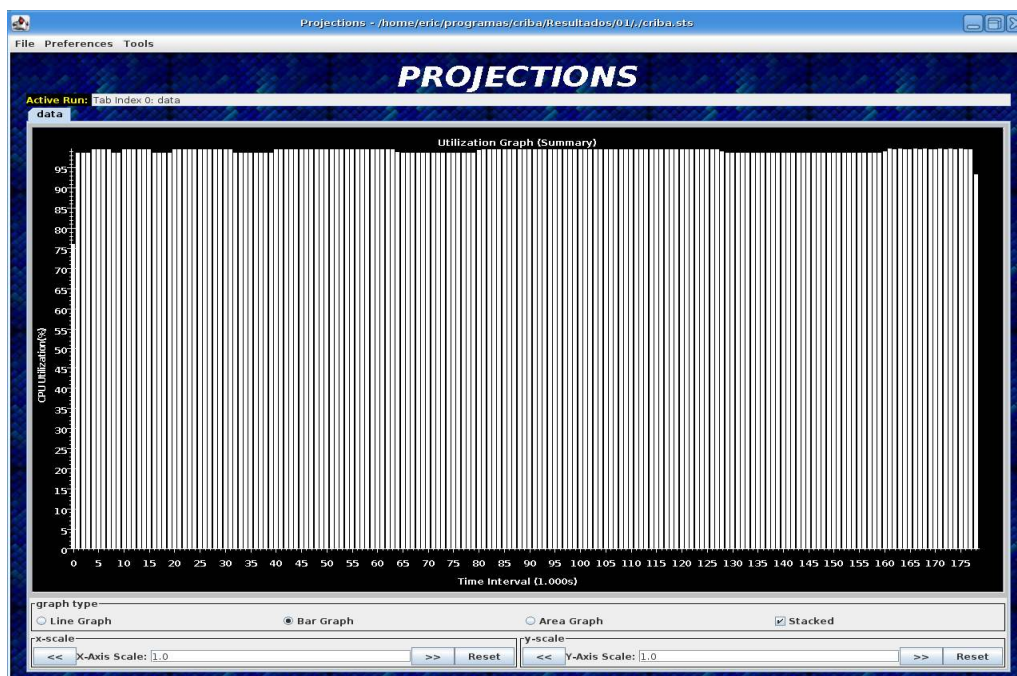


Fig. 5.22 Gráfica de procesamiento con 1 procesador

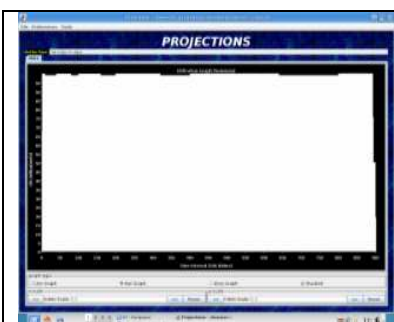


Fig. 5.23 Gráfica con 2 procesadores



Fig. 5.24 Gráfica con 3 procesadores



Fig. 5.25 Gráfica con 4 procesadores

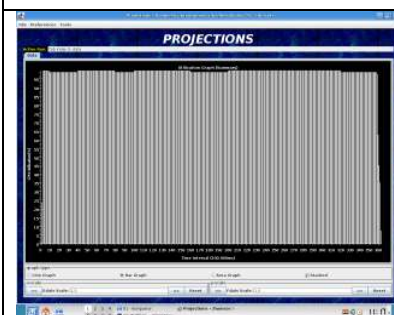


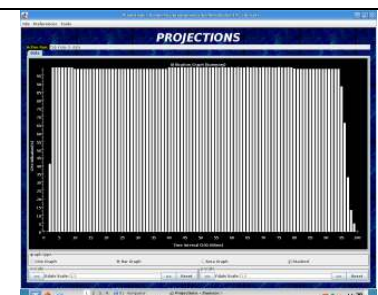
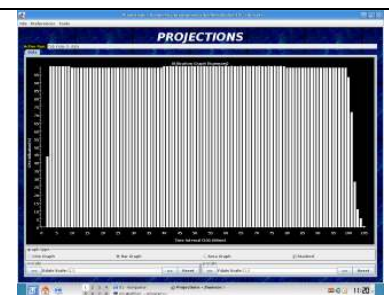
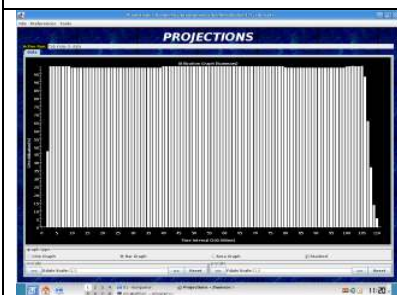
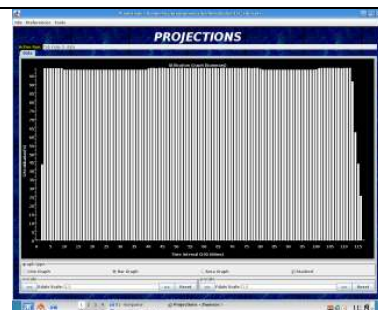
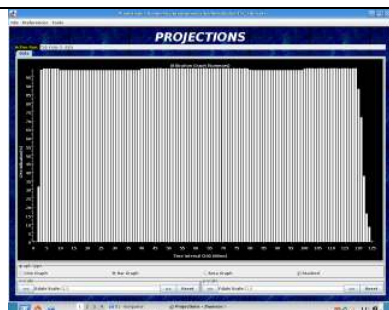
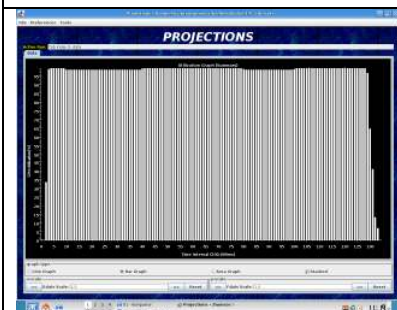
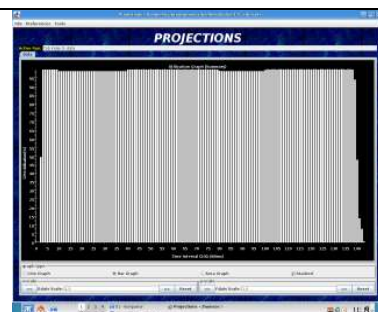
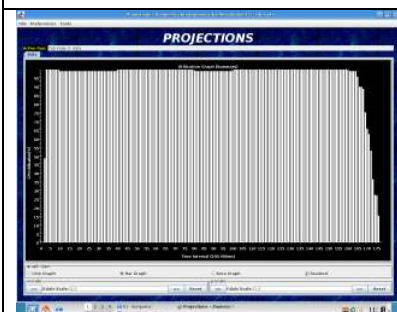
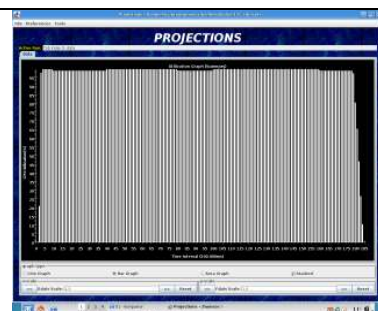
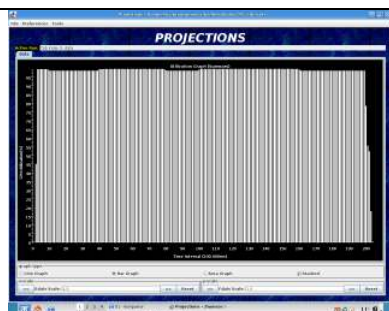
Fig. 5.26 Gráfica con 5 procesadores



Fig. 5.27 Gráfica con 6 procesadores



Fig. 5.28 Gráfica con 7 procesadores



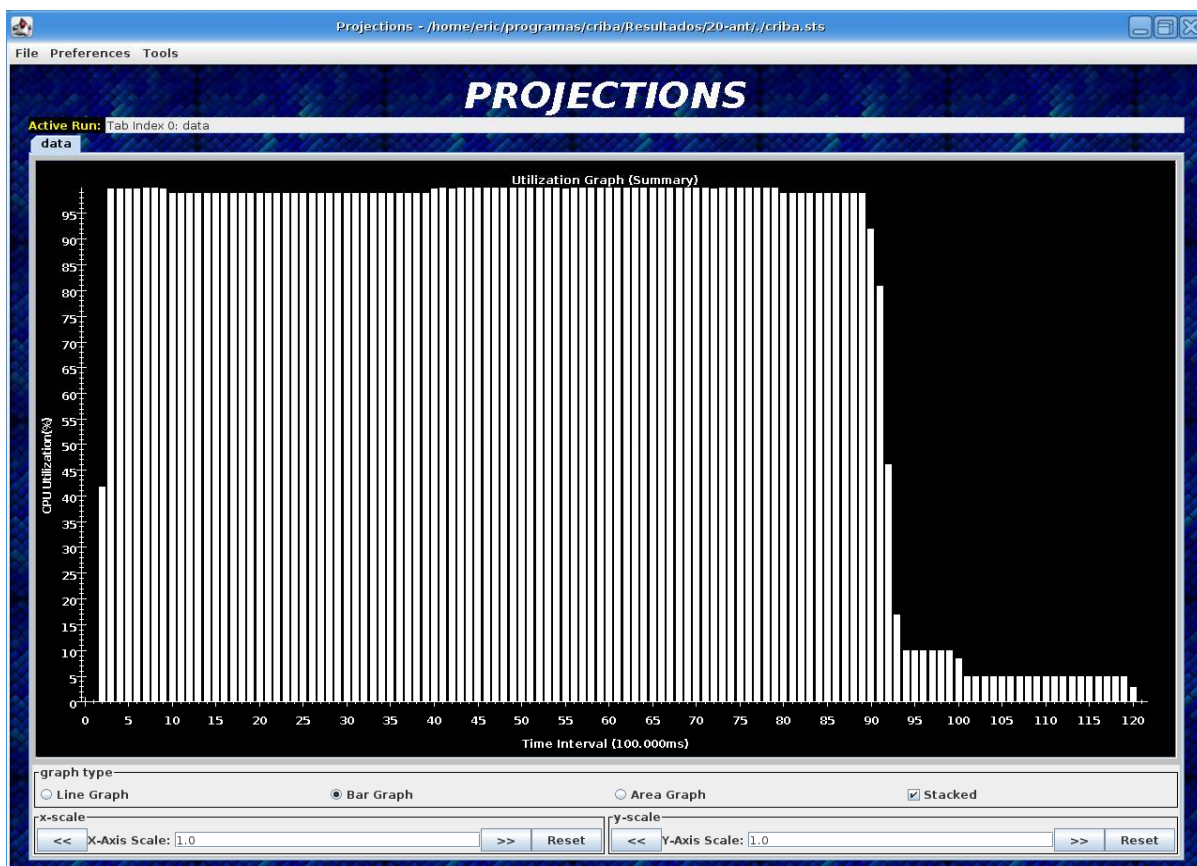


Fig. 5.41 Carga de los 20 procesadores en la aplicación

En esta última gráfica se muestra la carga en 20 procesadores y visualizamos que después de 9 segundos aproximadamente la carga ya no está tan equilibrada, y como veremos en la siguiente gráfica el segundo procesador que corresponde con el nodo maestro es el que tarda más en procesar la tarea que le fue asignada. Los demás procesadores muestran una carga similar. Como habíamos mencionado quizá el ser el nodo maestro no se pudo hacer el uso del 100% del procesador.

**Projections** también tiene otra herramienta de visualización que permite revisar las llamadas a funciones que se ejecutan sobre cada procesador. En estas gráficas se muestran los procesos que se llevan a cabo en cada procesador cuando usamos 20 procesadores, en este caso solo hay un proceso cada uno.

También nos muestra el momento en que las llamadas hacia los métodos de entrada son realizadas al final del primer procesador no se ve con claridad pero existe una línea desde el procesador 0 en el inicio y también en el final de todos los procesos, en color amarillo representa la función *buscar* y en color blanco se tenemos el tiempo procesador está sin realizar tarea alguna. Al final del procesador cero no se logra apreciar pero está la llamada a la función *finalizar* que envía un mensaje a todos los procesadores para terminar los procesos pendientes.

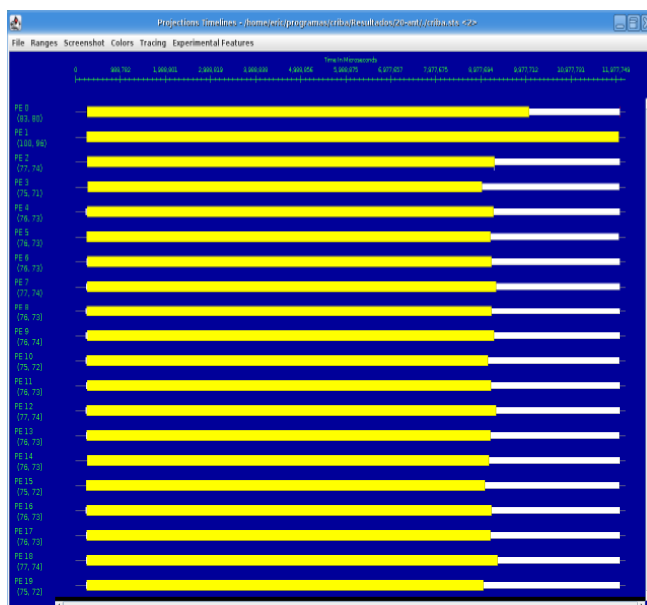


Fig. 5.42 La carga de todos los procesadores

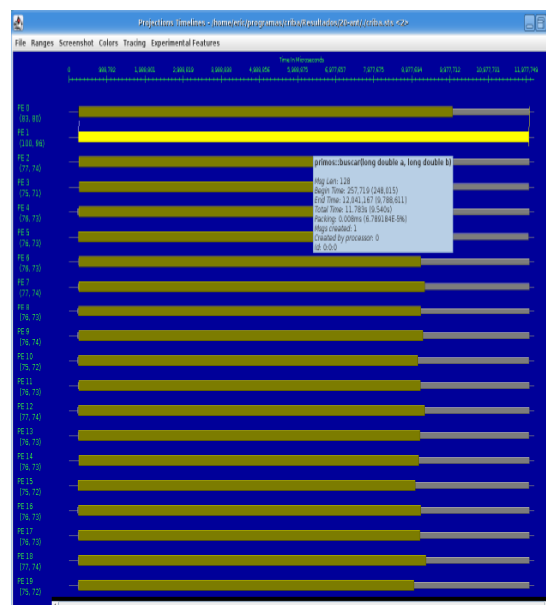


Fig. 5.43 La carga del segundo procesador tarda más su ejecución.

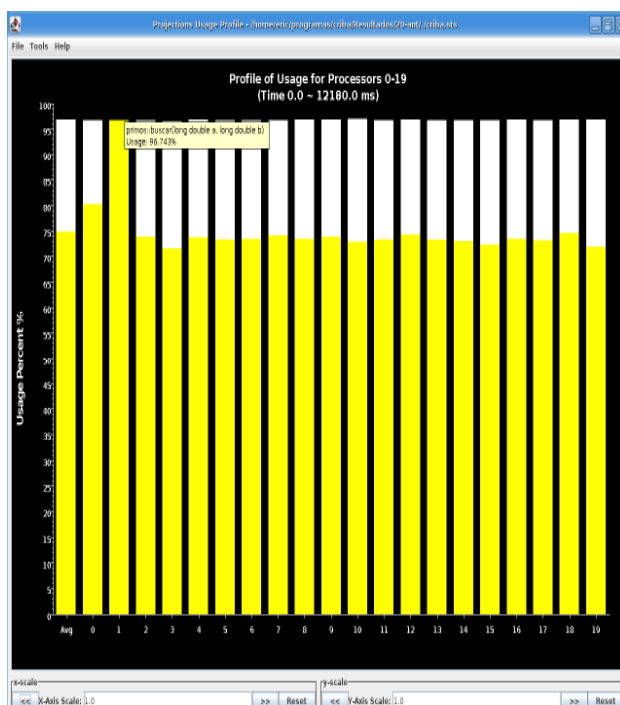


Fig. 5.44 Carga de cada procesador

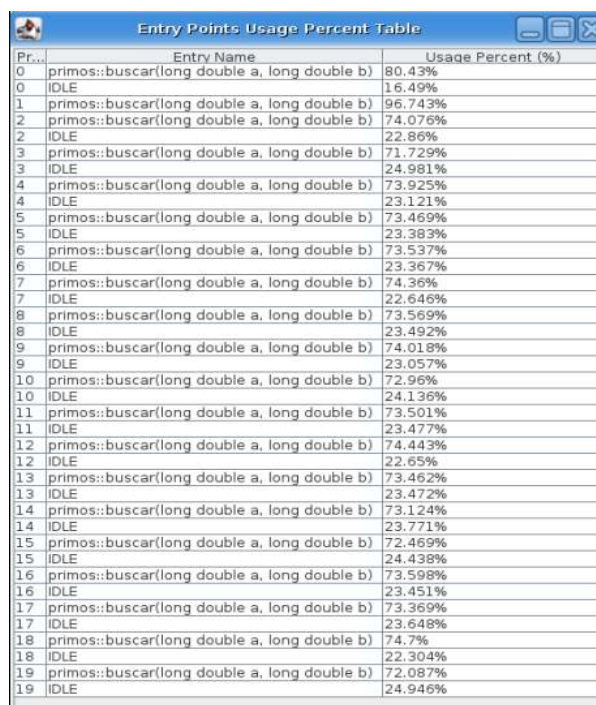


Fig. 5.45 Porcentajes de carga en cada procesador de una función que se ejecuta

En las gráficas anteriores tenemos uso del CPU en cada procesador en porcentajes, estos porcentajes están relacionados con el total de procesamiento total. En la primera columna tenemos un promedio de todos los procesadores, después van apareciendo desde el

procesador 0 hasta el 20 y si comparamos con la gráfica anterior es similar. A la derecha tenemos los datos expresados numéricamente y que porcentaje corresponde al proceso en este caso a *primos::buscar* y el porcentaje donde el procesador se mantiene sin trabajo.

Otra herramienta de visualización que presenta *projections* es una animación donde se muestra como va aumentando o disminuyendo la carga sobre un procesador, a diferencia de las gráficas anteriores en este caso si muestra lo que se esta utilizando en términos de recurso de procesamiento individual en cada procesador y también como se van liberando hasta quedar libre de trabajo por parte de la aplicación, en la parte de abajo muestra una barra para comparar dependiendo del color que porcentaje del procesador se utiliza. Como la carga y la liberación de recursos son casi inmediatas en la animación se muestra poco detalle cuando finalizan los procesos.

Se tienen controles para definir si la animación va más rápida o mas lenta, en todas las gráficas que hemos visto es posible analizar un lapso en específico en la ejecución de la aplicación.

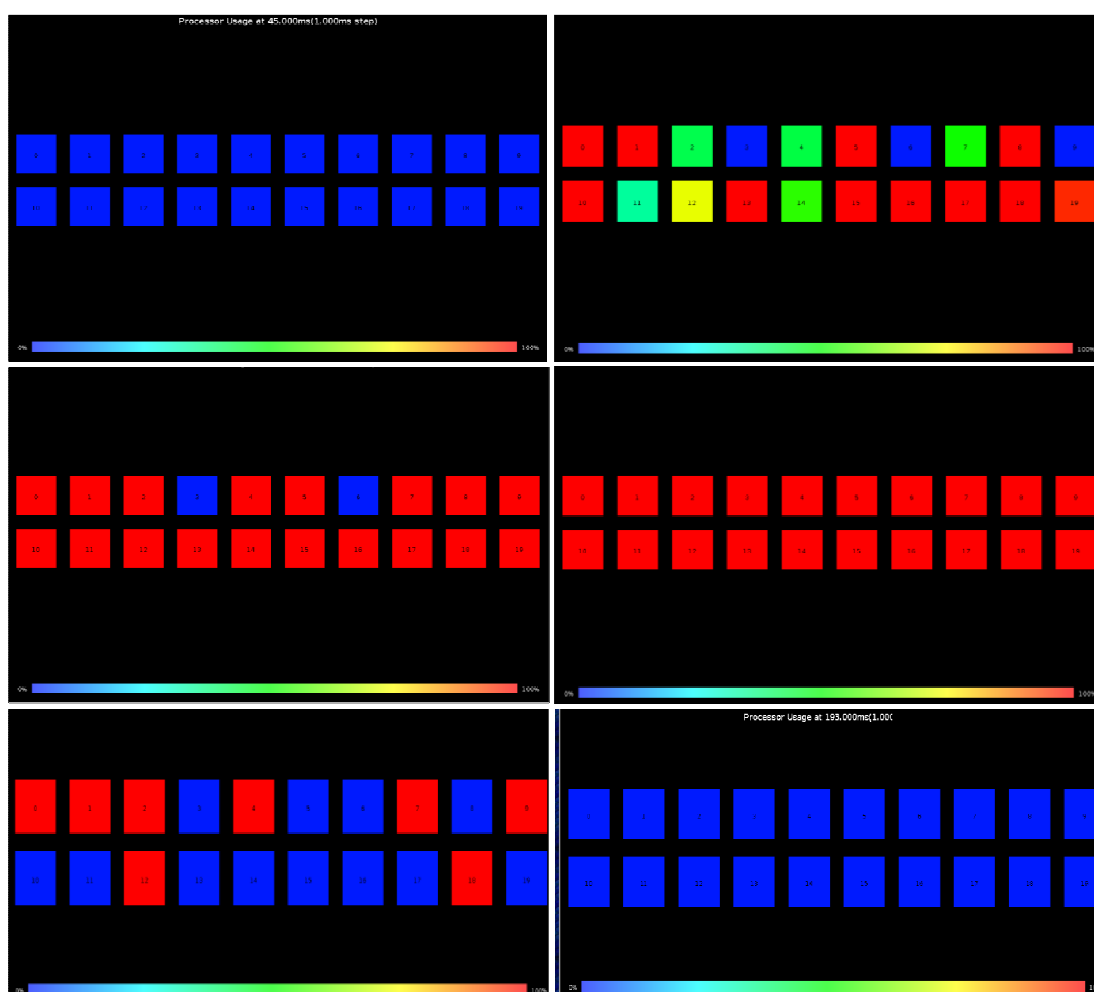


Fig. 5.46 Animación de la carga de trabajo en los procesadores al ejecutar la aplicación

## 5.3 Factorizar un número con el método de Fermat

En el mundo de la criptografía se han utilizado varios métodos para lograr ocultar información, uno de los más utilizados en la actualidad es la encriptación RSA el utiliza una llave privada y una llave pública para cifrar y descifrar mensajes.

En 1977 Ron Rivest, Adi Shamir y Len Adleman propusieron un sistema de encriptación utilizando una clave pública y una privada que posteriormente se denominó el problema RSA, basado en la factorización de enteros grandes.

El método RSA selecciona al azar 2 números primos muy grandes y calcula su producto:  $n=pq$  entonces se obtiene otro número  $\phi(n) = (p-1)(q-1)$ , a continuación se elige un número  $1 < e < \phi(n)$  tal que  $e \equiv 1 \pmod{\phi(n)}$ , o lo que sería lo mismo encontrar un número  $e$  tal que el máximo común divisor entre  $e$  y  $\phi(n)$  sea 1, dicho número existe y además es único, de esta forma obtenemos la clave pública  $P(e, n)$ . Además se calcula una llave privada que es  $d = \text{inv}[e, \phi(n)]$  el cual es el modulo inverso de  $e$  con  $\phi(n)$ .

Ahora con la clave pública  $P(e, n)$  y la clave privada  $d$ . La función de cifrado se aplica de la siguiente forma

$$\text{encriptar}(m) = m^e \pmod{n}$$

Donde  $m$  es el mensaje a cifrar. La función de descifrado es:

$$\text{desencriptar}(c) = c^d \pmod{n}$$

Donde  $c$  es el mensaje cifrado. Por ejemplo si tenemos los siguientes datos

$$\begin{aligned} p &= 61 \\ q &= 53 \\ n &= pq = (61)(53) = 3,233 \\ \phi(n) &= (61-1)(53-1) = 3,180 \\ e &= 17 \\ d &= 2753 \end{aligned}$$

Para cifrar el mensaje 123, debemos calcular:

$$\text{encriptar}(123) = 123^{17} \pmod{3233} = 855$$

Y para descifrar el mensaje codificado, realizamos:

$$\text{desencriptar}(855) = 855^{2753} \pmod{3233} = 123$$

Ambos cálculos pueden realizarse de forma eficiente usando el algoritmo de multiplicación cuadrática para exponenciación modular o el teorema del resto chino.

La fortaleza de este método se basa en la factorización de números grandes de 512 bits en adelante. Actualmente se considera una encriptación fuerte usar números arriba de 1024 bits ya que el avance en la potencia de los equipos de computo han logrado factorizar varios números utilizados por el método RSA.

En este ejemplo utilizaremos el método de Fermat para encontrar factores primos de números grandes. En este caso utilizaremos hasta el valor máximo de 16 cifras. Pero los métodos utilizan números de 100 a 150 cifras.

Lo que realizaremos será un programa que implemente la factorización mediante el teorema de Fermat. Para tener una noción de lo que significa factorizar un número grande tenemos la siguiente tabla donde aparecen el total de operaciones que se tendrían que realizar para factorizar un número en sus factores primos.

| No. De Bits | No. De Dígitos | No. Instrucciones    |
|-------------|----------------|----------------------|
| 166         | 50             | $1.6 \times 10^{10}$ |
| 332         | 100            | $3.7 \times 10^{14}$ |
| 512         | 155            | $9.5 \times 10^{17}$ |
| 664         | 200            | $2.2 \times 10^{20}$ |
| 1024        | 308            | $6.0 \times 10^{24}$ |
| 1664        | 500            | $9.0 \times 10^{30}$ |
| 3322        | 1000           | $6.0 \times 10^{41}$ |

**Tabla 5** Tabla comparativa de número de instrucciones en comparación de la longitud del número a factorizar

Usando la **Roadrunner**, la supercomputadora más rápida del mundo que tiene una potencia de 1,026 petaFLOPS, es decir,  $1.026 \times 10^{15}$ , para factorizar un número de 50 dígitos tomaría 15 milisegundos, de 100 dígitos 3 centésimas de segundo, de 155 dígitos 15 minutos. Para factorizar un número de 1024 bits que es el que se recomienda como mínimo la **Roadrunner** tardaría 188 años en poder factorizar un número de esta longitud que equivale a 308 dígitos.

### 5.3.8 Método de Fermat

El método que utilizaremos para la factorización es el de Fermat, cabe aclarar que no es el mas eficiente ya que tiene problemas de tiempo al tratar de encontrar factores que están muy alejados entre si.

Partimos de lo siguiente, si  $n$  es igual a  $x^2 - y^2$ , entonces  $n$  puede factorizarse de una forma muy sencilla:  $n = (x+y)(x-y)$ . Por definición  $x^2$  es mayor que  $n$  entonces se tiene que  $x$  es mayor a la raíz cuadrada de  $n$ .

Ahora el método es el siguiente: sea  $n$  un número entero positivo que deseamos factorizar, tomaremos otro entero  $x$  positivo mayor que la raíz cuadrada de  $n$ . Posteriormente calculamos el valor de  $x^2$  y restamos  $n$ , Si el número obtenido  $y$  es un cuadrado perfecto, es decir, un número cuya raíz cuadrada no tenga residuo habremos terminado, sino es así tomamos ahora  $x+1$  y calculamos  $(x+1)^2$ , nuevamente restamos  $n$  y si obtenemos un número  $y$  que sea cuadrado perfecto finalizamos. Procedemos de la misma forma hasta encontrar un cuadrado perfecto.

Una vez encontrados  $x$  e  $y$ , de la primer premisa tenemos que  $n = x^2 - y^2$  lo que es igual a  $n = (x+y)(x-y)$  entonces los factores primos serían  $p = (x+y)$  y  $q = (x-y)$ .

A diferencia del ejemplo anterior aquí no vamos a definir el archivo de cabeceras, las declaraciones correspondientes a este archivo (las variables y métodos a utilizar) las haremos dentro del mismo código fuente. De este modo comenzaremos con el archivo interfaz donde declaramos los métodos de entradas y las variables que correspondan a funciones de **Charm++** en este caso 2 variable de tipo **CProxy\_nombre** que los utilizaremos como punteros hacia los objetos que vamos a crear. Para que sean visibles en todos los objetos los colocamos al principio de cualquier **chare** y le anteponemos la palabra reservada **readonly** debido a que no será modificada.

### El archivo de interfaces (.ci)

Como vimos en el ejemplo anterior aquí declaramos los métodos de entrada para los el **chare main** y el arreglo de chares factores. Encontramos 2 funciones en main **encontrados** y **no\_encontrados** que tienen un atributo **[expedited]** que nos ayudara a ejecutar estas funciones lo más pronto posible, debido a que la ejecución es asíncrona no se asegura que se ejecute inmediatamente después del llamado.

```
mainmodule ataque {

    readonly CProxy_main mainProxy;
    readonly CProxy_factores fermat;

    mainchare main
    {
        entry main();
        entry [expedited] void encontrados(int proc, long
            double a, long double b);
        entry [expedited] void no_encontrados(int proc, int indice);
    };
};
```

En el arreglo de chares también usaremos un método de entrada con el atributo **[expedited]** que será el encargado de avisar que ya se han encontrado los factores para que los chares que aún estén en la búsqueda se detengan y finalicen.

```

array [1D] factores
{
    entry void factores (void);
    entry [expedited] void termina();
    entry void encuentra(long double a, long double b, long double c);
};

};

```

El código fuente donde se encuentra el funcionamiento de la aplicación, en esta ocasión también contendrá las declaraciones de los objetos que colocamos regularmente en el archivo de cabeceras (con extensión .h). Tenemos declarado el *mainProxy* y *fermat* como apuntadores a los objetos *main* y *factores*. En este punto vemos que es posible realizar las declaraciones de los objetos desde el código fuente vamos a usar unas variables globales que las declaramos como *private* y los métodos como *public*.

Se obtendrán como parámetros el número a factorizar junto con el total de procesadores que usaremos. Y tomaremos el tiempo desde que iniciemos los cálculos.

```

#include "ataque.decl.h"

CProxy_main mainProxy;
CProxy_factores fermat;

class main : public CBase_main {
private:
    long double N;
    unsigned long int e;
    double inicio, fin, tiempo;
    long double raizN, aux1, aux2;
    int np;

public:

    main(CkArgMsg* m)
    {
        int i;

        if(m->argc < 1){
            CkPrintf("Deben haber por lo menos un parámetro, numero a
            factorizar .\n");
            CkAbort("Faltan parámetros");
        }

        inicio = CmiWallTimer(); //tomamos el tiempo en que se inicia

```

Guardamos en variables el valor de  $N$  y borramos el mensaje  $m$ . Posteriormente calculamos la raíz cuadrada de  $N$  y la redondeamos al siguiente entero anterior con *floor()*, después desplegaremos en pantalla que información usamos  $N$  y la raíz cuadrada de  $N$ .

```

N = atof(m->argv[1]);
e = atoi(m->argv[2]);

```

```

delete(m);
np = CkNumPes();
raizN = floor(sqrt(N));
CkPrintf("Numero a factorizar: %Lf  \nRaiz del numero : %Lf
\n",N,raizN);

```

A continuación creamos el arreglo de chares tamaño *np* que es el total de procesadores que usaremos en la aplicación, usaremos como nombre el apuntador que definimos anteriormente *Fermat*, posteriormente enviaremos a ejecución el método *encuentra* de cada elemento del arreglo con rangos que iniciarán de la *raizN + 1* e irán avanzando hacia *2raizN*, *3raizN*, así sucesivamente hasta llegar a *N*. Por último asignamos a *mainProxy* el *Proxy* del objeto actual que es *main* mediante *thisProxy* para que los chares se puedan comunicar con *main* y su método encontrados.

```

CProxy_factores  fermat = CProxy_factores::ckNew(np);

aux1 = raizN + 1;
aux2 = raizN + raizN;

mainProxy = thisProxy;

for(i=0; i < np; i++){
    //lanza las partes a calcular los primos que forman N
    fermat[i].encuentra(aux1,aux2, N);

    aux1 += raizN;
    aux2 += raizN;
}
}

```

En el método *encontrados* se realizan las operaciones para obtener los primos ya que pasa como parámetros  $x^2$  e  $y^2$ , a continuación imprime en pantalla los factores, en que procesador y el tiempo que tomo encontrarlos. Por último hay una llamada *CkExit()* para finalizar los procesos pendientes.

```

void encontrados(int proc, long double a, long double b){

    long double x,y;
    x = a + b;
    y = a - b;

    fin = CmiWallTimer();
    tiempo = (fin-inicio);

    CkPrintf("Se encontraron el procesador : %d -- 1er. factor : %Lf --
2o. factor : %Lf \n",proc,x,y);
    CkPrintf("En %f segundos \n", tiempo);
    CkExit();
}

```

Después tenemos el método *no\_encontrados* donde vamos a enviar nuevamente la tarea si es que en el rango que designamos en un principio no encontramos ningún factor,

usemos un nuevo rango de búsqueda. Si llegamos al final de los rangos posibles, es decir, llegamos a  $N$  entonces finalizaremos la aplicación con un mensaje de la posibilidad que este número sea primo. Volvemos a lanzar el método *encuentra* usando la referencia *fermat* y el índice del elemento que termino que viene con la variable *indice*.

```
void no_encontrados(int proc, int indice){

    CkPrintf("No se encontraron factores en el procesador %d \n",proc);
    CkPrintf("Del elemento %d del arreglo, después de %f segundos
    \n",indice, tiempo);

    aux1 += raizN;

    if ((aux2+raizN) > N) {
        aux2 = N-1;
    } else{
        aux2 += raizN;
    }

    fermat[indice].encuentra(aux1,aux2, N);

    if ((aux1+raizN) > N){
        CkPrintf(" No se encontraron factores por lo que es posible que
        el numero %Lf sea primo\n",N);
        CkPrintf("Cálculos finalizados después de %f segundos \n",
        tiempo);
        CkExit();
    }
}

};
```

Ahora pasamos al objeto factores, primero tenemos una variable *esta* para el objeto que usaremos una vez encontrados los factores, después tenemos los métodos empezando por el constructor para el arreglo, aquí inicializamos la variable *esta* y asignamos a *fermat* el Proxy de este objeto que será visible por otros objetos en este caso *main*. También tenemos el constructor para migrar los objetos que tiene el parámetro *\*m* de tipo *CkMigrateMessage*.

```
class factores : public CBase_factores{

private:
    int esta;

public:

    factores(){
        esta = 0;
        fermat = thisProxy;
    }

}
```

```
factores(CkMigrateMessage *m) {
}
```

Ahora tenemos otro método **termina** el cual va a cambiar el valor de esta a 1 para indicar que hemos encontrado los factores.

```
void termina(){
    esta = 1;
    //CkPrintf("Desde el procesador %d \n", CkMyPe());
}
```

Aquí tenemos una función **se\_encontro** donde mostramos que se pueden crear funciones o métodos que no sean métodos de entrada, este simplemente verifica si la variable esta fue modificada respecto a su valor inicial.

```
int se_encontro(){
    if( esta == 0 )
        return 0;
    else
        return 1;
}
```

El método **encuentra** lo que realiza es: dentro un rango [a,b] se busca un número  $x$  que cumpla con la ecuación  $y^2 = x^2 - n$  donde la variable  $y$  debe ser un cuadrado perfecto, si no es así, se incrementa el valor de  $x$  y se vuelven a hacer los cálculos. Se continúa con las iteraciones hasta encontrar el valor que satisfaga la ecuación. Cuando es encontrado este número es enviado al objeto **main** mediante su método **encontrados** utilizando como referencia a **mainProxy** para enviar el resultado a pantalla y finalizar los procesos pendientes.

Existe un llamado a **fermat.termina()** es en este punto donde se realizara una llamada **broadcast** a todos los miembros del arreglo de **chare** para que cambien el valor de la variable **esta** a 1 y se puedan finalizar los cálculos que se ejecuten actualmente a partir de este llamado. Este si debe ser método de entrada a diferencia de la función **se\_encontro()**, ya que cada elemento del arreglo tiene su propia variable **esta**, fuera también una función o método solamente cambiará el valor del elemento del arreglo de **chares** desde el cual se hizo el llamado, los demás elementos permaneces con el mismo valor inicial.

```
void encuentra(long double a, long double b, long double c)
{
    int res, cont=0;
    long double i, Naux, x, y, aux=0;

    x = a;
    y = ((x*x) - c);

    res = CkMyPe();
}
```

```

CkPrintf("Procesador : %d----- LI: %Lf --- LS: %Lf --- N: %Lf \n",
res,a,b,c);

for (i=a; i<b; i++)
{

```

En esta parte es donde verificamos si ya fueron encontrados los factores para terminar con este proceso desde este punto.

```

    if (se_encontro() != 0)
    {
        CkExit();
        return;
    }

    aux =0;

    if (y>0){
        aux = ceil(sqrt(y));
    }

    Naux = aux*aux;

    if (y == Naux)
    {
        fermat.termina();
        CkPrintf("Procesador : %d ----- x: %Lf ----- y:
%Lf esta : %d\n", res,x,aux,esta);
        mainProxy.encontrados(res,x,aux);
        break;
    } else {
        x+=1;
        y = ((x*x) - c);
    }
}
mainProxy.no_encontrados(res,thisIndex);
}

};

#include "ataque.def.h"

```

Como en reales no existe la función **mod()**, para validar si un número es cuadrado perfecto se le extrae la raíz cuadrada y se redondea al siguiente entero con la función **ceil()**, si es cuadrado perfecto no se verá afectado por esta operación, depuse se utiliza un variable **aux** con este resultado, se eleva al cuadrado y se comparan **Naux** e **y**, si son iguales es un cuadrado perfecto si no continúa con las iteraciones.

A continuación presentamos las tablas de tiempos que tardo en encontrar la solución de los factores:

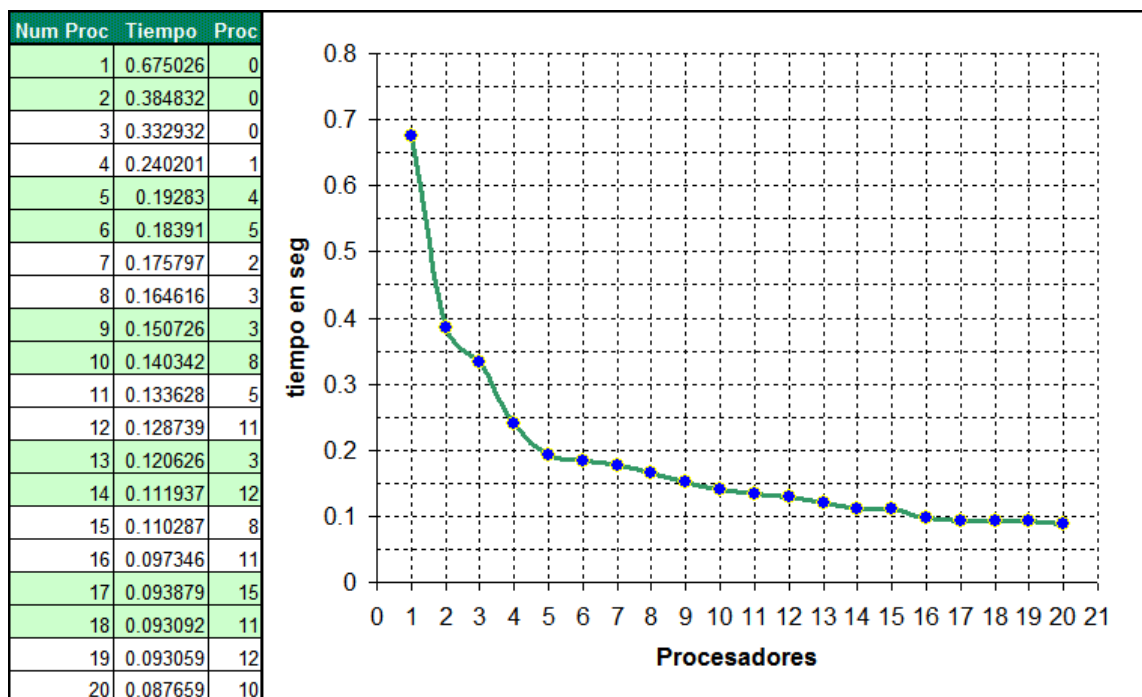


Tabla 6 Tabla de resultados y gráfica de tiempos de ejecución

Ahora tenemos las gráficas obtenemos del seguimiento que de *projections* a la aplicación respecto al uso de procesamiento en cada procesador.

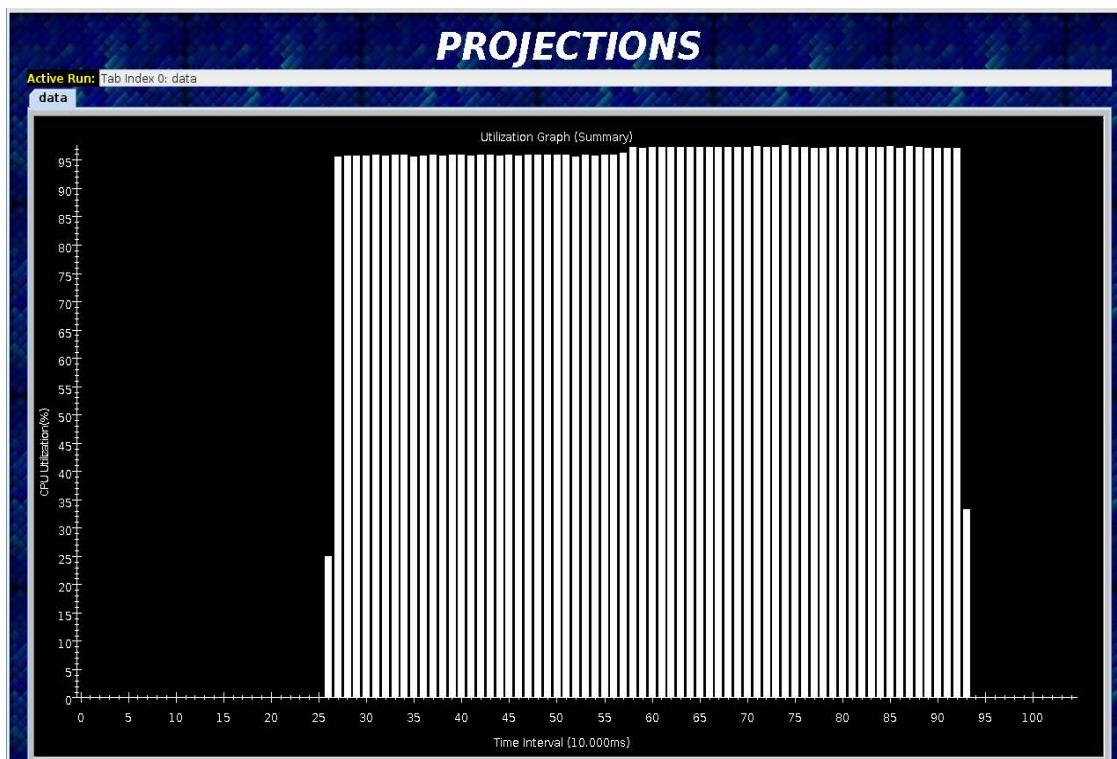


Fig. 5.47 Gráfica de utilización del CPU para 1 procesador

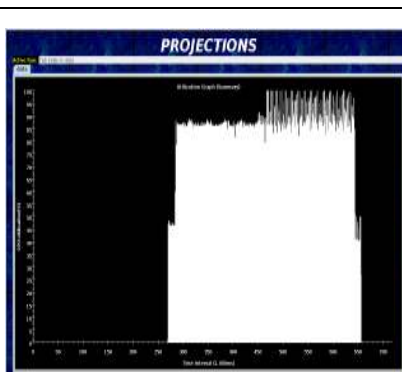


**Fig. 5.48** Gráfica para 2 procesadores

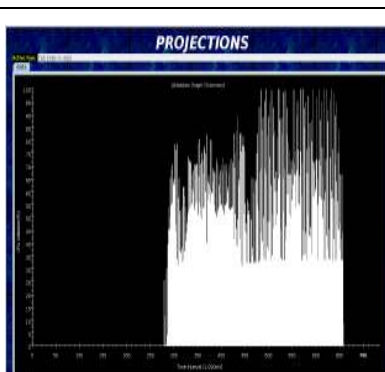


**Fig. 5.49** Gráfica para 3 procesadores

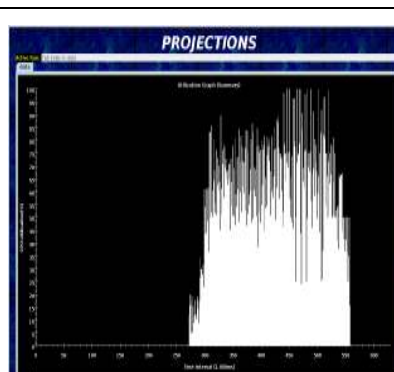


**Fig. 5.50** Gráfica para 4 procesadores

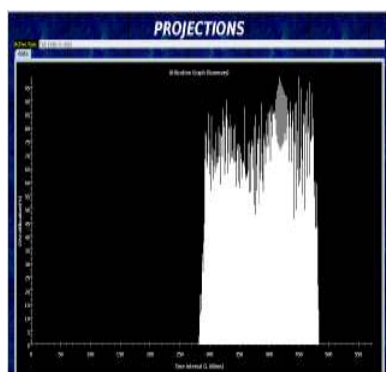


**Fig. 5.51** Gráfica para 5 procesadores

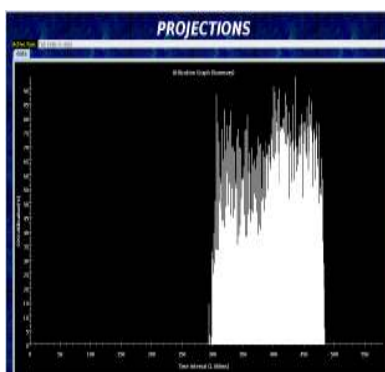


**Fig. 5.52** Gráfica para 6 procesadores

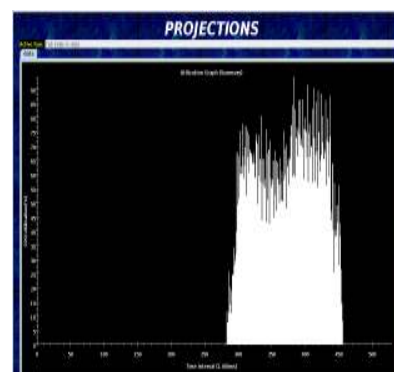


**Fig. 5.53** Gráfica para 7 procesadores

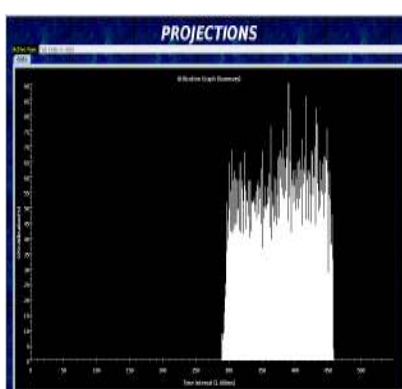


**Fig. 5.54** Gráfica para 8 procesadores

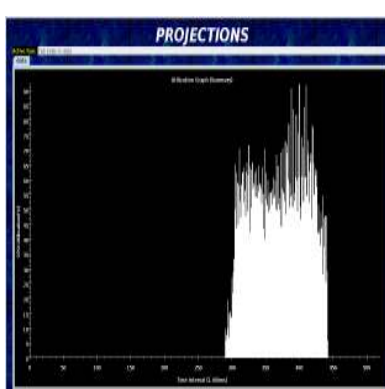


**Fig. 5.55** Gráfica para 9 procesadores

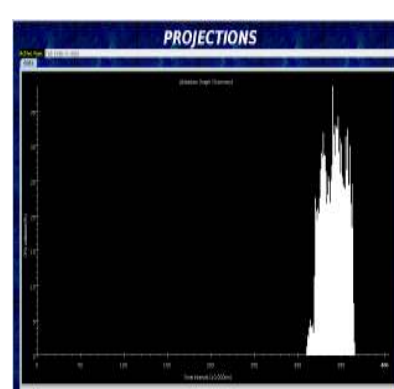


**Fig. 5.56** Gráfica para 10 procesadores

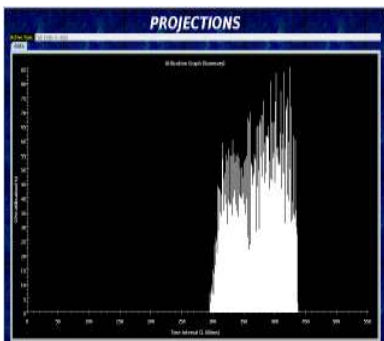


**Fig. 5.57** Gráfica para 11 procesadores

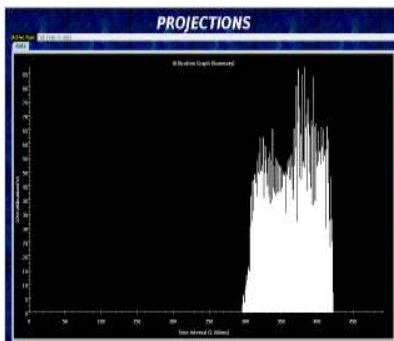


**Fig. 5.58** Gráfica para 12 procesadores

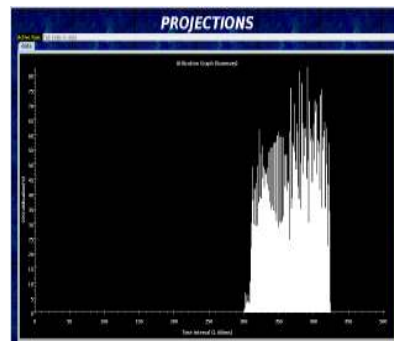


**Fig. 5.59** Gráfica para 13 procesadores

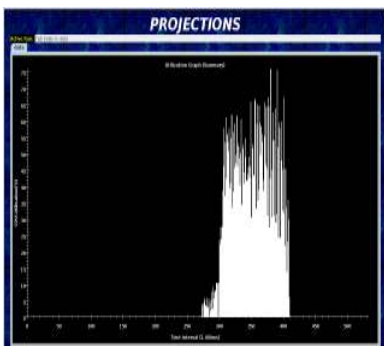


**Fig. 5.60** Gráfica para 14 procesadores

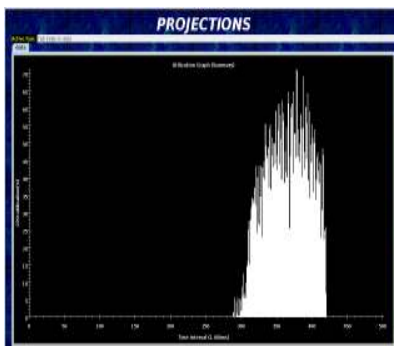


**Fig. 5.61** Gráfica para 15 procesadores

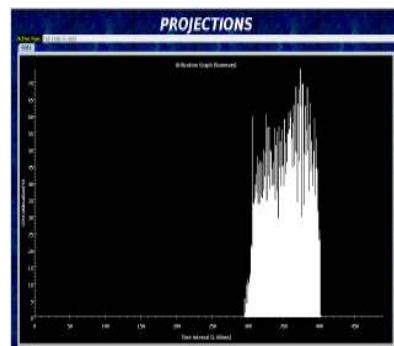


**Fig. 5.62** Gráfica para 16 procesadores

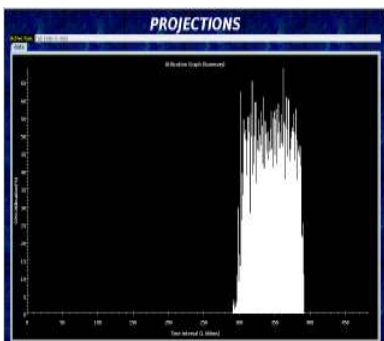


**Fig. 5.63** Gráfica para 17 procesadores

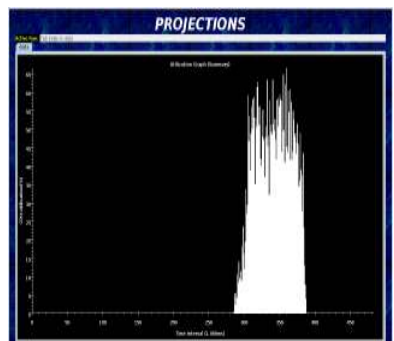


**Fig. 5.64** Gráfica para 18 procesadores

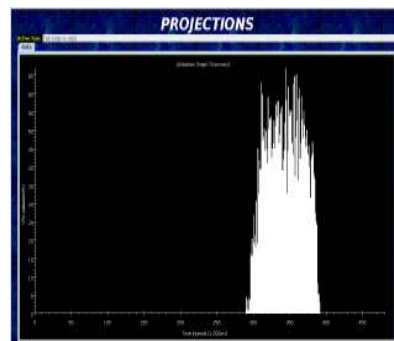


**Fig. 5.65** Gráfica para 19 procesadores

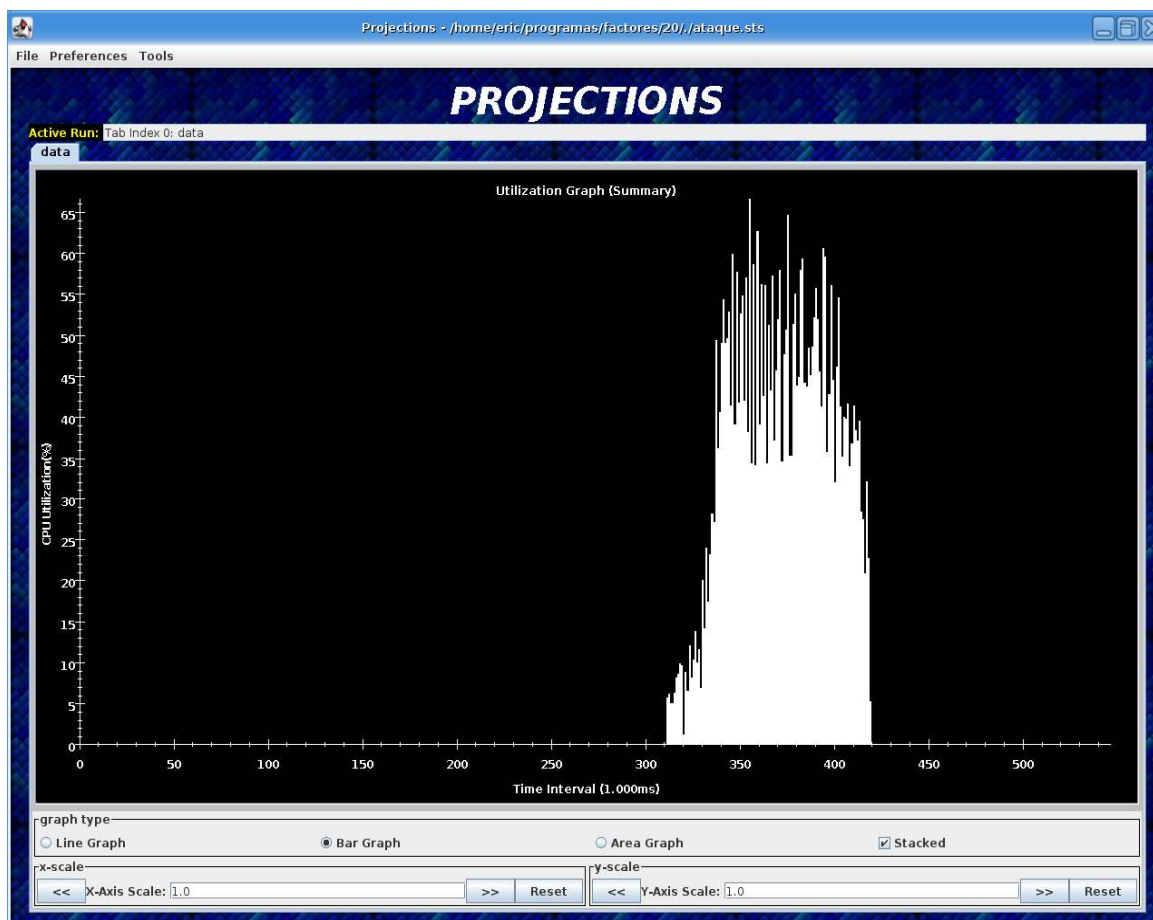


Fig. 5.66 Gráfica de procesamiento con 20 procesadores

Con este ejemplo es nos permite visualizar con más detalle la interacción entre procesadores, ya que se realizan llamados a funciones de objetos que se encuentran en otro procesador. En las siguientes gráficas presentamos las funciones y las llamadas que se realizan durante la ejecución, notamos que en el procesador cero aparecen varias partes en color verde, son las llamadas a la función *no\_encontrados* que corresponde al objeto *main* el cual siempre es creado en el primer procesador, este es el que se encarga de iniciar los procesos.

En la gráfica que le continúa ahora podemos ver mejor las llamadas que se realizan entre los procesadores, son las líneas que aparecen en color amarillo. Se observa que las líneas son más cercanas en el inicio y se van separando en el final, esto es por que los procesadores tienen que realizar más operaciones a medida que van creciendo los intervalos de búsqueda. En este tipo de gráfica se pueden ver las llamadas de un procesador a otro, el procesador 10 es el que se encuentra con mayor iluminación y es el que esta interactuando con el procesador 0.



Fig. 5.67 Gráfica de llamadas a funciones y comunicación.

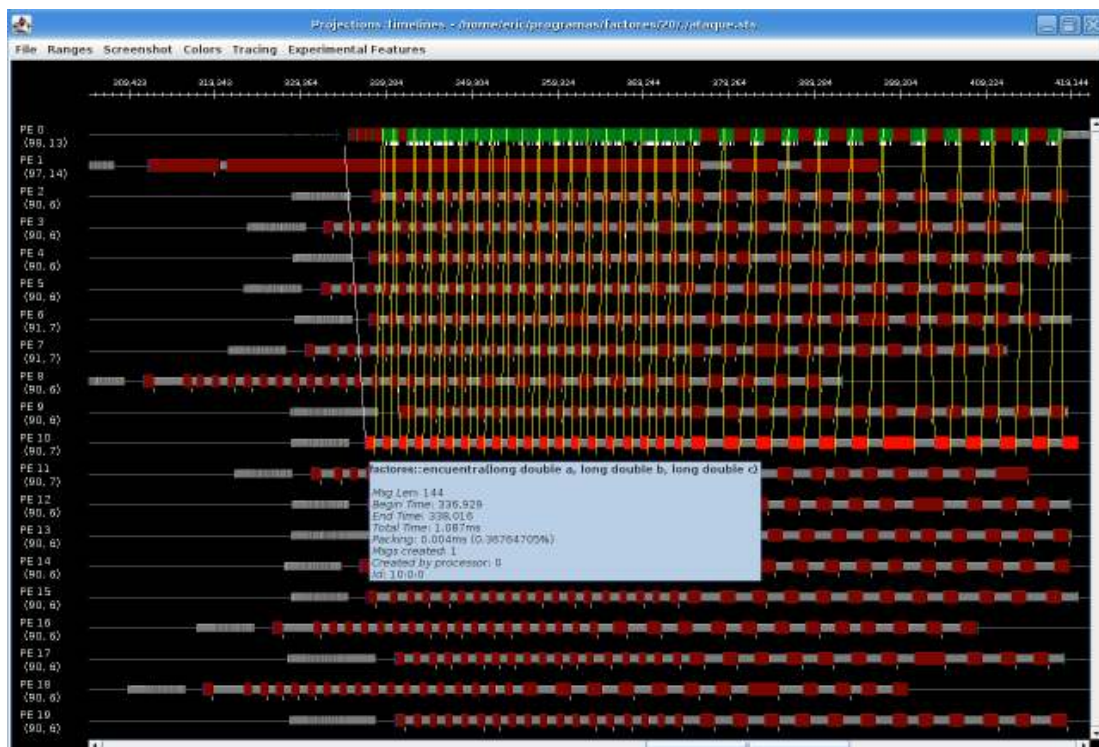


Fig. 5.68 Llamadas a funciones de chares entre procesadores

La próxima gráfica nos muestra cuando un procesador en este caso el número 12 encuentra los factores, entonces este envía dos mensajes al procesador 0 el primero de tipo **Broadcast** para que ejecutar la función **factores::termina()** la cual se encarga de cambiar la bandera **esta** a **1** para que se finalicen los procesos que se ejecuten en ese momento, y la otra llamada es a la función **main::encontrados()** con lo cual manda a pantalla los factores y finaliza hace un llamado a todos los procesadores a no ejecutar los procesos pendientes.

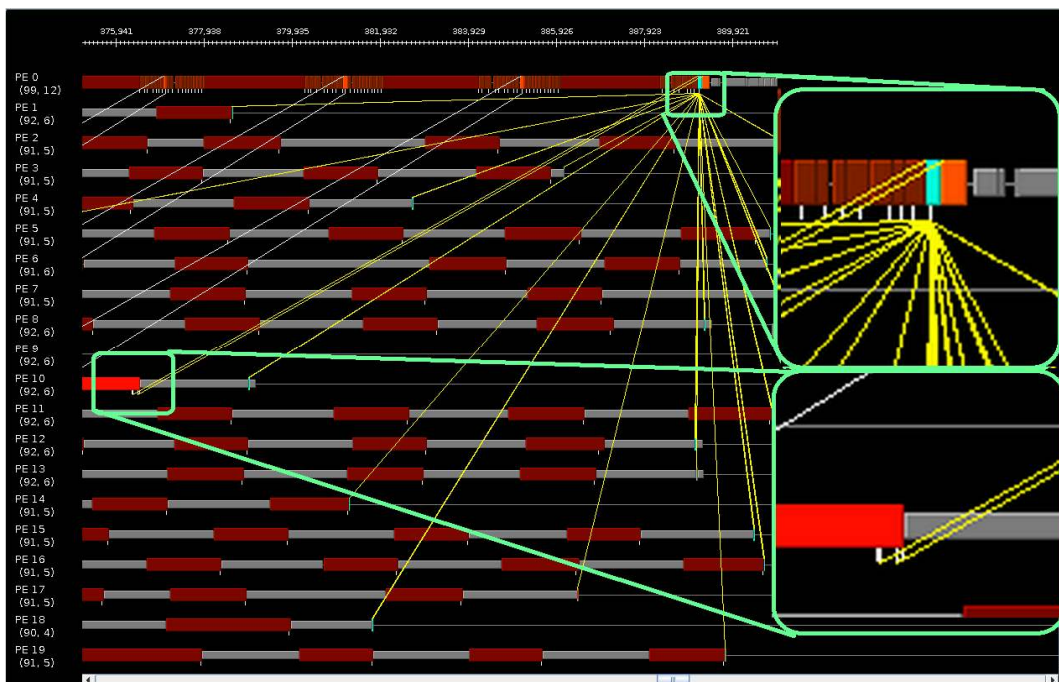


Fig. 5.69 Llamada tipo *Broadcast* **factores::termina()** hacia todos los procesadores

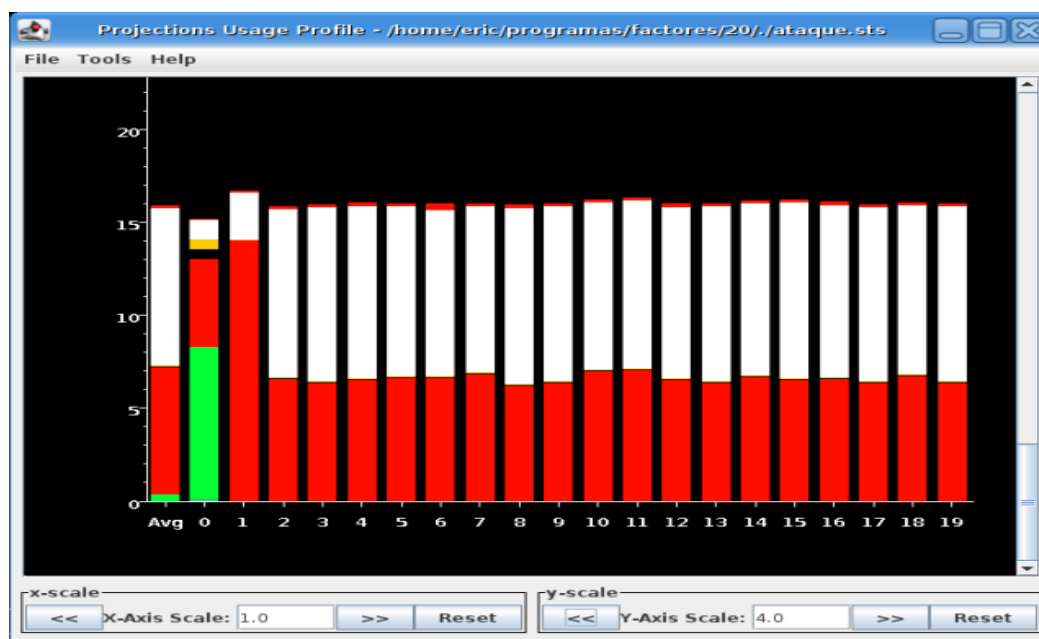


Fig. 5.70 Porcentaje de carga sobre cada procesador respecto a las funciones.

La siguiente figura muestra la cantidad de llamadas recibidas, tenemos en verde a la función que envía y en amarillo la que recibe la llamada, además de algunas estadísticas respecto a la cantidad de información recibida.



Fig. 5.71 Tabla de llamadas a funciones

Las siguientes figuras muestran de forma gráfica los mensajes enviados y recibidos, aquí la primer gráfica muestra los mensajes enviados de cualquier tipo, y vemos que el que mas mensajes envía es el procesador 0, la segunda gráfica es el total de mensajes recibidos por cada procesador y la última son los mensajes externos recibidos que a comparación de la segunda gráfica no aparece la sección roja, esto es por que el procesador 0 también se envía mensajes a si mismo para ejecutar tareas, la sección en verde corresponde a la llamada *main::no\_encontrados()*

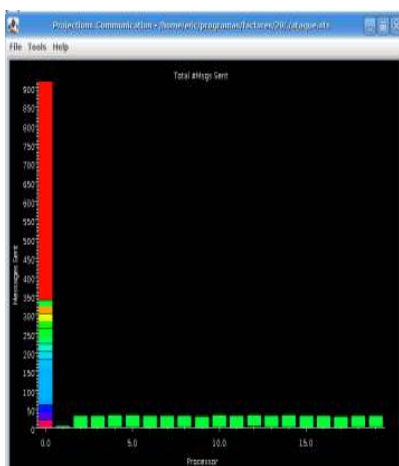


Fig. 5.72 Total de mensajes enviados

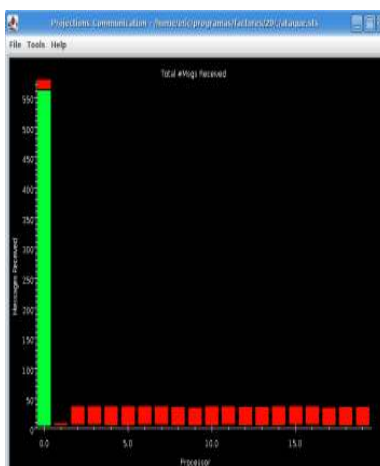


Fig. 5.73 Total de mensajes recibidos

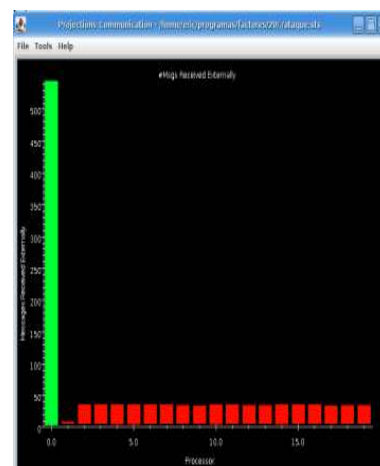


Fig. 5.74 Total de mensajes externos recibidos

La última gráfica nos muestra una comparación de la cantidad de mensajes enviados en la línea de tiempo, seguimos con la misma nomenclatura respecto a los mensajes, al iniciar es cuando existe la mayor parte de mensajes, ya que es la inicialización de todos los objetos y las llamadas a las funciones. Y en la ultima gráfica tenemos la animación de cómo van aumentando la carga de trabajo sobre los procesadores.

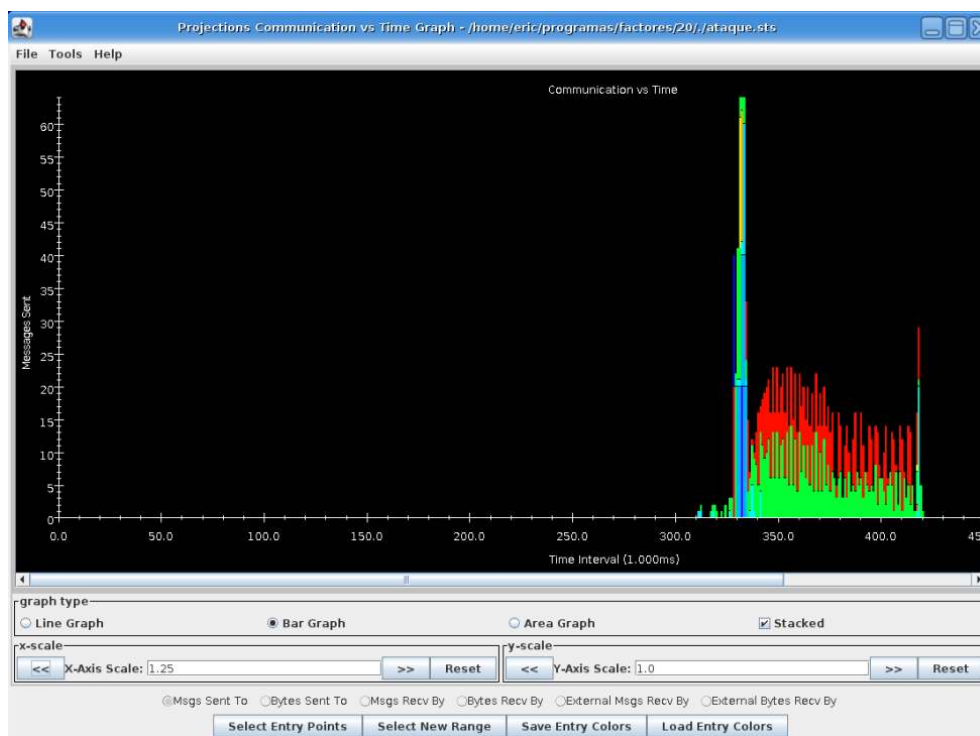


Fig. 5.75 Gráfica de comunicación contra tiempo.

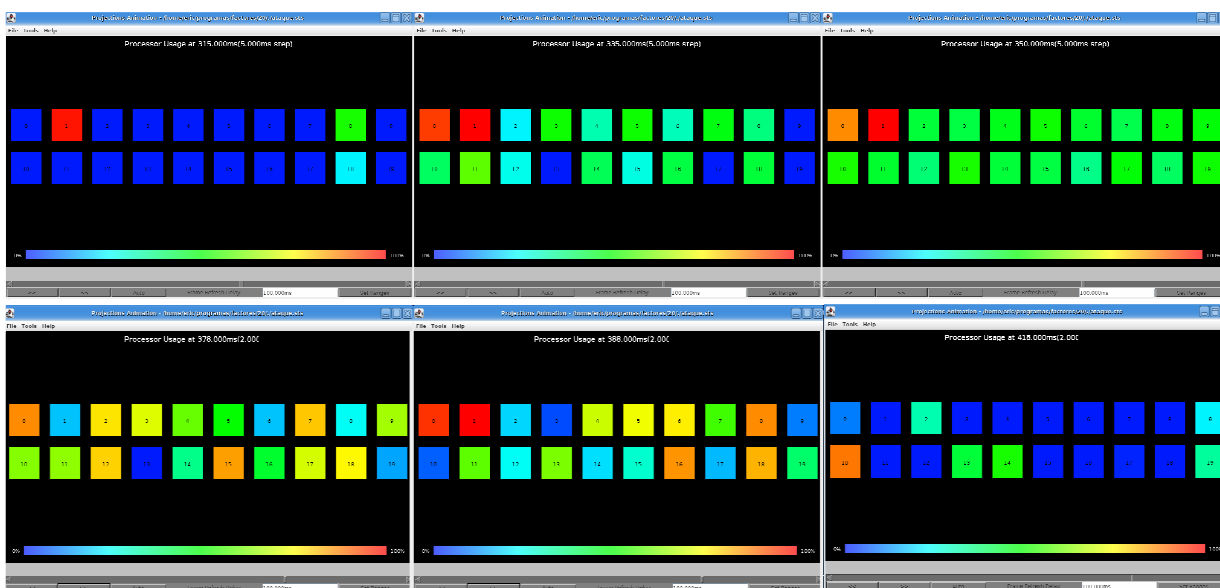


Fig. 5.76 Animación de la carga de los procesadores.

## 5.4 Buscar factores de números más grandes

Con esta aplicación solo nos es posible realizarlo para algunos valores, lo que sea posible para la variable ***long double*** debido a su precisión, así que usamos la librería ***gmp*** para poder obtener factores de números más grandes. La aplicación es exactamente la misma solo usamos variables enteras de gran precisión.

Lo que cambia respecto a las declaraciones anteriores es que utilizaremos cadenas como parámetros en las funciones, ***main::encontrados*** y ***factores::encuentra***, en lugar de ***long double*** en el archivo de interfaz.

```
mainmodule ataque {

    readonly CProxy_main mainProxy;
    readonly CProxy_primos fermat;

    mainchare main
    {
        entry main();
        entry [expedited] void encontrados(int proc, char factor1[100], char
            factor2[100]);
        entry [expedited] void no_encontrados(int proc, int indice);
    };

    array [1D] factores
    {
        entry void factores(void);
        entry [expedited] void termina();
        entry void encuentra(char a[100], char b[100], char c[100]);
    };

};
```

En este caso usaremos ***mpz\_t*** como tipo para un entero de gran precisión, ***mpz\_init (entero)***; para inicializarlo y ***mpz\_clear (entero)***; para liberar memoria. Para guardar los datos usamos cadenas, de las cuales obtendremos los valores con la función ***mpz\_set\_str (entero, cadena, base)***; donde base será 10 por que vamos a representar decimales. Con ***mpz\_sqrt (raiz, entero)***; obtenemos solamente la parte entera de la raíz cuadrada de un entero.

```
#include "ataque.decl.h"
#include "string.h"
#include "../libreria/include/gmp.h"

CProxy_main mainProxy;
CProxy_factores fermat;

class main : public CBase_main
{
```

---

```

private:
    unsigned long int e;
    double inicio, fin, tiempo;
    char N[100], raizN[100], aux1[100], aux2[100];
    int np;

public:

main(CkArgMsg* m)
{
    char *numero;

    mpz_t entero;
    mpz_t raiz;
    mpz_t a;
    mpz_t b;
    mpz_t aux;

    mpz_init (entero);
    mpz_init (raiz);
    mpz_init (a);
    mpz_init (b);
    mpz_init (aux);

    mainProxy = thisProxy;
    int i;

    if(m->argc < 1)
    {
        CkPrintf("Deben haber por lo menos un parámetro, numero a
factorizar .\n");
        CkAbort("Faltan parámetros");
    }

    inicio = CmiWallTimer(); //tomamos el tiempo en que se inicia

    np = CkNumPes();

    numero = m->argv[1];
    e = atoi(m->argv[2]);
    delete(m);

    mpz_set_str (entero, numero, 10);
    mpz_get_str(N, 10, entero);
    mpz_sqrt (raiz, entero);
    mpz_get_str(raizN, 10, raiz);

    CkPrintf("Numero a factorizar: %s \n", numero);
    CkPrintf("Raiz del numero: %s\n", raizN);

    CProxy_factores fermat = CProxy_factores::ckNew(np);

```

En esta parte con la función ***mpz\_set (valor1, valor2)***; asignamos valores entre variables, si deseamos asignar algún valor específico usamos ***mpz\_set\_ui(variable, valor\_numerico)*** ;

---

```

    mpz_set (a, raiz);          //aux1 = raizN;
    mpz_add (b, a, raiz);      // aux2 = aux1 + partInt;
    mpz_add_ui (a, a, 1);      //aux1 = raizN + 1;

    for(i=0; i < np; i++){

        //lanza las partes a calcular los primos que forman N
        //CkPrintf("tarea : %d ----Limite inferior : %Lf -----
        Limite superior : %Lf  \n",i,aux1,aux2);
        mpz_get_str(aux1, 10, a);
        mpz_get_str(aux2, 10, b);

        fermat[i].encuentra(aux1,aux2,N);

        mpz_add (a,a, raiz);    //aux1 += partInt;
        mpz_add (b,a, raiz);    //aux2 += partInt;

    }

    mpz_clear (entero);
    mpz_clear (raiz);
    mpz_clear (a);
    mpz_clear (b);
    mpz_clear (aux);
}

void encontrados(int proc, char factor1[100], char factor2[100]){

    fin = CmiWallTimer();
    tiempo = (fin-inicio);

    CkPrintf("Se encontraron el procesador : %d\nler. factor: %s  \n2o.
    factor: %s \n", proc, factor1,factor2);
    CkPrintf("En %f segundos \n", tiempo);
    CkExit();
}

void no_encontrados(int proc, int indice){
    mpz_t entero;
    mpz_t raiz;
    mpz_t a;
    mpz_t b;
    mpz_t aux;

    mpz_init (entero);
    mpz_init (raiz);
    mpz_init (a);
    mpz_init (b);
    mpz_init (aux);
    fin = CmiWallTimer();

```

---

```

    tiempo = (fin-inicio);

    CkPrintf("No se encontraron factores en el procesador %d \n",proc);
    CkPrintf("Del elemento %d del arreglo, después de %f segundos
    \n",indice, tiempo);

    mpz_set_str (entero, N, 10);
    mpz_set_str (a, aux1, 10);
    mpz_set_str (b, aux2, 10);
    mpz_set_str (raiz, raizN, 10);

    mpz_add (a, a, raiz); // aux1 = aux1 + raizN;

    mpz_set (aux, b);
    mpz_add (aux, aux, raiz);

    if (mpz_cmp (aux, entero ) > 0 ) { // (aux2+raizN) > N
        mpz_set_str (b, N, 10);
        mpz_sub_ui (b, b, 1); // aux2 = N-1;

        } else{
            mpz_add (b, b, raiz); //aux2 += raizN;
        }

    mpz_get_str(aux1, 10, a);
    mpz_get_str(aux2, 10, b);

    fermat[indice].encuentra(aux1, aux2, N);
    CkPrintf("Rango nuevo para la tarea: %d \n LI: %s -- LS: %s \n",
    indice,aux1,aux2);
    mpz_clear (entero);
    mpz_clear (raiz);
    mpz_clear (a);
    mpz_clear (b);
    mpz_clear (aux);
}

};

class factores : public CBase_factores
{
private:
    int esta;

public:

    factores(){
        esta = 0;
        fermat = thisProxy;
    }

    factores(CkMigrateMessage *m) {}

```

---

---

```

void termina(){
    esta = 1;
    //CkPrintf("Desde el procesador %d \n", CkMyPe());
}

int se_encontro(){
    if( esta == 0 )
        return 0;
    else
        return 1;
}

void encuentra(char a[100], char b[100], char c[100])
{

    mpz_t x;
    mpz_t y;
    mpz_t z;
    mpz_t i;
    mpz_t j;
    mpz_t aux1;
    mpz_t aux2;
    mpz_t modulo;

    mpz_init (x);
    mpz_init (y);
    mpz_init (z);
    mpz_init (i);
    mpz_init (j);
    mpz_init (aux1);
    mpz_init (aux2);
    mpz_init (modulo);

    char primo1[100]={0};
    char primo2[100]={0};
    int res = CkMyPe();

    CkPrintf("Procesador : %d ---- 1: %s ---- 2: %s ---- 3: %s \n",
res,a,b,c);

    mpz_set_str (j, b, 10);    // j = b
    mpz_set_str (x, a, 10);    //x = a
    mpz_set_str (x, a, 10);    // i = a
    mpz_set_str (z, c, 10);    // z = c
    mpz_pow_ui (aux1, x, 2);    // N = x^2
    mpz_sub (aux2, aux1, z);    // aux = x^2 - z
    mpz_set (y, aux2);

    do{
        if (se_encontro() != 0){
            CkExit();
            return;

```

---

```

    }

    mpz_sqrtrem (aux2, modulo,y);

    if (mpz_cmp_ui (modulo, 0) == 0)
    {
        fermat.termina();
        mpz_add (aux1, x, aux2); // factor1 = x + y;
        mpz_get_str (primo1, 10, aux1);

        mpz_sub (aux1, x, aux2); // factor2 = x - y;
        mpz_get_str (primo2, 10, aux1);

        mainProxy.encontrados(res, primo1,primo2);
        break;

    } else {

        mpz_add_ui (x, x, 1); //x++
        mpz_pow_ui (aux1, x, 2); // N = x^2
        mpz_sub (aux2, aux1, z); // aux = N - z
        mpz_set (y, aux2); // y = x^2 - z

    }

    mpz_add_ui (i, i, 1);

}while (mpz_cmp (i, j)< 0);

mpz_clear (x);
mpz_clear (y);
mpz_clear (z);
mpz_clear (i);
mpz_clear (j);
mpz_clear (aux1);
mpz_clear (aux2);
mpz_clear (modulo);

    mainProxy.no_encontrados(res,thisIndex);
}
};

#include "ataque.def.h"

```

Con esta librería ya es posible realizar búsquedas de factores a números de mas dígitos, ya que estábamos limitados a la precisión del tipo ***long double*** a aplicarlo con números de 13 a 15 dígitos aproximadamente. Ahora ejecutamos la aplicación con la librería para números de 15 hasta 24 dígitos, ya no se aplica a 25 dígitos por el tiempo que toma cada vez que aumentando el tamaño del número. En la siguiente tabla podemos apreciar un poco más que este método no es del todo eficiente ya que en 2 cifras con el mismo número de dígitos tienen tiempos muy diferentes, esto es debido que los números que son factores se encuentran mas alejados en una cifra respecto a la otra. En estos resultados se utilizaron 20 procesadores para el cálculo y los resultados en tiempos se elevan de gran forma que la gráfica de tiempo esta representada de forma logarítmica.

| Digitos | No. A factorizar         | Factores     |              | Tiempo en Seg | Procesador donde se encontro |
|---------|--------------------------|--------------|--------------|---------------|------------------------------|
|         |                          | Factor 1     | Factor 2     |               |                              |
| 15      | 349697112237601          | 24710753     | 14151617     | 0.403641      | 19                           |
| 16      | 6661221567265547         | 99999989     | 66612223     | 0.981897      | 18                           |
| 17      | 53323302814460117        | 749858947    | 71111111     | 96.610517     | 18                           |
| 18      | 138927726816123061       | 531667127    | 261305843    | 13.33087      | 19                           |
| 18      | 141810422852374057       | 749858947    | 189116131    | 53.960025     | 9                            |
| 19      | 5050248847763459303      | 3781378039   | 1335557777   | 174.186696    | 6                            |
| 20      | 60945641853771737983     | 9000000001   | 6771737983   | 47.533801     | 19                           |
| 21      | 277042680236381091449    | 27704267971  | 10000000019  | 1334.0814     | 6                            |
| 22      | 4791104951140977677777   | 77777677777  | 61600000001  | 318.403071    | 17                           |
| 22      | 1990000003480999999943   | 19899999997  | 10000000019  | 444.073339    | 8                            |
| 23      | 90298053599815111516031  | 55555555551  | 162536496481 | 37129.88429   | 1                            |
| 24      | 101111009999990000111111 | 999999000001 | 101111111111 | 143732.5622   | 12                           |

**Tabla 7** Tabla de tiempos y números que se utilizaron

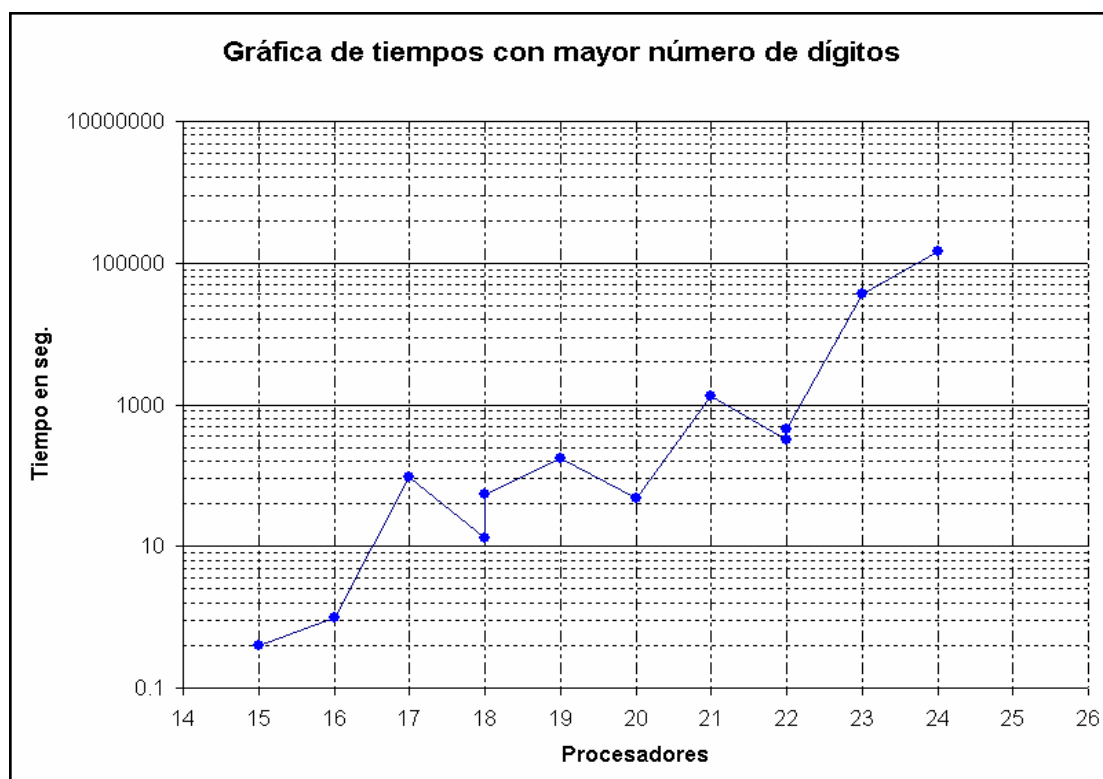


**Tabla 8** Gráfica de tiempos de ejecución expresada en logaritmo

# Conclusiones

En el presente documento se hace una breve explicación acerca de las arquitecturas de computadoras, cual es el diseño y su funcionamiento, así como las aplicaciones que se les ha dado para tener un panorama de lo que significa *físicamente* el poder utilizar varias computadoras para resolver problemas de gran magnitud.

Se da una revisión a las características y componentes principales que debe tener un cluster de computadoras, se presenta un poco de su historia y como han evolucionado a través de los años llegando incluso a utilizar una consola de videojuegos que desde otra perspectiva también puede considerarse como una computadora; se muestran algunas formas de clasificarlos y como estos han simplificado el tiempo en los cálculos, donde antes sólo eran posible usando supercomputadoras a las cuales no todos tienen acceso por el costo. Se hace un repaso a la lista *top500* la cual se actualiza 2 veces al año que nos muestra las computadoras más potentes del mundo, en esta lista se encuentran varios clusters de computadoras siendo uno de estos, la *Roadrunner*, el que ocupa la primera posición.

Se discute el concepto de sistema operativo distribuido sus características y programación para tener conocimiento de los complejo que llegan a ser estos sistemas al tratar de realizar tareas en conjunto, también se explica qué es un programa paralelo y en qué se diferencia de uno distribuido, además algunas recomendaciones de las características que debe contener un cluster para un funcionamiento eficiente.

Se da una explicación de las posibilidades de Charm++ como herramienta de programación paralela, las entidades que lo conforman, sus funciones más importantes y las facilidades que ofrece al programador el uso de este lenguaje, debido a la ventaja de que implica el no preocuparse de la forma en que los objetos o procesos van a comunicarse entre sí. Charm++ al estar orientado a objetos resulta más sencillo el poder interactuar con objetos que interactuar a las computadoras entre sí. Se expone lo que se conoce como el Sistema De Ejecución de Charm++ que es quien se encarga entre otras cosas del balance de carga y la distribución de los objetos a través de la red.

Se revisó la herramienta de visualización *Projections* que permite analizar como se comporta un programa a lo largo de su ejecución, además de obtener estadísticas y gráficas del rendimiento del procesador, los mensajes enviados a través de la red y las interacciones entre objetos y procesadores. También se comenta un depurador gráfico que permite analizar la ejecución de una aplicación a fin de encontrar alguna falla en el código y así hacer las correcciones necesarias. Estas dos herramientas pueden llegar a complementarse bien.

Finalmente se presentan tres aplicaciones donde se puede apreciar la utilidad del lenguaje charm++ y la facilidad que este ofrece para realizar programas paralelos. Este es un lenguaje que permite centrarnos en la repartición de tareas dejando a un lado la forma en que los procesos se comunicarán, incluso se tiene la posibilidad de no preocuparnos por el número de procesadores que utilicemos, además de permitir la portabilidad manteniendo el mismo código podremos ejecutar la misma aplicación en otro tipo de sistemas y en otro tipo de arquitectura. En la primera aplicación donde se usa un método probabilístico para realizar un cálculo sobre datos muy grandes, se puede ver como utilizar los arreglos de *chares* para realizar la misma tarea sobre todos los procesadores y se muestra que al ir aumentando el número de procesadores la aplicación requiere un tiempo menor para su ejecución. En la segunda aplicación se utiliza un método de los llamados de fuerza bruta, que es la Criba de Eratóstenes para encontrar los números que son primos en algún intervalo determinado, en este ejemplo mostramos el uso de los grupos *chare* para asignar tareas a los procesadores directamente en lugar de enviar un grupo de *chares* para que fueran repartidos en todos los procesadores en este ejemplo notamos un punto donde al aumentar el número de procesadores ya no mejora el tiempo de ejecución del programa, este punto es donde las comunicaciones, el acceso a disco y memoria ya no son lo suficiente rápidos y se ven rebasados por la velocidad de procesamiento. En el último ejemplo: la factorización de un entero, observamos la interacción entre objetos cuando el proceso de búsqueda en un rango termina sin éxito y solicita otro rango al procesador cero o master y sigue con su búsqueda; mediante la herramienta *Projections* observamos la asincronía en la respuesta, ya que el procesador cero también realiza tareas y atiende otras peticiones, así que el envío de un nuevo rango no será inmediato.

En general se presenta un documento para ofrecer una alternativa a MPI y PVM que son los lenguajes que se utilizan generalmente para aplicaciones paralelas, donde el programador puede centrarse de forma más libre a lo que es un algoritmo o una solución en paralelo o distribuida; y dejar la parte de comunicaciones al Sistema de Ejecución de Charm++.

## Trabajo a futuro

Este lenguaje de programación es posible aplicarlo directamente a problemas específicos o trabajos donde se venían utilizando otros lenguajes, ya no solo como demostración de las capacidades del mismo.

Debido a la transparencia que ofrece el lenguaje en lo que respecta a las comunicaciones entre los procesadores, es una posibilidad real el comparar las ventajas y desventajas del uso de Charm++ contra MPI, PVM o Java que son lenguajes mas conocidos, como una herramienta de aprendizaje en los cursos de Programación Paralela y Sistemas Distribuidos impartidos a nivel Licenciatura y Postgrado.

Estudiar la precisión numérica de Charm++ para poder realizar cálculos sin estar atados a la precisión de C++, así como la incorporación de otras bibliotecas: gráficas, visuales, numéricas, y otras que sean de ayuda en el desarrollo de nuevas aplicaciones.

# Apéndice A

## Instalación de Charm++

Para realizar la instalación de Charm++, en la carpeta donde este el paquete utilizamos el script build que tiene la siguiente sintaxis:

```
build <target> <version> [opciones ...] [--basedir=dir] [--libdir=dir] [--incdir=dir] [charmc-options ...]
```

si necesitamos información utilizamos el sufijo -h o -help para el script, es decir vamos a ejecutar:

```
./build -help
```

Los parámetros que son necesarios para la construcción del Sistema son los siguientes:

- **<target>** Especifica que vamos a compilar que en este caso será Charm++, con lo que se crean las librerías y los archivos ejecutables para poder utilizar Charm++. Otras opciones que existen son *AMPI* y *FEM* para utilizar librerías de *MPI*.
- **<version>** Con esta opción definimos que tipo de procesador, Sistema Operativo y Tipo de Comunicación utilizan las computadoras.

Los demás parámetros son opcionales y los describimos a continuación:

- **[options]** se proporciona información más detallada para la construcción como el compilador que se utilizará. El compilador por defecto es *gcc/g++*.
- **[--libdir=dir]** de esta forma se especifican las rutas de librerías adicionales para la construcción del sistema.
- **[--incdir=dir]** si es necesario alguna ruta donde hayan componentes adicionales se incluyen con esta opción.
- **[--basedir=dir]** de esta forma colocamos donde queremos que sea construido el Sistema, si se omite se creara dentro de la misma carpeta donde se compilara y llevara como nombre el tipo de sistema que se use como **<target>**.

A continuación tenemos una tabla que nos ayudara a elegir la opción que necesitemos colocar como *<version>*. Sin embargo, podemos usar *[opciones]* para especificar otro.

| Versión de Charm | Sistema       | Comunicación | Compilador por defecto |
|------------------|---------------|--------------|------------------------|
| net-linux        | PC Linux      | UDP/Myrinet  | Compilador GNU         |
| net-sol          | Solaris       | UDP          | Compilador GNU         |
| net-win32        | Win32         | UDP          | MS Visual C++          |
| net-cygwin       | Win32/cygwin  | UDP          | Compilador A/IX xIC    |
| mpi-sp           | IBM A/IX      | MPI          | Compilador SGI C++     |
| mpi-origin       | Origin2000    | MPI          | Compilador GNU C++     |
| net-ppc-darwin   | MacOS X       | UDP/Myrinet  | Compilador GNU         |
| net-linux-ia64   | IA64 Linux    | UDP          | Compilador GNU         |
| net-linux-amd64  | Opteron Linux | UDP          | Compilador GNU         |
| net-irix         | IRIX          | UDP          | Compilador GNU         |
| net-axp          | Alpha         | UDP          | Compilador GNU         |
| net-hp           | HP-UX         | UDP          | Compilador GNU         |
| mpi-linux        | PC Linux      | MPI          | Compilador GNU         |
| mpi-ppc-darwin   | MacOS X       | MPI          | Compilador GNU C++     |
| mpi-linux-ia64   | IA64 Linux    | MPI          | Compilador GNU         |
| mpi-linux-amd64  | Opteron Linux | MPI          | Compilador GNU         |
| mpi-axp          | Alpha         | MPI          | Compilador GNU         |
| mpi-linux-axp    | Alpha Linux   | MPI          | Compilador GNU         |
| origin2000       | Origin2000    | shared-mem   | Compilador SGI C++     |
| t3e              | Cray T3E      | shared-mem   | Compilador Cray C++    |

**Tabla A.9 Versiones a elegir para la construcción de un Sistema Cham++**

Esta tabla es una guía de las opciones más comunes pero puede construirse de otra manera para ello pueden utilizar la siguiente regla

Comunicación-Sistema Operativo Compilador

donde Comunicación puede ser alguna de las siguientes:

- **net-** Charm++ se comunica utilizando el protocolo TCP/IP, el cual trabaja sobre la mayoría de los sistemas pero regularmente es lento. Esta opción se utiliza para redes de estaciones de trabajo, clusters o solo una computadora.

- ***mpi-*** Se puede utilizar el sistema MPI sobre computadoras que tienen una buena implementación de este sistema como las ***Origin 2000***.
- ***exemplar, ncube-2, paragon-red, sp3, y t3e*** Este tipo de comunicación utiliza llamadas directas a través de primitivas de comunicación de la maquina.
- ***sim- y uth-*** Actualmente sin de uso en Charm++. Hay maquinas que usan ***uth-*** para simular procesadores como threads a nivel de usuario y ***sim-*** para cambiar entre procesadores y comunicaciones.

Sistema operativo se construye como:

- ***linux*** Para sistemas basados en Linux
- ***win32*** Para sistemas MS Windows con MS Visual C++ compiler
- ***cygwin*** Para sistemas MS Windows con Cygnus (Cygwin Unix layer)
- ***irix*** Para sistemas SGI IRIX
- ***origin*** Para sistemas SGI Origin 2000 IRIX
- ***sol*** Para sistemas Solaris
- ***sun*** Para sistemas SunOS
- ***rs6k*** Para sistemas IBM R/S 6000 A/IX
- ***sp*** Para sistemas IBM SP A/IX
- ***hp*** Para sistemas Hewlett-Packard HP-UX
- ***axp*** Para sistemas DEC Alpha DECUNIX

Recientemente ha aparecido nuevos sistemas debido al cambio del hardware pero basados en los anteriores como:

- ***-x86*** Para sistemas Solaris pero con hardware x86 en lugar de Sun.
- ***-axp*** Para sistemas Linux, con hardware Alpha en lugar de PC x86.
- ***-ia64*** Para hardware basado en Itanium.
- ***-amd64*** Para hardware basado en AMD Opteron.

Para el compilador tenemos la siguiente lista de opciones:

- ***gcc3*** para usar el compilador GNU GCC/G++ versión 3.
- ***acc*** para usar el compilador HP aC++.
- ***cc*** para compiladores Sun WorkShop C++.
- ***cc64*** para compiladores de 64 bits Sun WorkShop C++ o IBM xLC.
- ***cxx*** para compilador DIGITAL C++.
- ***kcc*** para compilador KAI C++.
- ***pgcc*** para compilador Portland Group's C++.
- ***icc*** compilador Intel C/C++ para Linux IA32.
- ***ecc*** compilador Intel C/C++ para Linux IA64.
- ***mpcc*** compilador SUN Solaris C++ para MPI.
- ***scyld*** para sistemas Beowulf Scyld basados en bproc.
- ***clustermatic*** para sistemas Clustermatic Beowulf cluster basados en bproc.
- ***gm*** para librerías Myrinet GM de MyriCom.
- ***vmi*** para tener soporte de librerías VMI de NCSA.
- ***tcp*** versión para redes que usen comunicación UDP. Con esta opción cambiamos la comunicación a TCP.
- ***smp*** Habilita el soporte para maquinas SMP. Agiliza la comunicación en maquinas que tienen físicamente más de un procesador.
- ***bluegene*** Compila Charm++ emulando que esta sobre una maquina Blue Gene.

Con la opción ***help*** se muestran las opciones anteriores para una determinada versión o arquitectura. Por ejemplo para el caso que usemos ***net-linux*** obtendremos lo siguiente:

```
> ./build charm++ net-linux help
supported options: gcc3 gm icc kcc pgcc scyld smp bluegene tcp
```

# Apéndice B

## Ejecutando un programa

La ejecución de una aplicación en **Charm++** es similar a una en **C** o **C++** con la diferencia que utilizaremos el programa **charmrun** para poder realizar la ejecución en paralelo, de lo contrario solamente se ejecutara en el procesador local o desde donde se realice la orden. Por ejemplo si queremos ejecutar un programa **pgm** en 2 procesadores realizaremos los siguientes:

```
> ./charmrun ./pgm argumentos +p2
```

**Charmrun** es utilizado como vinculo entre la aplicación en **C++** y el Sistema de Ejecución de **Charm++** que es el que se encarga de repartir los procesos y migrarlos entre los procesadores disponibles. En el caso de usar **MPI** funciona como interfaz para inicializar **mpirun**. Cuando realiza la ejecución en red se utiliza **rsh** o **ssh** para invocar los procesos remotos. Se debe definir el archivo **nodelist** que es donde estará la lista de las computadoras que se utilizaran, en caso de no utilizar se tomara el archivo **~/.nodelist** en el **home** que esta configurado con **localhost** así que solo se ejecutará localmente. Un ejemplo del archivo seria el siguiente:

```
group main ++shell /bin/ssh  
host <nombre_maquina>
```

para ejecutar un programa con otro archivo **nodelist** a la orden de ejecución se le agrega **+nodelist archivo\\_nodelist**, así se pueden tener varios archivos y utilizar uno en especifico con las maquinas que se deseen. Por ejemplo la instrucción de ejecución quedaría de la siguiente forma:

```
> ./charmrun ./pgm argumentos +p2 +nodelist archivo\_nodelist
```

La interfaz remota por defecto es **rsh**, pero se puede definir otro diferente por ejemplo **ssh** usando la orden **++shell**. Cualquier método que se utilice hay que tener en cuenta que

debe ser posible conectarse a otro equipo sin contraseña para que pueda ejecutarse la aplicación en paralelo

Una aplicación hecha con **Charm++** tiene la posibilidad de ejecutarse en un solo equipo aunque se haya definido el archivo **nodelist** con los equipos que se utilizaran, esto puede ayudar a realizar pruebas antes de lanzar la aplicación sobre todos los equipos. Para realizar esto al momento de ejecutar la aplicación agregamos después del nombre del programa la orden **++local** de la siguiente forma:

```
./charmrun ++local ./pgm 12 100 +p2
```

# Bibliografía

[1] **"High Performance Cluster Computing: Architectures and Systems Vol. 1"**,

Rajkumar Buyya,

Prentice Hall PTR, NJ, USA, 1999.

[2] Manual de referencia:

**"The Charm++ Programming Language Manual Version 5.8"**,

Parallel Programming Laboratory University of Illinois at Urbana-Champaign

Dept. Of Computer Science, University of Illinois,USA

[3] Manual de referencia:

**"Charm++ Debugger Manual version 0.2"**

Filippo Gioachin, Rashmi Jyothi, Orion Lawlor, Eric Bohm.

Parallel Programming Laboratory University of Illinois at Urbana-Champaign

Dept. Of Computer Science, University of Illinois,USA

[4] Manual de referencia:

**"Projections Manual"**

By Mike DeNardo, Sid Cammeresi, Theckla Louchios, Orion Lawlor, Gengbin Zheng, Chee

Wai Lee, Isaac Dooley, and Sindhura Bandhakavi

Dept. of Computer Science, University of Illinois,USA

[5] Artículo:

**"A portable Concurrent Object Oriented System Based On C++ "**

Laxmikant V. Kale, Sanjeev Krishnan

Parallel Programming Laboratory University of Illinois at Urbana-Champaign

1993

[6] Artículo:

**"Debugging Support for Charm++ "**

Rashmi Jyothi, Orion Sky Lawlor, L. V. Kalé

Department of Computer Science, University of Illinois at Urbana-Champaign.

PADTAD Workshop for IPDPS 2004, IEEE Press,

2004, p. 294.

[6] Manual de referencia:

**"Beowulf HOWTO"**

Jacek Radajewski and Douglas Eadline, v1.1.1, November 1998.

- 
- [7] Reporte técnico de la Supercomputadora CICESE2000:  
**"Metodologías de Paralelización en la Supercomputadora CICESE2000"**  
 Areli García López, Julián J. Delgado y Salvador Castañeda,  
 Departamento de Cómputo Dirección de Telemática Centro de Investigación Científica y de  
 Educación Superior de Ensenada, México, Febrero 2002
- [8] Proyecto de construcción de un cluster:  
**"Cluster Heterogéneo De Computadoras"**  
 Emilio José Plaza Nieto, enero de 2002
- [9] Notas de clustering:  
**"Introducción a las tecnologías de Clustering en GNU/Linux"**  
 Rosa María Yáñez Gómez, 2005
- [10] Notas de programación distribuida:  
**"Introducción a la programación distribuida"**  
 Diego J. Bodas Sagi, CES Felipe II (UCM), 2003.
- [11] Notas:  
**"Criptografía y Seguridad en Computadores"**  
 Cuarta Edición. Versión 0.7.53, Manuel Lucena, Universidad de Jaén.  
 Libro electrónico gratuito disponible en  
<http://wwwdi.ujaen.es/~mlucena/wiki/pmwiki.php?n=Main.LCripto>
- [12] **"Método de Montecarlo Segunda Edición 1983"**  
 I.M. Sóbol  
 Editorial MIR Moscú
- [13] **"Sistemas Operativos Distribuidos"**  
 Adrew S. Tanenbaum  
 Prentice Hall Hispanoamericana S.A.  
 1996
- [14] Tesis  
**"Factorización LU de matrices dispersas en multiprocesadores"**  
 Rafael Asenjo Plaza  
 Universidad de Málaga (España) en 1997
- [15] **"High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI"**  
 Joseph D. Sloan  
 O'Reilly, November 2004
- [16] **"Notas de Programación Distribuida y Paralela"**  
 Depto. de Lenguajes y Sistemas Informáticos Universidad de Granada.  
 Profesor de teoría y prácticas: José Miguel Mantas Ruiz  
<http://lsi.ugr.es/~jmantas/pdp/pdp.html>
-

[17] **“Parallel and Distributed Simulation Systems”**

Richard M. Fujimoto  
Ed. John Wiley & Sons, Inc  
EUA, 2000

[19] Página Web:

**www.top500.org**

TOP500 Supercomputing Sites  
Ranking of supercomputers according to the LINPACK benchmark  
2009

[20] Artículo:

**“Diez años de computación con clusters”**

José Luís Gordillo Ruiz  
Revista online: Enter@te en línea, Internet Cómputo y Telecomunicaciones  
www.enterate.unam.mx  
Computo Académico UNAM  
Universidad Nacional Autónoma de México  
Núm. 27, Marzo de 2004

[21] Página Web:

**http://www.neoteo.com/un-superordenador-hecho-con-consolas-ps3.neo**

Un superordenador hecho con consolas PS3  
Constanza Villanueva  
31 de julio de 2007

[22] Página Web:

**http://en.wikipedia.org/wiki/IBM\_Roadrunner**

[23] Página Web:

Página Sun Constellation Linux Cluster: Ranger  
**http://www.tacc.utexas.edu/resources/hpc/**

[24] Página Web:

JÜLICH SUPERCOMPUTING CENTRE (JSC)  
**http://www.fz-juelich.de/jsc/juropa/**

[25] Página Web:

Cluster Aitzaloea  
**http://supercomputo.izt.uam.mx/index.php?id=Cluster\_Aitzaloea**

[26] Página Web:

**“Introducción a los sistemas distribuidos”**

J. Oscar Rojo 2003  
**http://www.augcyl.org/?q=glol-intro-sistemas-distribuidos**

[27] **“Tools and Environments for Parallel and Distributed Computing”**

S. Hariri, M. Parashar

Wiley Series on Parallel and Distributed Computing

John Wiley & Sons, Inc. USA 2004

[28] **“Parallel Metaheuristics: A New Class of Algorithms”**

Enrique Alba

Wiley-Interscience

USA 2005

[29] **“The Mathematics Of Ciphers”**

Number Theory and RSA Cryptography

S.C. Coutinho

A.K. Peters Natick Massachusetts

USA, 1997